Design Concepts in Programming Languages

Franklyn Turbak and David Gifford

Version created on November 23, 2004 at 3:53.
This is a draft. Please do not cite.
Send all bugs, feedback, etc. to fturbak@wellesley.edu.

Contents

Preface

Acknowledgments

This book owes its existence to many people. We are grateful to the following individuals for their contributions:

- Both as an early teaching assistant for the MIT course (6.821 Programming Languages) upon which this book is based and as a technical editor during the final push to turn the course notes into a book, Mark Sheldon made innumerable contributions to the content and form of the book. Mark also played a key role in the development of the material on data, pattern matching, and abstract types and created the index for the book.

- Jonathan Rees profoundly influenced the content of this book while he was a 6.821 teaching assistant. Many of the mini-languages, examples, exercises, and software implementations, as well as some of the sections of text, had their origins with Jonathan. Jonathan was also the author of an early data type and pattern matching facility used in course software that strongly influenced the facilities described in the book.

- Brian Reistad and Trevor Jim greatly improved the quality of the book. As 6.821 teaching assistants, they unearthed and fixed innumerable bugs, improved the presentation and content of the material, and created many new exercises. Brian also played a major role in implementing software for testing the mini-languages in the book.

- In addition to his contributions as a 6.821 teaching assistant, Alex Salcianu also collected and edited homework and exam problems from fifteen years of the course for inclusion in the book.

- Valuable contributions and improvements to this book were made by other 6.821 teaching assistants: Alexandra Andersson, Michael (Ziggy) Blair,

Barbara Cutler, Joshua Glazer, Robert Grimm, Alex Hartemink, David Huynh, Eddie Kohler, Gary Leavens, Ravi Nanavati, Jim O'Toole, Dennis Quan, Alex Snoeren, Patrick Sobalvarro, Peter Szilagyi, Bienvenido Velez-Rivera, Earl Waldin, and Qian Wang.

- In Fall 2002, Michael Ernst taught 6.821 based on an earlier version of this book, and his detailed comments resulted in many improvements.

- Based on teaching 6.821 at MIT and using the course materials at Hong Kong University and at Georgia Tech, Olin Shivers has made many excellent suggestions on how to improve the content and presentation of the material.

- While using the course materials at other universities, Gary Leavens, Andrew Myers, Randy Osborne, and Kathy Yelick provided helpful feedback.

- Early versions of the pragmatics system were written by Doug Grundman, with major extensions by Raymie Stata and Brian Reistad.

- Pierre Jouvelot did the lion's share of the implementation of FX (a language upon which early versions of 6.821 were based) with some help from Mark Sheldon and Jim O'Toole.

- Guillermo Rozas taught us many nifty pragmatics tricks. Our pragmatics coverage is heavily influenced by his source-to-source front end to the MIT Scheme compiler.

- David Espinosa introduced us to embedded interpreters and helped us to improve our presentation of dynamic semantics.

- Ken Moody provided helpful feedback on the course material, especially on the PostFix Equivalence Theorem.

- Numerous 6.821 students have improved this book in various ways, from correcting bugs to suggesting major reorganizations. In this regard, we are especially grateful to: Atul Adya, Kavita Bala, Ron Bodkin, Philip Bogle, Miguel Castro, Anna Chefter, Natalya Cohen, Richard Davis, Andre deHon, Michael Frank, Robert Grimm, Yevgeny Gurevich, Viktor Kuncak, Mark Lillibridge, Andrew Myers, Michael Noakes, John Pezaris, Matt Power, Roberto Segala, Mark Torrance, and Carl Witty.

- Jue Wang uncovered numerous typos and inconsistencies in her careful proofreading of a late draft of the book.

- Special thanks go to Jeanne Darling, who has been the 6.821 course administrator for over ten years. Her administrative, editing, and technical skills, as well as her can-do spirit and cheerful demeanor, were critical in keeping both the course and the book project afloat.

Chapter 1

Introduction

Order and simplification are the first steps toward the mastery of a subject — the actual enemy is the unknown.

— The Magic Mountain, Thomas Mann

1.1 Programming Languages

Programming is a great load of fun. As you have no doubt experienced, clarity and simplicity are the keys to good programming. When you have a tangle of code that is difficult to understand, your confidence in its behavior wavers, and the code is no longer any fun to read or update.

Designing a new programming language is a kind of meta-level programming activity that is just as much fun as programming in a regular language (if not more so). You will discover that clarity and simplicity are even more important in language design than they are in ordinary programming. Today hundreds of programming languages are in use — whether they be scripting languages for Internet commerce, user interface programming tools, spreadsheet macros, or page format specification languages that when executed can produce formatted documents. Inspired application design often requires a programmer to provide a new programming language or to extend an existing one. This is because flexible and extensible applications need to provide some sort of programming capability to their end users.

Elements of programming language design are even found in "ordinary" programming. For instance, consider designing the interface to a collection data

structure. What is a good way to encapsulate an iteration idiom over the elements of such a collection? The issues faced in this problem are similar to those in adding a looping construct to a programming language.

The goal of this book is to teach you the great ideas in programming languages in a simple framework that strips them of complexity. You will learn several ways to specify the meaning of programming language constructs and will see that small changes in these specifications can have dramatic consequences for program behavior. You will explore many dimensions of the programming language design space, study decisions to be made along each dimension, and consider how decisions from different dimensions can interact. We will teach you about a wide variety of neat tricks for extending programing languages with interesting features like undoable state changes, exitable loops, pattern matching, and multitasking. Our approach for teaching you this material is based on the premise that when language behaviors become incredibly complex, the descriptions of the behaviors must be incredibly simple. It is the only hope.

1.2 Syntax, Semantics, and Pragmatics

Programming languages are traditionally viewed in terms of three facets:

1. **Syntax** — the *form* of programming languages.
2. **Semantics** — the *meaning* of programming languages.
3. **Pragmatics** — the *implementation* of programming languages.

Here we briefly describe these facets.

Syntax

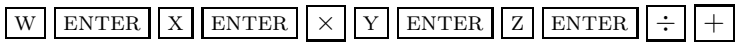
Syntax focuses on the concrete notations used to encode programming language phrases. Consider a phrase that indicates the sum of the product of w and x and the quotient of y and z. Such a phrase can be written in many different notations: as a traditional mathematical expression

$$wx + y/z$$

or as a Lisp parenthesized prefix expression

```
(+ (* w x) (/ y z))
```

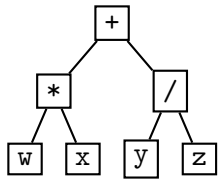
or as a sequence of keystrokes on a postfix calculator



or as a layout of cells and formulae in a spreadsheet

	1	2	3	4
A	`w=`		`w*x =`	*A2 * B2*
B	`x=`		`y/z =`	*C2 / D2*
C	`y=`		`ans =`	*A4 + B4*
D	`z=`			

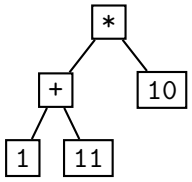
or as a graphical tree



Although these concrete notations are superficially different, they all designate the same abstract phrase structure (the sum of a product and a quotient). The syntax of a programming language specifies which concrete notations (strings of characters, lines on a page) in the language are legal and which tree-shaped abstract phrase structure is denoted by each legal notation.

Semantics

Semantics specifies the mapping between the structure of a programming language phrase and what the phrase means. Such phrases have no inherent meaning: their meaning is only determined in the context of a system for interpreting their structure. For example, consider the following expression tree:



Suppose we interpret the nodes labeled `1`, `10`, and `11` as the usual decimal notation for numbers, and the nodes labeled `+` and `*` as the sum and product of the values of their subnodes. Then the root of the tree stands for $(1+11)\cdot 10 = 120$. But there are many other possible meanings for this tree. If `*` stands for exponentiation rather than multiplication, the meaning of the tree could be 12^{10}. If the numerals are in binary notation rather than decimal notation, the tree could stand for (in decimal notation) $(1+3)\cdot 2 = 8$. Alternatively, `1` and `11` might represent the set of odd integers, `10` might represent the set of even

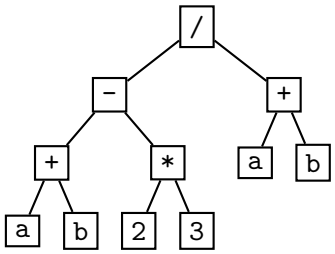
integers, and `+` and `*` might represent addition and multiplication on integer sets; in this case, the meaning of the tree would be the set of even integers. Perhaps the tree does not indicate an evaluation at all, and only stands for a property intrinsic to the tree, such as its height (3), its number of nodes (5), or its shape (perhaps it describes a simple corporate hierarchy). Or maybe the tree is an arbitrary encoding of a particular object of interest, such as a rock or a book.

This example illustrates how a single program phrase can have many possible meanings. Semantics describes the relationship between the abstract structure of a phrase and its meaning.

Pragmatics

Whereas semantics deals with *what* a phrase means, pragmatics focuses on the details of *how* that meaning is computed. Of particular interest is the effective use of various resources, such as time, space, and access to shared physical devices (storage devices, network connections, video monitors, printers, etc.).

As a simple example of pragmatics, consider the evaluation of the following expression tree (under the first semantic interpretation described above):



Suppose that `a` and `b` stand for particular numeric values. Because the phrase `(+ a b)` appears twice, a naïve evaluation strategy will compute the same sum twice. An alternative strategy is to compute the sum once, save the result, and use the saved result the next time the phrase is encountered. The alternative strategy does not change the meaning of the program, but does change its use of resources; it reduces the number of additions performed, but may require extra storage for the saved result. Is the alternative strategy better? The answer depends on the details of the evaluation model and the relative importance of time and space.

Another potential improvement in the example is the phrase `(* 2 3)`, which always stands for the number 6. If the sample expression is to be evaluated many times (for different values of `a` and `b`), it may be worthwhile to replace `(* 2 3)`

by `6` to avoid unnecessary multiplications. Again, this is a purely pragmatic concern that does not change the meaning of the expression.

1.3 Goals

The goals of this book are to explore the semantics of a comprehensive set of programming language design idioms, show how they can be combined into complete practical programming languages, and discuss the interplay between semantics and pragmatics. Except for establishing a few syntactic conventions at the outset, we won't say much about syntax at all. We will introduce a number of tools for describing the semantics of programming languages, and will use these tools to build intuitions about programming language features and study many of the dimensions along which languages can vary. Our coverage of pragmatics is mainly at a high level: we will study some simple programming language implementation techniques and program improvement strategies rather than focus on squeezing the last ounce of performance out of a particular computer architecture.

We will discuss programming language features in the context of several **mini-languages**. Each of these is a simple language that captures the essential features of a class of existing programming languages. In many cases, the mini-languages are so pared down that they are hardly suitable for serious programming activities. Nevertheless, these languages embody all of the key ideas in programming languages. Their simplicity saves us from getting bogged down in needless complexity in our explorations of semantics and pragmatics. And like good modular building blocks, the components of the mini-languages are designed to be "snapped together" to create practical languages.

1.4 PostFix: A Simple Stack Language

We will introduce the tools for syntax, semantics, and pragmatics in the context of a mini-language called PostFix. PostFix is a simple stack-based language inspired by the PostScript graphics language, the Forth programming language, and Hewlett Packard calculators. Here we give an informal introduction to PostFix in order to build some intuitions about the language. In subsequent chapters, we will introduce tools that allow us to study PostFix in more depth.

1.4.1 Syntax

The basic syntactic unit of a PostFix program is the **command**. Commands are of the following form:

- Any integer numeral. E.g., `17`, `0`, `-3`.
- One of the following special command tokens: `add`, `div`, `eq`, `exec`, `gt`, `lt`, `mul`, `nget`, `pop`, `rem`, `sel`, `sub`, `swap`.
- An **executable sequence** — a single command that serves as a subroutine. It is written as a parenthesized list of subcommands separated by whitespace.[1] E.g., `(7 add 3 swap)` and `(2 (5 mul) exec add)`.

Since executable sequences contain other commands (including other executable sequences), they can be arbitrarily nested. An executable sequence counts as a single command despite its hierarchical structure.

A PostFix **program** is a parenthesized sequence consisting of (1) the token `postfix` followed by (2) a natural number (i.e., non-negative integer) indicating the number of program parameters followed by (3) zero or more PostFix commands. For example, here are some sample PostFix programs:

```
(postfix 0 4 7 sub)

(postfix 2 add 2 div)

(postfix 4 4 nget 5 nget mul mul swap 4 nget mul add add)

(postfix 1 ((3 nget swap exec) (2 mul swap exec) swap)
           (5 sub) swap exec exec)
```

In PostFix, as in all the languages we'll be studying, all parentheses are required and none are optional. Moving parentheses around changes the structure of the program and most likely changes its behavior. Thus, while the following PostFix executable sequences use the same numerals and command tokens in the same order, they are distinguished by their parenthesization, which, as we shall see below, makes them behave differently.

```
((1) (2 3 4) swap exec)
((1 2) (3 4) swap exec)
((1 2) (3 4 swap) exec)
```

[1] Whitespace is any contiguous sequence of characters that leave no mark on the page, such as spaces, tabs, and newlines.

1.4.2 Semantics

The meaning of a PostFix program is determined by executing its commands in left to right order. Each command manipulates an implicit stack of values that initially contains the integer arguments of the program (where the first argument is at the top of the stack and the last argument is at the bottom). A value on the stack is either (1) an integer numeral or (2) an executable sequence. The result of a program is the integer value at the top of the stack after its command sequence has been completely executed. A program signals an error if (1) the final stack is empty, (2) the value at the top of the final stack is not an integer, or (3) an inappropriate stack of values is encountered when one of its commands is executed.

The behavior of PostFix commands is summarized in Figure 1.1. Each command is specified in terms of how it manipulates the implicit stack. We use the notation $P \xrightarrow{args} v$ to mean that executing the PostFix program P on the integer argument sequence $args$ returns the value v. The notation $P \xrightarrow{args}$ `error` means that executing the PostFix program P on the arguments signals an error. Errors are caused by inappropriate stack values or an insufficient number of stack values. In practice, it is desirable for an implementation to indicate the type of error. We will use comments (delimited by squiggly braces) to explain errors and other situations.

To illustrate the meanings of various commands, we show the results of some simple program executions. For example, numerals are pushed onto the stack, while `pop` and `swap` are the usual stack operations.

`(postfix 0 1 2 3)` $\xrightarrow{[]}$ 3 {*Only the top stack value is returned.*}
`(postfix 0 1 2 3 pop)` $\xrightarrow{[]}$ 2
`(postfix 0 1 2 swap 3 pop)` $\xrightarrow{[]}$ 1
`(postfix 0 1 swap)` $\xrightarrow{[]}$ `error` {*Not enough values to swap.*}
`(postfix 0 1 pop pop)` $\xrightarrow{[]}$ `error` {*Empty stack on second pop.*}

Program arguments are pushed onto the stack (from last to first) before the execution of the program commands.

`(postfix 2)` $\xrightarrow{[3,4]}$ 3 {*Initial stack has 3 on top with 4 below.*}
`(postfix 2 swap)` $\xrightarrow{[3,4]}$ 4
`(postfix 3 pop swap)` $\xrightarrow{[3,4,5]}$ 5

It is an error if the actual number of arguments does not match the number of parameters specified in the program.

`(postfix 2 swap)` $\xrightarrow{[3]}$ `error` {*Wrong number of arguments.*}
`(postfix 1 swap)` $\xrightarrow{[3]}$ `error` {*Not enough values to swap.*}

- N : Push the numeral N onto the stack.
- `sub` : Call the top stack value v_1 and the next-to-top stack value v_2. Pop these two values off the stack and push the result of $v_2 - v_1$ onto the stack. If there are fewer than two values on the stack or the top two values aren't both numerals, signal an error. The other binary arithmetic operators — `add` (addition), `mul` (multiplication), `div` (integer division[a]) and `rem` (remainder of integer division) — behave similarly. Both `div` and `rem` signal an error if v_1 is zero.
- `lt` : Call the top stack value v_1 and the next-to-top stack value v_2. Pop these two values off the stack. If $v_2 < v_1$, then push a `1` (a true value) on the stack, otherwise push a `0` (false). The other binary comparison operators — `eq` (equals) and `gt` (greater than) — behave similarly. If there are fewer than two values on the stack or the top two values aren't both numerals, signal an error.
- `pop` : Pop the top element off the stack and discard it. Signal an error if the stack is empty.
- `swap` : Swap the top two elements of the stack. Signal an error if the stack has fewer than two values.
- `sel` : Call the top three stack values (from top down) v_1, v_2, and v_3. Pop these three values off the stack. If v_3 is the numeral `0`, push v_1 onto the stack; if v_3 is a non-zero numeral, push v_2 onto the stack. Signal an error if the stack does not contain three values, or if v_3 is not a numeral.
- `nget` : Call the top stack value v_{index} and the remaining stack values (from top down) v_1, v_2, ..., v_n. Pop v_{index} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$ and v_i is a numeral, push v_i onto the stack. Signal an error if the stack does not contain at least one value, if v_{index} is not a numeral, if i is not in the range $[1, n]$, or if v_i is not a numeral.
- `(`C_1 ... C_n`)` : Push the *executable sequence* `(`C_1 ... C_n`)` as a single value onto the stack. Executable sequences are used in conjunction with `exec`.
- `exec` : Pop the executable sequence from the top of the stack, and prepend its component commands onto the sequence of currently executing commands. Signal an error if the stack is empty or the top stack value isn't an executable sequence.

[a]The integer division of n and d returns the integer quotient q such that $n = qd + r$, where r (the remainder) is such that $0 \leq r < |d|$ if $n \geq 0$ and $-|d| < r \leq 0$ if $n < 0$.

Figure 1.1: English semantics of PostFix commands.

Note that program arguments must be integers — they cannot be executable sequences.

Numerical operations are expressed in postfix notation, in which each operator comes after the commands that compute its operands. `add`, `sub`, `mul`, and `div` are binary integer operators. `lt`, `eq`, and `gt` are binary integer predicates returning either `1` (true) or `0` (false).

`(postfix 1 4 sub)` $\xrightarrow{[3]}$ -1
`(postfix 1 4 add 5 mul 6 sub 7 div)` $\xrightarrow{[3]}$ 4
`(postfix 5 add mul sub swap div)` $\xrightarrow{[7,6,5,4,3]}$ -20
`(postfix 3 4000 swap pop add)` $\xrightarrow{[300,20,1]}$ 4020
`(postfix 2 add 2 div)` $\xrightarrow{[3,7]}$ 5 {*An averaging program.*}
`(postfix 1 3 div)` $\xrightarrow{[17]}$ 5
`(postfix 1 3 rem)` $\xrightarrow{[17]}$ 2
`(postfix 1 4 lt)` $\xrightarrow{[3]}$ 1
`(postfix 1 4 lt)` $\xrightarrow{[5]}$ 0
`(postfix 1 4 lt 10 add)` $\xrightarrow{[3]}$ 11
`(postfix 1 4 mul add)` $\xrightarrow{[3]}$ `error` {*Not enough numbers to add.*}
`(postfix 2 4 sub div)` $\xrightarrow{[4,5]}$ `error` {*Divide by zero.*}

In all the above examples, each stack value is used at most once. Sometimes it is desirable to use a number two or more times or to access a number that is not near the top of the stack. The `nget` command is useful in these situations; it puts at the top of the stack a copy of a number located on the stack at a specified index. The index is 1-based, from the top of the stack down, not counting the index value itself.

`(postfix 2 1 nget)` $\xrightarrow{[4,5]}$ 4 {*4 is at index 1, 5 at index 2.*}
`(postfix 2 2 nget)` $\xrightarrow{[4,5]}$ 5

It is an error to use an index that is out of bounds or to access a non-numeric stack value (i.e., an executable sequence) with `nget`.

`(postfix 2 3 nget)` $\xrightarrow{[4,5]}$ `error` {*Index 3 is too large.*}
`(postfix 2 0 nget)` $\xrightarrow{[4,5]}$ `error` {*Index 0 is too small.*}
`(postfix 1 (2 mul) 2 nget)` $\xrightarrow{[3]}$ `error` {*Value at index 2 is not a number.*}

The `nget` command is particularly helpful for expressing numerical programs, where it is common to reference arbitrary parameter values and use them multiple times.

`(postfix 1 1 nget mul)` $\xrightarrow{[5]}$ 25 {*A squaring program.*}

`(postfix 4 4 nget 5 nget mul mul swap 4 nget mul add add)` $\xrightarrow{[3,4,5,2]}$ 25
{*Given a, b, c, x, calculates* $ax^2 + bx + c$.}

As illustrated in the last example, the index of a given value increases every time a new value is pushed on the stack.

Executable sequences are compound commands like `(2 mul)` that are pushed onto the stack as a single value. They can be executed later by the `exec` command. Executable sequences act like subroutines in other languages; execution of an executable sequence is similar to a subroutine call, except that transmission of arguments and results is accomplished via the stack.

`(postfix 1 (2 mul) exec)` $\xrightarrow{[7]}$ `14` {`(2 mul)` *is a doubling subroutine.*}

`(postfix 0 (0 swap sub) 7 swap exec)` $\xrightarrow{[]}$ -7
{`(0 swap sub)` *is a negation subroutine.*}

`(postfix 0 (7 swap exec) (0 swap sub) swap exec)` $\xrightarrow{[]}$ -7

`(postfix 0 (2 mul))` $\xrightarrow{[]}$ `error` {*Final top of stack is not an integer.*}

`(postfix 0 3 (2 mul) gt)` $\xrightarrow{[]}$ `error`
{*Executable sequence where number expected.*}

`(postfix 0 3 exec)` $\xrightarrow{[]}$ `error` {*Number where executable sequence expected.*}

```
(postfix 1 ((3 nget swap exec) (2 mul swap exec) swap)
           (5 sub) swap exec exec)
```
$\xrightarrow{[7]}$ 9
{*Given n, calculates 2n - 5.*}

The last example illustrates that evaluations involving executable sequences can be rather contorted.

The `sel` command selects between two values based on a test value, where zero is treated as false and any non-zero integer is treated as true. It can be used in conjunction with `exec` to conditionally execute one of two executable sequences.

`(postfix 1 2 3 sel)` $\xrightarrow{[1]}$ 2

`(postfix 1 2 3 sel)` $\xrightarrow{[0]}$ 3

`(postfix 1 2 3 sel)` $\xrightarrow{[17]}$ 2 {*Any non-zero number is "true".*}

`(postfix 0 (2 mul) 3 4 sel)` $\xrightarrow{[]}$ `error` {*Test not a number.*}

`(postfix 4 lt (add) (mul) sel exec)` $\xrightarrow{[3,4,5,6]}$ 30

`(postfix 4 lt (add) (mul) sel exec)` $\xrightarrow{[4,3,5,6]}$ 11

`(postfix 1 1 nget 0 lt (0 swap sub) () sel exec)` $\xrightarrow{[-7]}$ 7
{*An absolute value program.*}

`(postfix 1 1 nget 0 lt (0 swap sub) () sel exec)` $\xrightarrow{[6]}$ 6

▷ **Exercise 1.1** Determine the value of the following PostFix programs on an empty stack.

a. `(postfix 0 10 (swap 2 mul sub) 1 swap exec)`

b. `(postfix 0 (5 (2 mul) exec) 3 swap)`

c. `(postfix 0 (() exec) exec)`

d. `(postfix 0 2 3 1 add mul sel)`

e. `(postfix 0 2 3 1 (add) (mul) sel)`

f. `(postfix 0 2 3 1 (add) (mul) sel exec)`

g. `(postfix 0 0 (2 3 add) 4 sel exec)`

h. `(postfix 0 1 (2 3 add) 4 sel exec)`

i. `(postfix 0 (5 6 lt) (2 3 add) 4 sel exec)`

j. `(postfix 0 (swap exec swap exec) (1 sub) swap (2 mul)`
`swap 3 swap exec)` ◁

▷ **Exercise 1.2** Write executable sequences that compute the following logical operations. Recall that `0` is false and all other numerals are treated as true.

a. *not*: return the logical negation of a single argument.

b. *and*: given two numeric arguments, return `1` if their logical conjunction is true, and `0` otherwise.

c. *short-circuit-and*: return 0 if the first argument is false; otherwise return the second argument.

d. Demonstrate the difference between *and* and *short-circuit-and* by writing a PostFix program that has a different result if *and* is replaced by *short-circuit-and*. ◁

▷ **Exercise 1.3**

a. Without `nget`, is it possible to write a PostFix program that squares its single argument? If so, write it; if not, explain.

b. Is it possible to write a PostFix program that takes three integers and returns the smallest of the three? If so, write it; if not, explain.

c. Is it possible to write a PostFix program that calculates the factorial of its single argument (assume it's non-negative)? If so, write it; if not, explain. ◁

1.4.3 The Pitfalls of Informal Descriptions

The "by-example" and English descriptions of PostFix given above are typical of the way that programming languages are described in manuals, textbooks, courses, and conversations. That is, a syntax for the language is presented, and the semantics of each of the language constructs is specified using English prose and examples. The utility of this method for specifying semantics is apparent from the fact that the vast majority of programmers learn to read and write programs via this approach.

But there are many situations in which informal descriptions of programming languages are inadequate. Suppose that we want to improve a program by substituting one phrase for another throughout the program. How can we be sure that the substitution preserves the meaning of the program?

Or suppose that we want to prove that the language as a whole has a particular property. For instance, it turns out that every PostFix program is guaranteed to terminate (i.e., a PostFix program cannot enter an infinite loop). How would we go about proving this property based on the informal description? Natural language does not provide any rigorous framework for reasoning about programs or programming languages. Without the aid of some formal reasoning tools, we can only give hand-waving arguments that are not likely to be very convincing.

Or suppose that we wish to extend PostFix with features that make it easier to use. For example, it would be nice to name values, to collect values into arrays, to query the user for input, and to loop over sequences of values. With each new feature, the specification of the language becomes more complex, and it becomes more difficult to reason about the interaction between various features. We'd like techniques that help to highlight which features are orthogonal and which can interact in subtle ways.

Or suppose that a software vendor wants to develop PostFix into a product that runs on several different machines. The vendor wants any given PostFix program to have exactly the same behavior on all of the supported machines. But how do the development teams for the different machines guarantee that they're all implementing the "same" language? If there are any ambiguities in the PostFix specification that they're implementing, different development teams might resolve the ambiguity in incompatible ways. What's needed in this case is an unambiguous specification of the language as well as a means of proving that an implementation meets that specification.

The problem with informal descriptions of a programming language is that they're neither concise nor precise enough for these kinds of situations. English is often verbose, and even relatively simple ideas can be unduly complicated

to explain. Moreover, it's easy for the writer of an informal specification to underspecify a language by forgetting to cover all the special cases (e.g., error situations in PostFix). It isn't that covering all the special cases is impossible; it's just that the natural language framework doesn't help much in pointing out what the special cases are.

It is possible to overspecify a language in English as well. Consider the PostFix programming model introduced above. The current state of a program is captured in two entities: the stack and the current command sequence. To programmers and implementers alike, this might imply that a language implementation *must* have explicit stack and command sequence elements in it. Although these would indeed appear in a straightforward implementation, they are not in any way *required*; there are alternative models and implementations for PostFix (see Exercise 3.12). It would be desirable to have a more abstract definition of what constitutes a legal PostFix implementation so that a would-be implementer could be sure that an implementation was faithful to the language definition regardless of the representations and algorithms employed.

In the remaining chapters of the first segment of this book, we introduce a number of tools that address the inadequacies outlined above. First, in Chapter 2 we present **s-expression grammars**, a simple specification for syntax that we will use to describe the structure of all of the mini-languages we explore. Then, using PostFix as our object of study, we introduce two approaches to formal semantics:

- An **operational semantics** (Chapter 3) explains the meaning of programming language constructs in terms of the step-by-step process of an abstract machine.
- A **denotational semantics** (Chapter 4) explains the meaning of programming language constructs in terms of the meaning of their subparts.

These approaches support the unambiguous specification of programming languages and provide a framework in which to reason about properties of programs and languages. This segment concludes in Chapter 5 with a presentation of a technique for determining the meaning of recursive specifications.

Throughout the book, mathematical concepts are formalized in terms of the **metalanguage** described in Appendix A. Readers are encouraged to familiarize themselves with this language by skimming the appendix early on and later referring to it in more detail on an "as needed" basis.

While we will emphasize formal tools throughout this book, we do not imply that formal tools are a panacea or that formal approaches are superior to informal ones in an absolute sense. In fact, informal explanations of language

features are usually the simplest way to learn about a language. In addition, it's very easy for formal approaches to get out of control, to the point where they are overly obscure, or require too much mathematical machinery to be of any practical use on a day-to-day basis. For this reason, we won't dwell on nitty gritty formal details and won't cover material as a dry sequence of definitions, theorems, and proofs. Instead, our goal is to show that the *concepts* underlying the formal approaches are indispensable for understanding particular programming languages as well as the dimensions of language design. The tools introduced in this segment should be in any serious computer scientist's bag of tricks.

Reading

No single book can entirely cover the broad area of programming languages. We recommend the following books for other perspectives of the field:

- Mitchell has authored two relevant books: [Mit96] is a mathematical exploration of programming language semantics based on a series of typed lambda calculi, while [Mit03] discusses the dimensions of programming languages in the context of many modern programming languages.
- Friedman, Wand, and Haynes [FWH01] uses interpreters and translators written in SCHEME to study essential programming language features in the context of some mini-languages.
- Reynolds [Rey98] gives a theoretical treatment of many programming language features.
- Gelernter and Jaganathan [GJ90] discusses a number of popular programming languages in a historical perspective and compare them in terms of expressiveness.
- MacLennan's text [Mac99] stands out as one of the few books on programming languages to enumerate a set of principles and then analyze popular languages in terms of these principles.
- Kamin [Kam90] uses interpreters written in PASCAL to analyze the core features of several popular languages.
- Marcotty and Ledgard [ML86] cover a wide range of programming language features and paradigms by presenting a sequence of mini-languages.
- Gunter [Gun92] provides an in-depth overview of formal programming language semantics.

- Winskel [Win93] presents a mathematical introduction to formal programming language semantics.
- Horowitz [Hor95] has collected an excellent set of classic papers on the design of programming languages that every programming language designer should be familiar with.

Chapter 2

Syntax

since feeling is first
who pays any attention
to the syntax of things
will never wholly kiss you;
...
for life's not a paragraph

And death i think is no parenthesis

— *e e cummings*

In the area of programming languages, syntax refers to the form of programs — how they are constructed from symbolic parts. A number of theoretical and practical tools — including grammars, lexical analyzers, and parsers — have been developed to aid in the study of syntax. By and large we will downplay syntactic issues and tools. Instead, we will emphasize the semantics of programs; we will study the meaning of language constructs rather than their form.

We are not claiming that syntactic issues and tools are unimportant in the analysis, design, and implementation of programming languages. In actual programming language implementations, syntactic issues are very important and a number of standard tools (like Lex and Yacc) are available for addressing them. But we do believe that syntax has traditionally garnered much more than its fair share of attention, largely because its problems were more amenable to solution with familiar tools. This state of affairs is reminiscent of the popular tale of the person who searches all night long under a street lamp for a lost item not because the item was lost there but because the light was better. Luckily, many investigators strayed away from the street lamp of parsing theory in order

to explore the much dimmer area of semantics. Along the way, they developed many new tools for understanding semantics, some of which we will focus on in later chapters.

Despite our emphasis on semantics, however, we can't ignore syntax completely. Programs must be expressed in *some* form, preferably one that elucidates the fundamental structure of the program and is easy to read, write, and reason about. In this chapter, we introduce a set of syntactic conventions for describing our mini-languages.

2.1 Abstract Syntax

We will motivate various syntactic issues in the context of EL, a mini-language of expressions. EL describes functions that map any number of numerical inputs to a single numerical output. Such a language might be useful on a calculator, say, for automating the evaluation of commonly used mathematical formulae.

Figure 2.1 describes (in English) the abstract structure of a legal EL program. EL programs contain numerical expressions, where a numerical expression can be constructed out of various kinds of components. Some of the components, like numerals, references to input values, and various kinds of operators, are **primitive** — they cannot be broken down into subparts.[1] Other components are **compound** — they are constructed out of constituent components. The components have names; e.g., the subparts of an arithmetic operation are the **rator** (short for "operator") and two **rands**, (short for "operands") while the subparts of the conditional expression are the **test**, the **consequent**, and the **alternate**.

There are three major classes of phrases in an EL program: whole programs that designate calculations on a given number of inputs, numerical expressions that designate numbers, and boolean expressions that designate truth values (i.e., true or false). The structural description in Figure 2.1 constrains the ways in which these expressions may be "wired together". For instance, the test component of a conditional must be a boolean expression, while the consequent and alternate components must be numerical expressions.

A specification of the allowed wiring patterns for the syntactic entities of a language is called a **grammar**. Figure 2.1 is said to be an **abstract grammar** because it specifies the logical structure of the syntax but does not give any indication how individual expressions in the language are actually written.

The structure determined by an abstract grammar for an individual program phrase can be represented by an **abstract syntax tree (AST)**. Consider an EL

[1] Numerals can be broken down into digits, but we will ignore this detail.

A legal EL program is a pair of (1) a *numargs* numeral specifying the number of parameters and (2) a *body* that is a *numerical expression*, where a numerical expression is either:

- an *intlit* — an integer numeral *num*;
- an *input* — a reference to one of the program inputs specified by an *index* numeral.
- an *arithmetic operation* — an application of a *rator*, in this case a binary *arithmetic operator*, to two numerical *rand* expressions, where an arithmetic operator is either
  - addition,
  - subtraction,
  - multiplication,
  - division,
  - remainder;
- a *conditional expression* — a choice between numerical *consequent* and *alternate* expressions determined by a boolean *test* expression, where a *boolean expression* is either
  - a *boollit* — a boolean literal *bool*;
  - a *relational operation* — an application of *rator*, in this case a binary *relational operator*, to two numerical *rand* expressions, where a relational operator is one of
    - less-than,
    - equal-to,
    - greater-than;
  - a *logical operation* — an application of a *rator*, in this case a binary *logical operator*, to two boolean *rand* expressions, where a logical operator is one of
    - and,
    - or.

Figure 2.1: An abstract grammar for EL programs.

program that returns zero if its first input is between 1 and 10 (exclusive) and otherwise returns the product of the second and third inputs. The abstract syntax tree for this program appears in Figure 2.2. Each node of the tree corresponds to a numerical or boolean expression. The leaves of the tree stand for primitive phrases, while the intermediate nodes represent compound phrases. The labeled edges from a parent node to its children show the relationship between a compound phrase and its components. The AST is defined purely in terms of these relationships; the particular way that the nodes and edges of a tree are arranged on the page is immaterial.

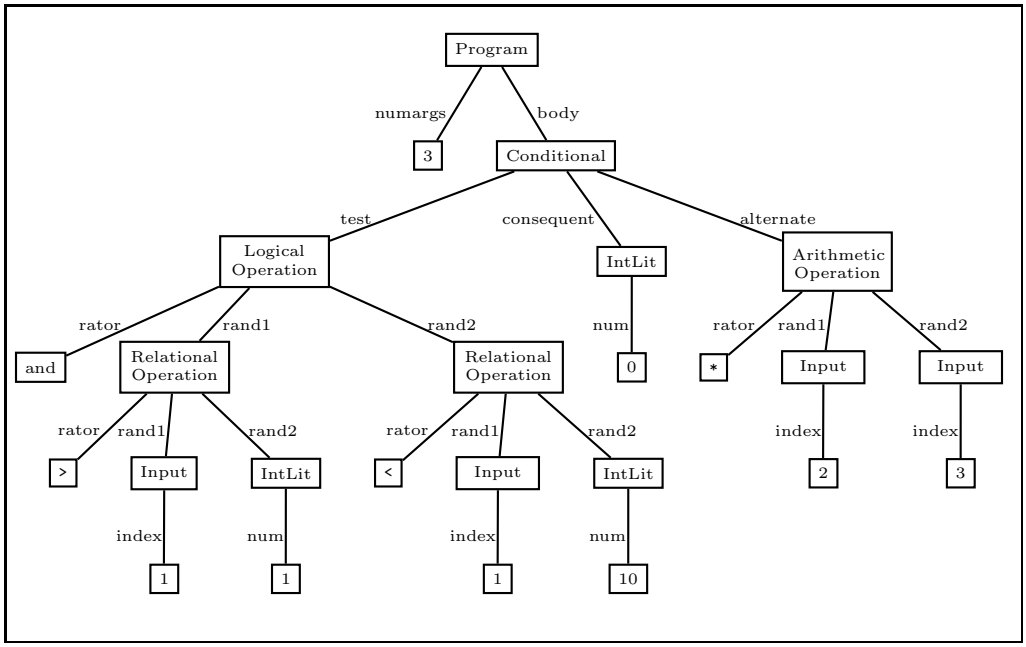


Figure 2.2: An abstract syntax tree for an EL program.

2.2 Concrete Syntax

Abstract grammars and ASTs aren't very helpful when it comes to representing programs in a textual fashion.[2] The same abstract structure can be expressed in

[2]It is also possible to represent programs more pictorially, and visual programming languages are an active area of research. But textual representations enjoy certain advantages over visual ones: they tend to be more compact than visual representations; the technology for processing them and communicating them is well-established; and, most importantly, they can effectively make use of our familiarity with natural language.

many different concrete forms. The sample EL conditional expression considered above, for instance, could be written down in some strikingly different textual forms. Here are three examples:

- `if $1 > 1 && $1 < 10 then 0 else $2 * $3 endif`
- ```
  (cond ((and (> (arg 1) 1) (< (arg 1) 10))
         0)
        (else (* (arg 2) (arg 3))))
  ```
- `1 input 1 gt 1 input 10 lt and {0} {2 input 3 input mul} choose`

The above forms differ along a variety of dimensions:

- *Keywords and operation names.* The keywords `if`, `cond`, and `choose` all indicate a conditional expression, while multiplication is represented by the names `*` and `mul`. Accessing the ith input to the program is written in three different ways: `$`i, `(arg` i`)`, and i `input`.
- *Operand order.* The example forms use infix, prefix, and postfix operations, respectively.
- *Means of grouping.* Grouping can be determined by precedence (`&&` has a lower precedence than `>` and `<` in the first example), keywords (`then`, `else`, and `endif` delimit the test, consequent, and alternate of the first conditional), or explicit matched delimiter pairs (such as the parentheses and braces in the last two examples).

These are only some of the possible dimensions; many more are imaginable. For instance, numbers could be written in many different numeral formats: e.g., decimal, binary, or octal numerals, scientific notation, or even roman numerals!

The above examples illustrate that the nature of concrete syntax necessitates making representational choices that are arbitrary with respect to the abstract syntactic structure. These choices are explicitly encoded in a **concrete grammar** that specifies how to **parse** a linear text string into a **concrete syntax tree (CST)**. A concrete syntax tree has the structural relationships of an abstract syntax tree embedded within it, but it is complicated by the handling of details needed to make the textual layout readable and unambiguous.

2.3 S-Expression Grammars Specify ASTs

While we will dispense with many of the complexities of concrete syntax, we still need *some* concrete notation for representing abstract syntax trees. Such a representation should be simple, yet permit us to precisely describe abstract
  ```

syntax trees and operations on such trees. Throughout this book, we need to operate on abstract syntax trees to determine the meaning of a phrase, the type of a phrase, the translation of a phrase, and so on. To perform such operations, we need a far more compact representation for abstract syntax trees than the English description in Figure 2.1 or the graphical one in Figure 2.2.

We have chosen to represent abstract syntax trees using **s-expression grammars**. An s-expression grammar unites LISP's fully parenthesized prefix notation with traditional grammar notations to describe the structure of abstract syntax trees via parenthesized sequences of symbols and meta-variables. Not only are these grammars very flexible for defininining unambiguous program language syntax, but it is easy to construct programs that process s-expression notation. This facilitates writing interpreters and compilers for the mini-languages we will study.

2.3.1 S-Expressions

An **s-expression** (short for **symbolic expression**) is a LISP notation for representing trees by parenthesized linear text strings. The leaves of the trees are **symbolic tokens**, where (to first approximation) a symbolic token is any sequence of characters that does not contain a left parenthesis (‘`(`’), a right parenthesis (‘`)`’), or a whitespace character. Examples of symbolic tokens include `x`, `foo`, `this-is-a-token`, `17`, `6.821`, and `4/3*pi*r^2`.[3]

An intermediate node in a tree is represented by a pair of parentheses surrounding the s-expressions that represent the subtrees. Thus, the s-expression

```
((this is) an ((example) (s-expression tree)))
```

designates the structure depicted in Figure 2.3. Whitespace is necessary for separating tokens that appear next to each other, but can be used liberally to enhance the readability of the structure. Thus, the above s-expression could also be written as

```
((this is)
 an
 ((example)
  (s-expression
   tree)))
```

without changing the structure of the tree.

[3] We always write s-expressions in `teletype-font`.

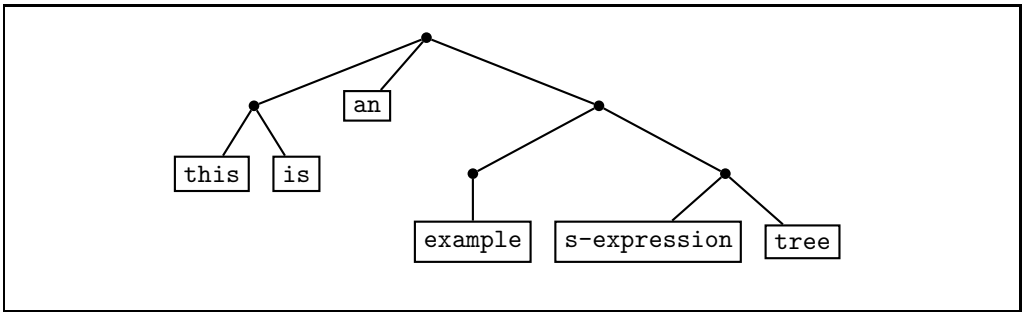


Figure 2.3: Viewing `((this is) an ((example) (s-expression tree)))` as a tree.

2.3.2 The Structure of S-Expression Grammars

An s-expression grammar combines the domain notation of Appendix A with s-expressions to specify the syntactic structure of a language. It has two parts:

1. A listing of **syntactic domains**, one for each kind of phrase.
2. A set of **production rules** that define the structure of compound phrases.

Figure 2.4 presents a sample s-expression grammar for EL.

A syntactic domain is a class of program phrases. **Primitive syntactic domains** are collections of phrases with no substructure. The primitive syntactic domains of EL are Intlit, BooleanLiteral, ArithmeticOperator, RelationalOperator, and LogicalOperator. Primitive syntactic domains are specified by an enumeration of their elements or by an informal description with examples. For instance, the details of what constitutes a numeral in EL are pretty much left to the reader's intuition.

Compound syntactic domains are collections of phrases built out of other phrases. Because compound syntactic domains are defined by a grammar's production rules, the syntactic domain listing does not explicitly indicate their structure. All syntactic domains are annotated with domain variables (such as *NE*, *BE*, and *N*) that range over their elements; these play an important role in the production rules.

The production rules specify the structure of compound domains. There is one rule for each compound domain. A production rule has the form

domain-variable ::= `pattern` [*phrase-type*]
| `pattern` [*phrase-type*]
...
| `pattern` [*phrase-type*]

where

Syntactic Domains:

$P \in$ Program
$NE \in$ NumExp
$BE \in$ BoolExp
$N \in$ Intlit $= \{\texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$B \in$ BooleanLiteral $= \{\texttt{true}, \texttt{false}\}$
$A \in$ ArithmeticOperator $= \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$
$R \in$ RelationalOperator $= \{\texttt{<}, \texttt{=}, \texttt{>}\}$
$L \in$ LogicalOperator $= \{\texttt{and}, \texttt{or}\}$

Production Rules:

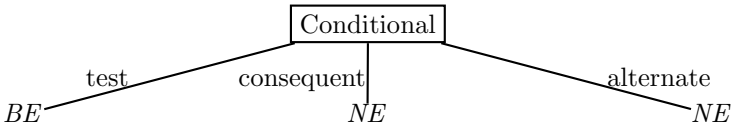
P	::= `(el` $N_{numargs}$ NE_{body}`)`	[Program]
NE	::= N_{num}	[IntLit]
	| `(arg` N_{index}`)`	[Input]
	| `(`A_{rator} NE_{rand1} NE_{rand2}`)`	[Arithmetic Operation]
	| `(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)`	[Conditional]
BE_{bool}	::= B	[BoolLit]
	| `(`R_{rator} NE_{rand1} NE_{rand2}`)`	[Relational Operation]
	| `(`L_{rator} BE_{rand1} BE_{rand2}`)`	[Logical Operation]

Figure 2.4: An s-expression grammar for EL.

- *domain-variable* is the domain variable for the compound syntactic domain being defined,
- *pattern* is an s-expression pattern (defined below), and
- *phrase-type* is a mnemonic name for the subclass of phrases in the domain that match the pattern. It corresponds to the labels of intermediate nodes in an AST.

Each line of the rule is called a **production**; it specifies a collection of phrases that are considered to belong to the compound syntactic domain being defined. The second production rule in Figure 2.4, for instance, has four productions specifying that a NumExp can be an integer literal, an indexed input, an arithmetic operation, or a conditional.

An s-expression pattern appearing in a production stands for the domain of all s-expressions that have the form of the pattern. S-expression patterns are like s-expressions except that domain variables may appear as tokens. For example, the pattern `(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)` contains the domain variables BE_{test}, $NE_{consequent}$, and $NE_{alternate}$. Such a pattern specifies the structure of a **compound phrase** — a phrase that is built from other phrases. Subscripts on the domain variables indicate their role in the phrase. This helps to distinguish positions within a phrase that have the same domain variable — e.g., the consequent and alternate of a conditional, which are both numerical expressions. This subscript appears as an edge label in the AST node corresponding to the pattern, while the phrase type of the production appears as the node label. So the `if` pattern denotes an AST node pattern of the form:



An s-expression pattern P is said to **match** an s-expression SX if P's domain variables d_1, ..., d_n can be replaced by matching s-expressions SX_1, ..., SX_n to yield SX. Each SX_i must be an element of the domain over which d_i ranges. A compound syntactic domain contains exactly those s-expressions that match the patterns of its productions in an s-expression grammar.

For example, Figure 2.5 shows the steps by which the NumExp production

`(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)`

matches the s-expression

```
(if (= (arg 1) 3) (arg 2) 4).
```

Matching is a recursive process: BE_{test} matches `(= (arg 1) 3)`, $NE_{consequent}$ matches `(arg 2)`, and $NE_{alternate}$ matches `4`. The recursion bottoms out at primitive syntactic domain elements (in this case, elements of the domain Intlit). Figure 2.5 shows how an AST for the sample `if` expression is constructed as the recursive matching process backs out of the recursion.

Note that the pattern `(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)` would not match any of the s-expressions `(if 1 2 3)`, `(if (arg 2) 2 3)`, or `(if (+ (arg 1) 1) 2 3)`, because none of the test expressions `1`, `(arg 2)`, or `(+ (arg 1) 1)` match any of the patterns in the productions for BoolExp.

More formally, the rules for matching an s-expression pattern to an s-expression are as follows:

- A symbolic token T in the pattern matches only T.
- A domain variable for a primitive syntactic domain D matches an s-expression SX only if SX is an element of D.
- A domain variable for a compound syntactic domain D matches an s-expression SX only if one of the patterns in the rule for D matches SX.
- A pattern `(`P_1 $\ldots P_n$`)` matches an s-expression `(`SX_1 $\ldots SX_n$`)` only if each subpattern P_i matches the corresponding subexpression SX_i.

We shall use the notation *s-exp*$_D$ to designate the domain element in D that an s-expression designates. When D is a compound domain, *s-exp*$_D$ corresponds to an abstract syntax tree that indicates *how* *s-exp* matches one of the rule patterns for the domain. For example,

`(if (and (> (arg 1) 1) (< (arg 1) 10)) 0 (* (arg 2) (arg 3)))`$_{\text{NumExp}}$

can be viewed as the abstract syntax tree depicted in Figure 2.2 on page 20. Each node of the AST indicates the production that successfully matches the corresponding s-expression, and each edge indicates a domain variable that appeared in the production pattern. The nodes are labeled by the phrase type of the production and the edges are labeled by the subscript names of the domain variables used in the production pattern.

In the notation *s-exp*$_D$, domain subscript D serves to disambiguate cases where *s-exp* belongs to more than one syntactic domain. For example, `1`$_{\text{Intlit}}$ is `1` as a primitive numeral, while `1`$_{\text{NumExp}}$ is `1` as a numerical expression. The subscript will be omitted when the domain is clear from context.

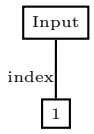
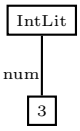
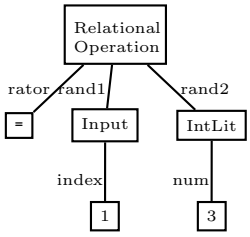
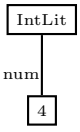
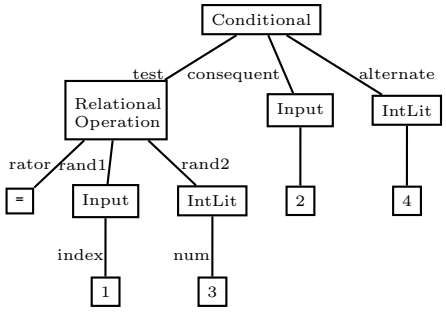
s-expression	domain	production	AST
`(arg 1)`	*NE*	`(arg` N_{index}`)`	
`3`	*NE*	N_{num}	
`(= (arg 1) 3)`	*BE*	`(`R_{rator} NE_{rand1} NE_{rand2}`)`	
`(arg 2)`	*NE*	`(arg` N_{index}`)`	
`4`	*NE*	N_{num}	
`(if (= arg 1) (arg 2) 4)`	*NE*	`(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)`	

Figure 2.5: The steps by which `(if (= (arg 1) 3) (arg 2) 4)` is determined to be a member of the syntactic domain NumExp. In each row, an s-expression matches a domain by a production to yield an abstract syntax tree.

2.3.3 Phrase Tags

S-expression grammars for our mini-languages will generally follow the Lisp-style convention that compound phrases begin with a **phrase tag** that unambiguously indicates the phrase type. In EL, `if` is an example of a phrase tag. The fact that all compound phrases are delimited by explicit parentheses eliminates the need for syntactic keywords in the middle of or at the end of phrases (e.g., `then`, `else`, and `endif` in a conditional).

Because phrase tags can sometimes be cumbersome, we will often omit them when no ambiguity results. Figure 2.6 shows an alternative syntax for EL in which every production is marked with a distinct phrase tag. In this alternative syntax, the addition of 1 and 2 would be written `(arith + (num 1) (num 2))` — quite a bit more verbose than `(+ 1 2)`! But most of the phrase tags can be removed without introducing ambiguity. Because numerals are clearly distinguished from other s-expressions, there is no need for the `num` tag. Likewise, we can dispense with the `bool` tag. Since the arithmetic operators are disjoint from the other operators, the `arith` tag is superfluous; similarly for the `rel` and `log` tags. The result of these optimizations is the original EL syntax in Figure 2.4.

P	::= (`el` $N_{numargs}$ NE_{body})	[Program]
NE	::= (`num` N_{num})	[IntLit]
	| (`arg` N_{index})	[Input]
	| (`arith` A_{rator} NE_{rand1} NE_{rand2})	[Arithmetic Operation]
	| (`if` BE_{test} $NE_{consequent}$ $NE_{alternate}$)	[Conditional]
BE	::= (`bool` B)	[Truth Value]
	| (`rel` R_{rator} NE_{rand1} NE_{rand2})	[Relational Operation]
	| (`log` L_{rator} BE_{rand1} BE_{rand2})	[Logical Operation]

Figure 2.6: An alternative syntax for EL in which every production has a phrase tag.

2.3.4 Sequence Patterns

As defined above, each component of an s-expression pattern matches only s-expressions. But sometimes it is desirable for a pattern component to match *sequences* of s-expressions. For example, suppose we want to extend the `+` operator of EL to accept an arbitrary number of numeric operands (making `(+ 1 2 3 4)` and `(+ 2 (+ 3 4 5) (+ 6 7))` legal numerical expressions in EL). Using the

simple patterns introduced above, this extension requires an infinite number of productions:

NE ::= ...
| `(+)` [Addition-0]
| `(+` NE_{rand1}`)` [Addition-1]
| `(+` NE_{rand1} NE_{rand2}`)` [Addition-2]
| `(+` NE_{rand1} NE_{rand2} NE_{rand3}`)` [Addition-3]
| ...

Here we introduce a concise way of handling this kind of syntactic flexibility within s-expression grammars. We extend s-expression patterns so that any pattern can be annotated with a postfix '*' character. Such a pattern is called a **sequence pattern**. A sequence pattern P* matches any consecutive sequence of zero or more s-expressions SX_1 ... SX_n such that each SX_i matches the pattern P.

For instance, the extended addition expression can be specified concisely by the pattern `(+` NE_{rand}*`)`. Here are some phrases that match this new pattern, along with the sequence matched by NE_{rand}* in each case:

`(+ 1 2 3 4)`	$NE_{rand}{*} = [\texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}]_{\text{NumExp}}$
`(+ 2 (+ 3 4 5) (+ 6 7))`	$NE_{rand}{*} = [\texttt{2}, \texttt{(+ 3 4 5)}, \texttt{(+ 6 7)}]_{\text{NumExp}}$
`(+ 1)`	$NE_{rand}{*} = [\texttt{1}]_{\text{NumExp}}$
`(+)`	$NE_{rand}{*} = [\,]_{\text{NumExp}}$

Note that a sequence pattern can match any number of elements, including zero or one. To specify that an addition should have a minimum of two operands, we could use the following pattern:

`(+` NE_{rand1} NE_{rand2} NE_{rest}*`)`.

A postfix '+' is similar to '*,' except the pattern matches a sequence with at least one element. Thus, `(+` $NE_{rand}{}^{+}$`)` is equivalent to `(+` NE_{rand} NE_{rest}*`)`.

A postfix '*' or '+' can be attached to any s-expression pattern, not just a domain variable. For example, in the s-expression pattern

`(cond (`BE_{test} NE_{action}`)*` `(else` $NE_{default}$`))`,

the subpattern `(`BE_{test} NE_{action}`)*` matches any sequence of parenthesized clauses containing a boolean expression followed by a numerical expression.

To avoid ambiguity, s-expression grammars are not allowed to use s-expression patterns in which multiple sequence patterns enable a single s-expression to match a pattern in more than one way. As an example of a disallowed pattern, consider `(op` NE_{rand1}* NE_{rand2}*`)`, which could match the s-expression `(op 1 2)` in three different ways:

- $NE_{rand1}{*} = [\texttt{1}, \texttt{2}]_{\text{NumExp}}$ and $NE_{rand2}{*} = []_{\text{NumExp}}$

- $NE_{rand1}{}^* = [1]_{\text{NumExp}}$ and $NE_{rand2}{}^* = [2]_{\text{NumExp}}$
- $NE_{rand1}{}^* = []_{\text{NumExp}}$ and $NE_{rand2}{}^* = [1,\ 2]_{\text{NumExp}}$.

A disallowed pattern can always be transformed into a legal pattern by inserting explicit parentheses to demarcate components. For instance, the following are all unambiguous legal patterns:

`(op (`$NE_{rand1}{}^*$`) (`$NE_{rand2}{}^*$`))`

`(op (`$NE_{rand1}{}^*$`)` $NE_{rand2}{}^*$`)`

`(op` $NE_{rand1}{}^*$ `(`$NE_{rand2}{}^*$`)).`

2.3.5 Notational Conventions

In addition to the s-expression patterns described above, we will employ a few other notational conventions for syntax.

Domain Variables

In addition to being used in s-expression patterns, domain variables can appear inside s-expressions when they denote particular s-expression. For example, if NE_1 is the s-expression `(+ 1 2)` and NE_2 is the s-expression `(- 3 4)`, then `(*` NE_1 NE_2`)` is the same syntactic entity as `(* (+ 1 2) (- 3 4))`.

Sequence Notation

Sequence notation, including the infix notations for the *cons* ('.') and *append* ('@') sequence functions (see Section A.3.4), can be intermixed with s-expression notation to designate sequence elements of compound syntactic domains. For example, all of the following are alternative ways of writing the same extended EL addition expression:

`(+ 1 2 3)`

`(+ [1, 2, 3])`

`(+ [1, 2] @ [3])`

`(+ 1 . [2, 3])`

Similarly, if $NE_1 = 1$, $NE_2{}^* = [2,$ `(+ 3 4)`$]$, and $NE_3{}^* = [$`(* 5 6), (- 7 8)`$]$, then `(+` NE_1 `.` NE_2`*)` designates the same syntactic entity as

`(+ 1 2 (+ 3 4))`,

and `(+` $NE_2{}^*$ `@` NE_3`*)` designates the same syntactic entity as

`(+ 2 (+ 3 4) (* 5 6) (- 7 8)).`

The sequence notation is only legal in positions where a production for a compound syntactic domain contains a sequence pattern. For example, the following notations are illegal because `if` expressions do not contain any component sequences:

```
(if [(< (arg 1) 1), 2, 3])

(if [(< (arg 1) 1), 2] @ [3])

(if (< (arg 1) 1) . [2, 3]).
```

Sequence notation can be used in s-expression patterns as well. For example, the pattern

$$\texttt{(+ } NE_{rand1} \texttt{ . } NE_{rest}\texttt{*)}$$

matches any addition expression with at least one operrand. The pattern

$$\texttt{(+ } NE_{rands1}\texttt{* @ } NE_{rands2}\texttt{*)}$$

can match an addition expression with any number of operands. If the expression has one or more arguments, the match is ambiguous (and therefore disallowed, see page 29) since there are multiple ways to bind NE_{rands1}* and NE_{rands2}* to sequences that append to the argument sequence.

Syntactic Functions

We will follow a convention (standard in the semantics literature) that functions on compound syntactic domains are defined by a series of clauses, one for each production. Figure 2.7 illustrates this style of definition for two functions on EL expressions: *nheight* specifies the height of a numerical expression, while *bheight* specifies the height of a boolean expression. Each clause consists of two parts: a *head* that specifies an s-expression pattern from a production; and a *body* that describes the meaning of the function for s-expressions that match the head pattern. The double brackets, $[\![\,]\!]$, are traditionally used in syntactic functions to demarcate a syntactic argument, and thus to clearly separate expressions in the language being defined (program code, for example) from the language of the semantics. These brackets may be viewed as part of the name of the syntactic function.

Functions on syntactic domains are formally maps from s-expressions to a result domain. However, for all intents and purposes, they can also be viewed as maps from abstract syntax trees to the result domain. Each clause of a syntactic function definition specifies how the function at the node of an AST is defined in terms of the result of applying this function to the components of the AST.

$nheight$: NumExp $\rightarrow$ *Nat*
$nheight[\![NE]\!] = 0$
$nheight[\![\texttt{(arg } NE\texttt{)}]\!] = 0$
$nheight[\![\texttt{(}A\ NE_1\ NE_2\texttt{)}]\!] = (1 + (max\ nheight[\![NE_1]\!]\ nheight[\![NE_2]\!]))$
$nheight[\![\texttt{(if } BE_{test}\ NE_{con}\ NE_{alt}\texttt{)}]\!]$
$= (1 + (max\ bheight[\![BE_{test}]\!]\ (max\ nheight[\![NE_{con}]\!]\ nheight[\![NE_{alt}]\!])))$

$bheight$: BoolExp $\rightarrow$ *Nat*
$bheight[\![B]\!] = 0$
$bheight[\![\texttt{(}R\ NE_1\ NE_2\texttt{)}]\!] = (1 + (max\ nheight[\![NE_1]\!]\ nheight[\![NE_2]\!]))$
$bheight[\![\texttt{(}L\ BE_1\ BE_2\texttt{)}]\!] = (1 + (max\ bheight[\![BE_1]\!]\ bheight[\![BE_2]\!]))$

Figure 2.7: Two examples illustrating the form of function definitions on syntactic domains.

2.4 The Syntax of PostFix

Equipped with our syntactic tools, we are now ready to formally specify the syntactic structure of PostFix, the stack language introduced in Section 1.4, and to explore some variations on this structure. Figure 2.8 presents an s-expression grammar for PostFix. Top-level programs are represented as s-expressions of the form `(postfix` $N_{numargs}$ Q_{body}`)`, where $N_{numargs}$ is a numeral specifying the number of arguments and Q_{body} is the command sequence executed by the program. The sequence pattern C* in the production for Commands (Q) indicates that it is a sequence domain over elements from the Command domain. Most of the elements of Command (C) are single tokens (e.g., `add` and `sel`), except for executable sequences, which are parenthesized elements of the Commands domain. The mutually recursive structure of Command and Commands permits arbitrary nesting of executable sequences.

The concrete details specified by Figure 2.8 are only one way of capturing the underlying abstract syntactic structure of the language. Figure 2.9 presents an alternative s-expression grammar for PostFix. In order to avoid confusion, we will refer to the language defined in Figure 2.9 as PostFix2.

There are two main differences between the grammars of PostFix and PostFix2.

1. The PostFix2 grammar strictly adheres to the phrase tag convention introduced in Section 2.3.3. That is, every element of a compound syntactic domain appears as a parenthesized structure introduced by a unique tag. For example, `1` becomes `(int 1)`, `pop` becomes `(pop)`, and `add` becomes

$P \in$ Program
$Q \in$ Commands
$C \in$ Command
$A \in$ ArithmeticOperator = {`add`, `sub`, `mul`, `div`, `rem`}
$R \in$ RelationalOperator = {`lt`, `eq`, `gt`}
$N \in$ Intlit = {..., `-2`, `-1`, `0`, `1`, `2`, ...}

P ::= `(postfix` $N_{numargs}$ Q_{body}`)`	[Program]
Q ::= C*	[Command Sequence]
C ::= N	[IntLit]
| `pop`	[Pop]
| `swap`	[Swap]
| A	[Arithmetic Operator]
| R	[Relational Operator]
| `nget`	[NumGet]
| `sel`	[Select]
| `exec`	[Execute]
| `(`Q`)`	[Executable Sequence]

Figure 2.8: An s-expression grammar for PostFix.

`(arithop add)`.[4]

2. Rather than representing command sequences as a sequence domain, PostFix2 uses the `:` and `(skip)` commands to encode such sequences. `(skip)` is intended to be a "no op" command that leaves the stack unchanged, while `(:` C_1 C_2`)` is intended first to perform C_1 on the current stack and then to perform C_2 on the stack resulting from C_1. The `:` and `(skip)` commands in PostFix2 serve the roles of $cons_{\text{Command}}$ and $[]_{\text{Command}}$ in PostFix. For example, the PostFix command sequence

$$[\texttt{1}, \texttt{2}, \texttt{add}]_{\text{Command}} = (cons\ \texttt{1}\ \ (cons\ \texttt{2}\ \ (cons\ \texttt{add}\ []_{\text{Command}})))$$

can be encoded in PostFix2 as a single command:

```
(: (int 1) (: (int 2) (: (arithop add) (skip)))).
```

The difference in phrase tags is a surface variation in concrete syntax that does not affect the structure of abstract syntax trees. Whether sequences are explicit (the original grammar) or implicit (the alternative grammar) is a deeper

[4] the `arithop` keyword underscores that the arithmetic operators are related; similarly for `relop`.

$P \in$ Program
$C \in$ Command
$A \in$ ArithmeticOperator = {`add`, `sub`, `mul`, `div`, `rem`}
$R \in$ RelationalOperator = {`lt`, `eq`, `gt`}
$N \in$ Intlit = {..., `-2`, `-1`, `0`, `1`, `2`, ...}

P ::= `(postfix` $N_{numargs}$ C_{body}`)` [Program]

C ::= `(int` N`)`	[IntLit]
| `(pop)`	[Pop]
| `(swap)`	[Swap]
| `(arithop` A`)`	[Arithmetic Operator]
| `(relop` R`)`	[Relational Operator]
| `(nget)`	[NumGet]
| `(sel)`	[Select]
| `(exec)`	[Execute]
| `(seq` C`)`	[Executable Sequence]
| `(:` C_1 C_2`)`	[Compose]
| `(skip)`	[Skip]

Figure 2.9: An s-expression grammar for PostFix2, an alternative syntax for PostFix.

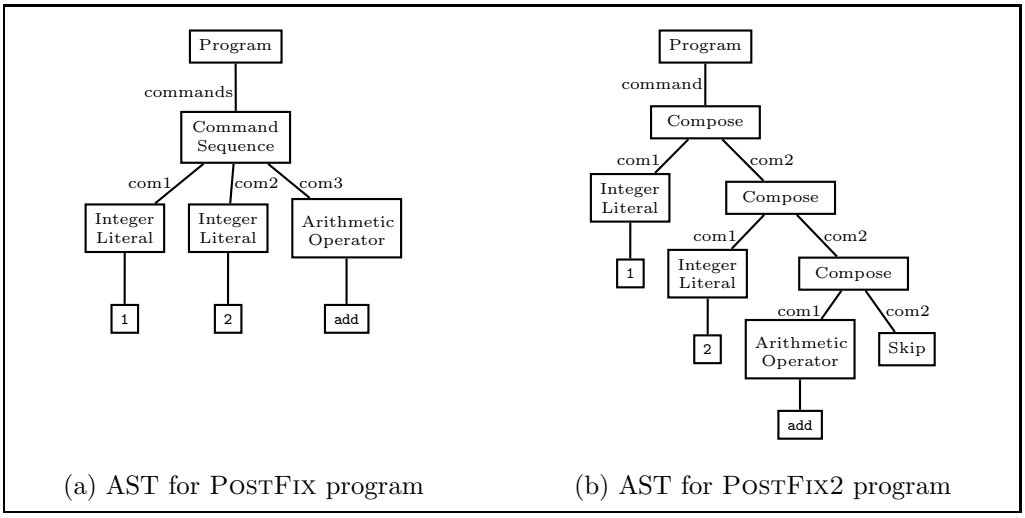
variation because the abstract syntax trees differ in these two cases (see Figure 2.10).

Although the tree structures are similar, it is not *a priori* possible to determine that the second tree encodes a sequence without knowing more about the semantics of compositions and skips. In particular, `:` and `(skip)` must satisfy two behavioral properties in order for them to encode sequences:

- `(skip)` must be an **identity** for `:`. I.e., `(:` C `(skip))` and `(: (skip)` C`)` must behave like C.
- `:` must be **associative**. I.e., `(:` C_1 `(:` C_2 C_3`))` must behave the same as `(: (:` C_1 C_2`)` C_3`)`.

These two properties amount to saying that (1) skips can be ignored and (2) in a tree of compositions, only the order of the leaves matters. With these properties, any tree of compositions is isomorphic to a sequence of the non-skip leaves. The informal semantics of `:` and `(skip)` given above satisfies these two properties.

Is one of the two grammars presented above "better" than the other? It depends on the context in which they are used. As the following example indicates, the PostFix grammar certainly leads to programs that are more concise than those generated by the PostFix2 grammar:



(a) AST for PostFix program (b) AST for PostFix2 program

Figure 2.10: A comparison of the abstract syntax trees for two encodings of a PostFix program.

```
; PostFix
(postfix 1 (1 2 add) (3 4 mul) sel exec)

; PostFix2
(postfix2 1
  (: (seq (: (int 1) (: (int 2) (: (arithop add)
                                   (skip)))))
     (: (seq (: (int 3) (: (int 4) (: (arithop mul)
                                      (skip)))))
        (: (sel)
           (: (exec) (skip))))))
```

Additionally, we shall see that the explicit sequences of PostFix make it more amenable to certain kinds of semantic analysis. On the other hand, other semantic and pragmatic tools are easier to apply to PostFix2 programs. Though we will focus on the PostFix grammar, we will consider PostFix2 when it is instructive to do so. In any event, the reader should be aware that even the fairly constrained boundaries of s-expression grammars leave some room for design decisions.

Reading

The notion of abstract syntax is due to McCarthy [McC62]. This notion is commonly used in operational and denotational semantics to ignore unimportant syntactic details (see the references in Chapters 3–4). Interpreters and compilers often have a "front-end" stage that converts concrete syntax into explicit data structures representing abstract syntax trees.

Our s-expression grammars are based on McCarthy's LISP s-expression notation [McC60], which is a trivially parsable generic and extensible concrete syntax for programming languages. Many tools — most notably Lex [Les75] and Yacc [Joh75] — are available for converting more complex concrete syntax into abstract syntax trees. A discussion of these tools, and the scanning and parsing theory behind them, can be found in almost any compiler textbook. For a particularly concise account, consult one of Appel's textbooks [App98b, App98a, AP02].

Chapter 3

Operational Semantics

And now I see with eye serene
The very pulse of the machine.

— She Was a Phantom of Delight, William Wordsworth

3.1 The Operational Semantics Game

Consider executing the following PostFix program on the arguments [4, 5]:

```
(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)
```

It helps to have a bookkeeping notation that represents the process of applying the informal rules presented in Chapter 1. For example, the table in Figure 3.1 illustrates one way to represent the execution of the above program. The table has two columns: the first column in each row holds the current command sequence; the second holds the current stack. The execution process begins by filling the first row of the table with the command sequence of the given program and an empty stack. Execution proceeds in a step-by-step fashion by using the rule for the first command of the current row to generate the next row. Each execution step removes the first command from the sequence and updates the stack. In the case of `exec`, new commands may also be prepended to the command sequence. The execution process terminates as soon as a row with an empty command sequence is generated. The result of the execution is the top stack element of the final row (`-3` in the example).

The table-based technique for executing PostFix programs exemplifies an **operational semantics**. Operational semantics formalizes the common intuition that program execution can be understood as a step-by-step process that

Commands	**Stack**
`(2 (3 mul add) exec) 1 swap exec sub`	`4` `5`

`1 swap exec sub`	`(2 (3 mul add) exec)`
	`4`
	`5`

`swap exec sub`	`1`
	`(2 (3 mul add) exec)`
	`4`
	`5`

`exec sub`	`(2 (3 mul add) exec)`
	`1`
	`4`
	`5`

`2 (3 mul add) exec sub`	`1`
	`4`
	`5`

`(3 mul add) exec sub`	`2`
	`1`
	`4`
	`5`

`exec sub`	`(3 mul add)`
	`2`
	`1`
	`4`
	`5`

`3 mul add sub`	`2`
	`1`
	`4`
	`5`

`mul add sub`	`3`
	`2`
	`1`
	`4`
	`5`

`add sub`	`6`
	`1`
	`4`
	`5`

`sub`	`7`
	`4`
	`5`

	`-3` `5`

Figure 3.1: A table showing the step-by-step execution of a PostFix program.

evolves by the mechanical application of a fixed set of rules. Sometimes the rules describe how the state of some physical machine is changed by executing an instruction. For example, assembly code instructions are defined in terms of the effect that they have on the architectural elements of a computer: registers, stack, memory, instruction stream, etc. But the rules may also describe how language constructs affect the state of some **abstract machine** that provides a mathematical model for program execution. Each state of the abstract machine is called a **configuration**.

For example, in the PostFix abstract machine implied by the table in Figure 3.1, each configuration is modeled by one row of the execution table: a pair of a program and a stack. The next configuration of the machine is determined from the current one based on the first command in the current program. The behavior of each command can be specified in terms of how it transforms the current configuration into the next one. For example, executing the `add` command removes it from the command sequence and replaces the top two elements of the stack by their sum. Executing the `exec` command pops an executable sequence from the top of the stack and prepends its commands in front of the commands following `exec`.

The general structure of an operational semantics execution is illustrated in Figure 3.2. An abstract machine accepts a program to be executed along with its inputs and then chugs away until it emits an answer. Internally, the abstract machine typically manipulates configurations with two kinds of parts:

1. The **code component**: a program phrase that controls the rest of the computation.

2. The **state components**: entities that are manipulated by the program during its execution. In the case of PostFix, the single state component is a stack, but configurations for other languages might include state components modeling random-access memory, a set of name/object bindings, a file system, a graphics state, various kinds of control information, etc. Sometimes there are no state components, in which case a configuration is just code.

The stages of the operational execution are as follows:

- The program and its inputs are first mapped by an **input function** into an **initial configuration** of the abstract machine. The code component of the initial configuration is usually some part of the given program, and the state components are appropriately initialized from the inputs. For instance, in an initial configuration for PostFix, the code component is the

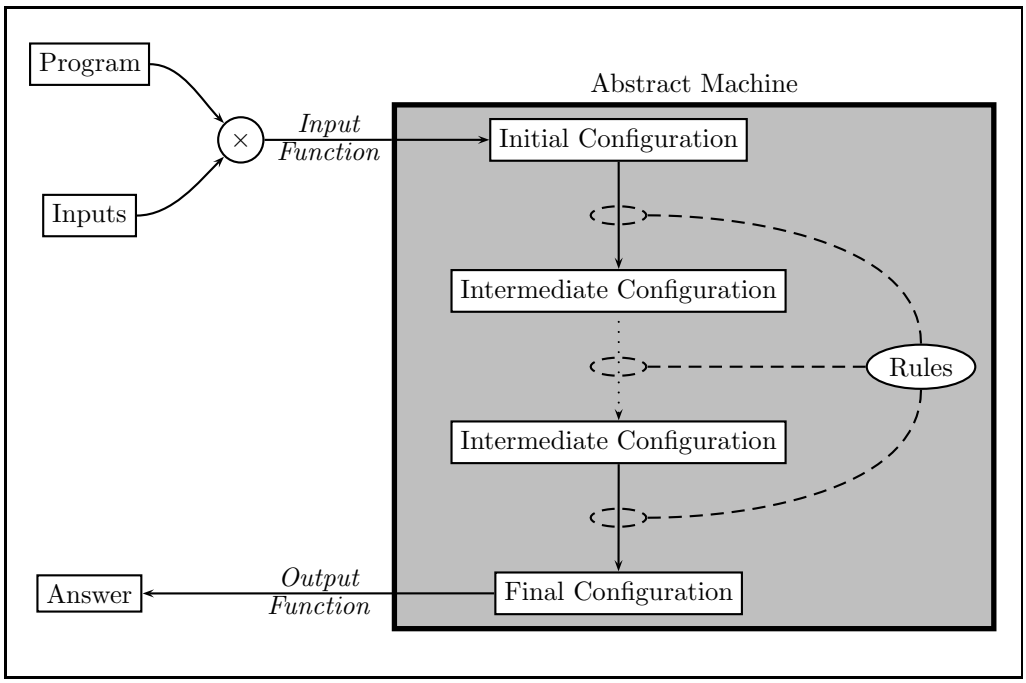


Figure 3.2: The operational semantics "game board."

command sequence body of the program and the single state component is a stack containing the integer arguments in order with the first argument at the top of the stack.

- After an initial configuration has been constructed, it's time to "turn the crank" of the abstract machine. During this phase, the rules governing the abstract machine are applied in an iterative fashion to yield a sequence of intermediate configurations. Each configuration is the result of one step in the step-by-step execution of the program. This stage continues until a configuration is reached that is deemed to be a **final configuration**. What counts as a final configuration varies widely between abstract machines. In the case of PostFix, a configuration is final when the code component is an empty command sequence.

- The last step of execution is mapping the final configuration to an **answer** via an **output function**. What is considered to be an answer differs greatly from language to language. For PostFix, the answer is the top stack value in a final configuration, if it's an integer. If the stack is empty or the top value is an executable sequence, the answer is an error token. In

other systems, the answer might also include elements like the final state of the memory, file system, or graphics screen.

Sometimes an abstract machine never reaches a final configuration. This can happen for one of two reasons:

1. The abstract machine may reach a non-final configuration to which no rules apply. Such a configuration is said to be a **stuck state**. For example, the initial configuration for the PostFix program `(postfix 1 sub)` is a stuck state because the rules in Figure 1.1 don't say how to handle `sub` when the stack doesn't contain at least two elements. (The configuration is not final because the command sequence [`sub`] is non-empty.) Stuck states often model error situations.

2. The rule-applying process of the abstract machine might not terminate. In any universal programming language[1] it is possible to write programs that loop forever. For such programs, the execution process of the abstract machine never terminates. As a consequence of the halting theorem[2], we can't do better than this: there's no general way to tweak the abstract machine of a universal language so that it always indicates when it is in an infinite loop.

We show in Section 3.4.3 that all PostFix programs must terminate. This implies that PostFix is not universal.

3.2 Small-step Operational Semantics (SOS)

3.2.1 Formal Framework

Above, we presented a high-level introduction to operational semantics. Here, we iron out all the details necessary to turn this approach into a formal framework known as **small-step operational semantics** (SOS[3]). An SOS is characterized by the use of **rewrite rules** to specify the step-by-step transformation of configurations in an abstract machine.

To express this framework formally, we will use the mathematical metalanguage described in Appendix A. Before reading further, you should at least

[1] A programming language is **universal** if it can express all computable functions.

[2] The **halting theorem** states that there is no program that can decide for all programs P and all inputs A whether P terminates on A.

[3] This framework, due to Plotkin [Plo81], was originally called **structured operational semantics**. It later became known as the small-step approach to distinguish it from – you guessed it – a big-step approach (see Section 3.3).

skim this appendix to familiarize yourself with the notational conventions of the metalanguage. Later, when you encounter an unfamiliar notation or concept, consult the relevant section of the appendix for a detailed explanation.

Consider a programming language L with legal programs $P \in$ Program, inputs $I \in$ Inputs, and elements $A \in$ Answer that are considered to be valid answers to programs. Then an SOS for L is a five-tuple $SOS = \langle CF, \Rightarrow, FC, IF, OF \rangle$, where:

- CF is the set of **configurations** for an abstract machine for L. The metavariable cf ranges over configurations.

- $\Rightarrow$, the **transition relation**, is a binary relation between configurations that defines the allowable transitions between configurations. The notation $cf \Rightarrow cf'$ means that there is a **(one step) transition** from the configuration cf to the configuration cf'. This notation, which is shorthand for $\langle cf, cf' \rangle \in \Rightarrow$, is pronounced "$cf$ rewrites to cf' in one step." The two parts of a transition have names: cf is called the **left hand side (LHS)** and cf' is called the **right hand side (RHS)**. The transition relation is usually specified by rewrite rules, as described below in Section 3.2.3.

  The reflexive, transitive closure of $\Rightarrow$ is written $\stackrel{*}{\Rightarrow}$. So $cf \stackrel{*}{\Rightarrow} cf'$ means that cf rewrites to cf' in zero or more steps. The sequence of transitions between cf and cf' is called a **transition path**. The **length** of a transition path is the number of transitions in the path. The notation $cf \stackrel{n}{\Rightarrow} cf'$ indicates that cf rewrites to cf' in n steps, i.e., via a transition path of length n. The notation $cf \stackrel{\infty}{\Rightarrow}$ indicates that there is an infinitely long transition path beginning with cf.

  A configuration cf is **reducible** if there is some cf' such that $cf \Rightarrow cf'$. If there is no such cf', then we write $cf \not\Rightarrow$ and say that cf is **irreducible**. CF can be partitioned into two sets, $Reducible_{SOS}$ (containing all reducible configurations) and $Irreducible_{SOS}$ (containing all irreducible ones). We omit the SOS subscript when it is clear from context. A transition relation $\Rightarrow$ is **deterministic** if for every $cf \in Reducible_{SOS}$ there is exactly one cf' such that $cf \Rightarrow cf'$. Otherwise, $\Rightarrow$ is said to be **non-deterministic**.

- FC, the set of **final configurations**, is a subset of $Irreducible_{SOS}$ containing all configurations that are considered to be final states in the execution of a program. The set $Stuck_{SOS}$ of stuck states is defined to be $(Irreducible_{SOS} - FC)$ — i.e., the non-final irreducible configurations. We omit the SOS subscript when it is clear from context.

- IF : Program $\times$ Inputs $\rightarrow$ CF is an **input function** that maps a program and its inputs into an initial configuration.

- OF : $FC \rightarrow$ Answer is an **output function** that maps a final configuration to an appropriate answer domain.

An SOS defines the behavior of a program in a way that we shall now make precise. What are the possible behaviors of a program? As discussed above, a program either (1) returns an answer (2) gets stuck in a non-final irreducible configuration or (3) loops infinitely. We model these via the following Outcome domain, where `stuckout` designates a stuck program and `loopout` designates a infinitely looping program:

$$
\begin{array}{l}
\quad \text{StuckOut} = \{\texttt{stuckout}\} \\
\quad \text{LoopOut} = \{\texttt{loopout}\} \\
o \in \text{Outcome} = \text{Answer} + \text{StuckOut} + \text{LoopOut} \\
\quad \texttt{stuck} = (\text{StuckOut} \mapsto \text{Outcome } \texttt{stuckout}) \\
\quad \infty = (\text{StuckOut} \mapsto \text{Outcome } \texttt{loopout})
\end{array}
$$

Suppose that an SOS has a deterministic transition relation. Then we can define the behavior of a program P on inputs I as follows:

$$
\begin{array}{l}
beh_{det} : \text{Program} \times \text{Inputs} \rightarrow \text{Outcome} \\
beh_{det}\ \langle P, I \rangle = \begin{cases} (\text{Answer} \mapsto \text{Outcome}\ \ (OF\ \ cf)) & \text{if } (IF\ \ \langle P, I \rangle) \stackrel{*}{\Rightarrow} cf \in FC \\ \texttt{stuck} & \text{if } (IF\ \ \langle P, I \rangle) \stackrel{*}{\Rightarrow} cf \in Stuck \\ \infty & \text{if } (IF\ \ \langle P, I \rangle) \stackrel{\infty}{\Rightarrow} \end{cases}
\end{array}
$$

In the first case, an execution starting at the initial configuration eventually reaches a final configuration, whose answer is returned. In the second case, an execution starting at the initial configuration eventually gets stuck at a non-final configuration. In the last case, there is an infinite transition path starting at the initial configuration, so the program never halts.

What if the transition relation is not deterministic? In this case, it is possible that there are multiple transition paths starting at the initial configuration. Some of these might end at final configurations with different answers. Others might be infinitely long or end at stuck states. In general, we must allow for the possibility that there are many outcomes, so the signature of the behavior function *beh* in this case must return a *set* of outcomes — i.e., an element of the

powerset domain $\mathcal{P}(\text{Outcome})$[4]

$$beh : \text{Program} \times \text{Inputs} \to \mathcal{P}(\text{Outcome})$$

$$o \in (beh\ \langle P, I\rangle) \text{ if } \begin{cases} o = (\text{Answer} \mapsto \text{Outcome}\ \ (OF\ \ cf)) \\ \quad \text{and } (IF\ \ \langle P, I\rangle) \stackrel{*}{\Rightarrow} cf \in FC \\ o = \texttt{stuck} \text{ and } (IF\ \ \langle P, I\rangle) \stackrel{*}{\Rightarrow} cf \in \mathit{Stuck} \\ o = \infty \text{ and } (IF\ \ \langle P, I\rangle) \stackrel{\infty}{\Rightarrow} \end{cases}$$

An SOS with a non-deterministic transition relation won't necessarily give rise to results that contain multiple outcomes. Indeed, we will see later (in Section 3.4.2) that some systems with non-deterministic transition relations can still have a behavior that is deterministic — i.e., the resulting set of outcomes is always a singleton.

3.2.2 Example: An SOS for PostFix

We can now formalize the elements of the PostFix SOS described informally in Section 3.1 (except for the transition relation, which will be formalized in Section 3.2.3). The details are presented in Figure 3.3. A stack is a sequence of values that are either integer numerals (from domain Intlit) or executable sequences (from domain Commands). PostFix programs take a sequence of integer numerals as their inputs, and, when no error is encountered, return an integer numeral as an answer. A configuration is a pair of a command sequence and a stack. A final configuration is one whose command sequence is empty and whose stack is non-empty with an integer numeral on top (i.e., an element of FinalStack). The input function *IF* maps a program and its numeric inputs to a configuration consisting of the body command sequence and an initial stack with the inputs arranged from top down. If the number of arguments N expected by the program does not match the actual number n of arguments supplied, then *IF* returns a stuck configuration $\langle []_{\text{Command}}, []_{\text{Value}}\rangle$ that represents an error. The output function *OF* returns the top integer numeral from stack of a final configuration.

The PostFix SOS in Figure 3.3 models errors using stuck states. By definition, stuck states are exactly those irreducible configurations that are non-final. In PostFix, stuck states are irreducible configurations whose command sequence is non-empty or those that pair an empty command sequence with a stack that is empty or has an executable sequence on top. The outcome of a program that reaches such a configuration will be `stuck`.

[4]The result of *beh* must in fact be a *non-empty* set of outcomes, since every program will have at least one outcome.

Domains

$V \in \text{Value} = \text{Intlit} + \text{Commands}$
$S \in \text{Stack} = \text{Value}^*$
$\text{FinalStack} = \{S \mid (\mathit{length}\ S) \geq 1 \text{ and } (\mathit{nth}\ 1\ S) = (\text{Intlit} \mapsto \text{Value}\ N)\}$
$\text{Inputs} = \text{Intlit}^*$
$\text{Answer} = \text{Intlit}$

SOS

Suppose that the PostFix SOS has the form $PFSOS = \langle CF, \Rightarrow, FC, IF, OF\rangle$. Then the SOS components are:

$CF = \text{Commands} \times \text{Stack}$

$\Rightarrow$ is a deterministic transition relation defined in Section 3.2.3

$FC = \{[\,]_{\text{Command}}\} \times \text{FinalStack}$

$IF : \text{Program} \times \text{Inputs} \to CF$
$= \lambda\langle \texttt{(postfix}\ N\ Q\texttt{)}, [N_1, \ldots, N_n]\rangle\ .$
$\quad$ **if** $N = \text{n}$ **then** $\langle Q, [(\text{Intlit} \mapsto \text{Value}\ N_1), \ldots, (\text{Intlit} \mapsto \text{Value}\ N_n)]\rangle$
$\quad$ **else** $\langle [\,]_{\text{Command}}, [\,]_{\text{Value}}\rangle$ **fi**

$OF : FC \to \text{Answer} = \lambda\langle [\,]_{\text{Command}}, (\text{Intlit} \mapsto \text{Value}\ N)\ .\ S'\rangle\ .\ N$

Figure 3.3: An SOS for PostFix.

Although it is convenient to use stuck states to model errors, it is not strictly necessary. With some extra work, it is always possible to modify the final configuration set FC and the output function OF so that such programs instead have as their meaning some error token in Answer. Using PostFix as an example, we can use a modified answer domain Answer′ that includes an error token, a modified final configuration set FC' that includes all irreducible configurations, and the modified OF' shown below:

$\text{Error} = \{\texttt{error}\}$
$\text{Answer}' = \text{Intlit} + \text{Error}$

$FC' = \mathit{Irreducible}_{PFSOS}$

$OF : FC' \to \text{Answer}'$
$= \lambda\langle Q, V^*\rangle\ .$ **matching** $\langle Q, V^*\rangle$
$\quad \triangleright \langle [\,]_{\text{Command}}, (\text{Intlit} \mapsto \text{Value}\ N)\ .\ S'\rangle \ [\!] \ (\text{Intlit} \mapsto \text{Answer}'\ N)$
$\quad \triangleright$ **else** $(\text{Error} \mapsto \text{Answer}'\ \texttt{error})$

With these modifications, the behavior of a PostFix program that encounters an error will be $(\text{Answer}' \mapsto \text{Outcome}\ (\text{Error} \mapsto \text{Answer}'\ \texttt{error}))$ rather than `stuck`.

▷ **Exercise 3.1** Look up definitions of the following kinds of automata and express each of them in the SOS framework: deterministic finite automata, non-deterministic finite automata, deterministic pushdown automata, and Turing machines. Represent strings, stacks, and tapes as sequences of symbols. ◁

3.2.3 Rewrite Rules

The transition relation, $\Rightarrow$, for an SOS is often specified by a set of **rewrite rules**. A rewrite rule has the form

$$\frac{\textit{antecedents}}{\textit{consequent}} \qquad [\textit{rule-name}]$$

where the antecedents and the consequent contain transition patterns (described below). Informally, the rule asserts: "If the transitions specified by the *antecedents* are valid, then the transition specified by the consequent is valid." The label [*rule-name*] on the rule is just a handy name for referring to the rule, and is not a part of the rule structure. A rewrite rule with no antecedents is an **axiom**; otherwise it is a **progress rule**. The horizontal bar is usually omitted when writing an axiom.

A complete set of rewrite rules for PostFix appears in Figure 3.4. All of the rules are axioms. Together with the definitions of CF, FC, IF, and OF, these rules constitute a formal SOS version of the informal PostFix semantics originally presented in Figure 1.1. We will spend the rest of this section studying the meaning of these rules and considering alternative rules.

3.2.3.1 Axioms

Since an axiom has no antecedents, it is determined solely by its consequent. As noted above, the consequent must be a **transition pattern**. A transition pattern looks like a transition except that the LHS and RHS may contain domain variables interspersed with the usual notation for configurations. Informally, a transition pattern is a schema that stands for all the transitions that match the pattern. An axiom stands for the collection of all configuration pairs that match the LHS and RHS of the transition pattern, respectively.

As an example, let's consider in detail the axiom that defines the behavior of PostFix numerals:

$$\langle N \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ N \ . \ S\rangle \qquad [\textit{num}]$$

This axiom stands for an infinite number of pairs of configurations of the form $\langle cf, cf'\rangle$. It says that if cf is a configuration in which the command sequence is

$$\langle N \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ N \ . \ S\rangle \qquad [num]$$

$$\langle (\,Q_{exec}\,) \ . \ Q_{rest}, \ S\rangle \Rightarrow \langle Q_{rest}, \ (\,Q_{exec}\,) \ . \ S\rangle \qquad [seq]$$

$$\langle \texttt{pop} \ . \ Q, \ V_{top} \ . \ S\rangle \Rightarrow \langle Q, \ S\rangle \qquad [pop]$$

$$\langle \texttt{swap} \ . \ Q, \ V_1 \ . \ V_2 \ . \ S\rangle \Rightarrow \langle Q, \ V_2 \ . \ V_1 \ . \ S\rangle \qquad [swap]$$

$$\langle \texttt{sel} \ . \ Q_{rest}, \ V_{false} \ . \ V_{true} \ . \ \texttt{0} \ . \ S\rangle \Rightarrow \langle Q_{rest}, \ V_{false} \ . \ S\rangle \qquad [sel\text{-}false]$$

$$\langle \texttt{sel} \ . \ Q_{rest}, \ V_{false} \ . \ V_{true} \ . \ N_{test} \ . \ S\rangle \Rightarrow \langle Q_{rest}, \ V_{true} \ . \ S\rangle, \quad \text{where } N_{test} \neq \texttt{0} \qquad [sel\text{-}true]$$

$$\langle \texttt{exec} \ . \ Q_{rest}, \ (\,Q_{exec}\,) \ . \ S\rangle \Rightarrow \langle Q_{exec} \ @ \ Q_{rest}, \ S\rangle \qquad [execute]$$

$$\langle A \ . \ Q, \ N_1 \ . \ N_2 \ . \ S\rangle \Rightarrow \langle Q, \ N_{result} \ . \ S\rangle, \quad \text{where } N_{result} = (calculate \ A \ N_2 \ N_1) \qquad [arithop]$$

$$\langle R \ . \ Q, \ N_1 \ . \ N_2 \ . \ S\rangle \Rightarrow \langle Q, \ \texttt{1} \ . \ S\rangle, \quad \text{where } (compare \ R \ N_2 \ N_1) \qquad [relop\text{-}true]$$

$$\langle R \ . \ Q, \ N_1 \ . \ N_2 \ . \ S\rangle \Rightarrow \langle Q, \ \texttt{0} \ . \ S\rangle, \quad \text{where } \neg\,(compare \ R \ N_2 \ N_1) \qquad [relop\text{-}false]$$

$$\langle \texttt{nget} \ . \ Q, \ N_{index} \ . \ [V_1, \ldots, V_{N_{size}}]\rangle \Rightarrow \langle Q, \ V_{N_{index}} \ . \ [V_1, \ldots, V_{N_{size}}]\rangle, \quad \text{where } (compare \ \texttt{gt} \ N_{index} \ \texttt{0}) \wedge \neg\,(compare \ \texttt{gt} \ N_{index} \ N_{size}) \qquad [nget]$$

Figure 3.4: Rewrite rules defining the transition relation ($\Rightarrow$) for PostFix.

a numeral N followed by Q and the stack is S, then there is a transition from cf to a configuration cf' whose command sequence is Q, and whose stack holds N followed by S.

In the [*num*] rule, N, Q, and S are domain variables that act as patterns that can match any element in the domain over which the variable ranges. Thus, N matches any integer numeral, Q matches any command sequence, and S matches any stack. When the same pattern variable occurs more than once within a rule, all occurrences must denote the same element; this constrains the class of transitions specified by the rule. Thus, the [*num*] rule matches the transition

$$\langle \texttt{(17 add swap)},\ [\texttt{19},\ \texttt{(2 mul)}]\rangle \Rightarrow \langle \texttt{(add swap)},\ [\texttt{17},\ \texttt{19},\ \texttt{(2 mul)}]\rangle$$

with $N = \texttt{17}$, $Q = [\texttt{add},\ \texttt{swap}]$, and $S = [\texttt{19},\ [\texttt{2},\ \texttt{mul}]]$. On the other hand, the rule does not match the transition

$$\langle \texttt{(17 add swap)},\ [\texttt{19},\ \texttt{(2 mul)}]\rangle \Rightarrow \langle \texttt{(add swap)},\ [\texttt{17},\ \texttt{19},\ \texttt{(2 mul)},\ \texttt{23}]\rangle$$

because there is no consistent interpretation for the pattern variable S — it is $[\texttt{19},\ [\texttt{(2 mul)}]]$ in the LHS of the transition, and $[\texttt{19},\ \texttt{(2 mul)},\ \texttt{23}]$ in the RHS.

As another example, the configuration pattern $\langle Q,\ N\ .\ N\ .\ S\rangle$ would only match configurations with stacks in which the top two values are the same integer numeral. If the RHS of the [*num*] rule consequent were replaced with this configuration pattern, then the rule would indicate that two copies of the integer numeral should be pushed onto the stack.

At this point, the meticulous reader may have noticed that in the rewrite rules and sample transitions we have taken many liberties with our notation. If we had strictly adhered to our metalanguage notation, then we would have written the [*num*] rule as

$$\langle (\text{Intlit} \mapsto \text{Command}\ N)\ .\ Q, S\rangle \Rightarrow \langle Q, (\text{Intlit} \mapsto \text{Value}\ N)\ .\ S\rangle \qquad [num]$$

and we would have written the matching transition as

$$\begin{aligned}
&\langle [\texttt{17}, \texttt{add}, \texttt{swap}]_{\text{Command}},\ [(\text{Intlit} \mapsto \text{Value}\ \texttt{19}),\\
&\qquad\qquad (\text{Commands} \mapsto \text{Value}\ [\texttt{2}, \texttt{mul}]_{\text{Command}})\ ]\rangle\\
&\Rightarrow \langle [\texttt{add}, \texttt{swap}]_{\text{Command}},\ [(\text{Intlit} \mapsto \text{Value}\ \texttt{17}),\\
&\qquad\qquad (\text{Intlit} \mapsto \text{Value}\ \texttt{19}),\\
&\qquad\qquad (\text{Commands} \mapsto \text{Value}\ [\texttt{2}, \texttt{mul}]_{\text{Command}})\ ]\rangle.
\end{aligned}$$

However, we believe that the more rigorous notation severely impedes the readability of the rules and examples. For this reason, we will stick with our stylized notation when it is unlikely to cause confusion. In particular, in operational semantics rules and sample transitions, we adopt the following conventions:

- Injections will be elided when they are clear from context. For example, if N appears as a command, then it stands for (Intlit $\mapsto$ Command N), while if it appears as a stack element, then it stands for (Intlit $\mapsto$ Value N).

- Sequences of syntactic elements will often be written as parenthesized s-expressions. For example, the PostFix command sequence

$$[\texttt{3},\ [\texttt{2},\ \texttt{mul}]_{\text{Command}},\ \texttt{swap}]_{\text{Command}}$$

  will be abbreviated as

```
(3 (2 mul) swap).
```

  The former is more precise, but the latter is easier to read. In PostFix examples, we have chosen to keep the sequence notation for stacks to visually distinguish the two components of a configuration.

Despite these notational acrobatics, keep in mind that we are manipulating well-defined mathematical structures. So it is always possible to add the appropriate decorations to make the notation completely rigorous.[5]

Some of the PostFix rules ([*arithop*], [*relop-true*], [*relop-false*], [*sel-true*], and [*nget*]) include **side conditions** that specify additional restrictions on the domain variables. For example, consider the axiom which handles a conditional whose test is true:

$$\langle \texttt{sel}\ .\ Q_{rest},\ \ V_{false}\ .\ V_{true}\ .\ N_{test}\ .\ S\rangle \Rightarrow \langle Q_{rest},\ \ V_{true}\ .\ S\rangle, \quad \text{where } N_{test} \neq \texttt{0} \qquad [\textit{sel-true}]$$

This axiom encodes the fact that `sel` treats any nonzero integer numeral as true. It says that as long as the test numeral N_{test} (the third element on the stack) is not the same syntactic object as `0`, then the next configuration is obtained by removing `sel` from the command sequence, and pushing the second stack element on the result of popping the top three elements off of the stack. The domain variable N_{test} that appears in the side condition $N_{test} \neq \texttt{0}$ stands for the same entity that N_{test} denotes in the LHS of the consequent, providing the link between the transition pattern and the side condition. Note how the domain variables and the structure of the components are used to constrain the pairs of configurations that satisfy this rule. This rule only represents pairs $\langle cf, cf'\rangle$ in which the stack of cf contains at least three elements, the third of which is a nonzero integer numeral. The rule does not apply to configurations whose stacks have fewer than three elements, or whose third element is an executable sequence or the numeral `0`.

[5] But those who pay too much attention to rigor may develop rigor mortis!

The side conditions in the [*arithop*], [*relop*], and [*nget*] rules deserve some explanation. The *calculate* function used in the side condition of [*arithop*] returns the numeral N_{result} resulting from the application of the operator A to the operands N_2 and N_1; it abstracts away the details of such computations.[6] We assume that *calculate* is a partial function that is undefined when A is `div` and N_1 is `0`, so division by zero yields a stuck state. The [*relop-true*] and [*relop-false*] rules are similar to [*arithop*]; here the auxiliary *compare* function is assumed to return the truth value resulting from the associated comparison. The rules then convert this truth value into a PostFix value of `1` (true) or `0` (false). In the [*nget*] rule, the *compare* function is used to ensure that the numeral N_{index} is a valid index for one of the values on the stack. If not, the configuration is stuck. In the side conditions, the symbol $\neg$ stands for logical negation and $\wedge$ stands for logical conjunction.

You should now know enough about the rule notation to understand all of the rewrite rules in Figure 3.4. The [*num*] and [*seq*] rules push the two different kinds of values onto the stack. The [*swap*], [*pop*], [*sel-true*], and [*sel-false*] rules all perform straightforward stack manipulations. The [*exec*] rule prepends an executable sequence from the stack onto the command sequence following the current command.

It is easy to see that the transition relation defined in Figure 3.4 is deterministic. The first command in the command sequence of a configuration uniquely determines which transition pattern might match, except for the case of `sel`, where the third stack value distinguishes whether [*sel-true*] or [*sel-false*] matches. The LHS of each transition pattern can match a given configuration in at most one way. So for any given PostFix configuration cf, there is at most one cf' such that $cf \Rightarrow cf'$.

3.2.3.2 Operational Execution

The operational semantics can be used to execute a PostFix program in a way similar to the table-based method presented earlier. For example, the execution of the PostFix program shown earlier in Figure 3.1 is illustrated in Figure 3.5. The input function is applied to the program to yield an initial configuration, and then a series of transitions specified by the rewrite rules are applied. In the

[6] Note that *calculate* manipulates *numerals* (i.e., names for integers) rather than the *integers* that they name. This may seem pedantic, but we haven't described yet how the meaning of an integer numeral is determined. In fact, integers are never even used in the SOS for PostFix. If we had instead defined the syntax of PostFix to use integers rather than integer numerals, then we could have used the usual integer addition operation here. But we chose integer numerals to emphasize the syntactic nature of operational semantics.

figure, the configuration resulting from each transition appears on a separate line and is labeled by the applied rule. When a final configuration is reached, the output function is applied to this configuration to yield `-3`, which is the result computed by the program. We can summarize the transition path from the initial to the final configuration as

$$\langle \texttt{((2 (3 mul add) exec) 1 swap exec sub)}, [\texttt{4}, \texttt{5}]\rangle \stackrel{10}{\Longrightarrow} \langle \texttt{()}, [\texttt{-3}, \texttt{5}]\rangle,$$

where 10 is the number of transitions. If we don't care about this number, we write $*$ in its place.

$(IF\ \langle$`(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)`$, [4,5]\rangle)$
$= \langle$`((2 (3 mul add) exec) 1 swap exec sub), [4, 5]`$\rangle$
$\Rightarrow \langle$`(1 swap exec sub), [(2 (3 mul add) exec), 4, 5]`$\rangle$ [*seq*]
$\Rightarrow \langle$`(swap exec sub), [1, (2 (3 mul add) exec), 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(exec sub), [(2 (3 mul add) exec), 1, 4, 5]`$\rangle$ [*swap*]
$\Rightarrow \langle$`(2 (3 mul add) exec sub), [1, 4, 5]`$\rangle$ [*execute*]
$\Rightarrow \langle$`((3 mul add) exec sub), [2, 1, 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(exec sub), [(3 mul add), 2, 1, 4, 5]`$\rangle$ [*seq*]
$\Rightarrow \langle$`(3 mul add sub), [2, 1, 4, 5]`$\rangle$ [*execute*]
$\Rightarrow \langle$`(mul add sub), [3, 2, 1, 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(add sub), [6, 1, 4, 5]`$\rangle$ [*arithop*]
$\Rightarrow \langle$`(sub), [7, 4, 5]`$\rangle$ [*arithop*]
$\Rightarrow \langle$`()`$, [\texttt{-3}, \texttt{5}]\rangle \in FC$ [*arithop*]
$(OF\ \langle$`()`$, [\texttt{-3}, \texttt{5}]\rangle) =$ `-3`

Figure 3.5: An SOS-based execution of a PostFix program.

Not all PostFix executions lead to a final configuration. For example, executing the program `(program 2 add mul 3 4 sub)` on the inputs $[5, 6]$ leads to the configuration $\langle$`(mul 3 4 sub), [11]`$\rangle$. This configuration is not final because there are still commands to be executed. But it does not match the LHS of any rewrite rule consequent. In particular, the [*arithop*] rule requires the stack to have two integers at the top, and here there is only one. This is an example of a stuck state. As discussed earlier, a program reaching a stuck state is considered to signal an error. In this case the error is due to an insufficient number of arguments on the stack.

▷ **Exercise 3.2** Use the SOS for PostFix to determine the values of the PostFix programs in Exercise 1.1. ◁

▷ **Exercise 3.3** Consider extending PostFix with a `rot` command defined by the following rewrite rule:

$$\langle \texttt{rot} \ . \ Q, \ N \ . \ V_0 \ . \ V_1 \ . \ \ldots \ . \ V_N \ . \ S\rangle \Rightarrow \langle Q, \ V_1 \ . \ \ldots \ . \ V_N \ . \ V_0 \ . \ S\rangle, \quad \text{where } (\textit{compare } \texttt{gt } N \ \texttt{0}) \qquad [\textit{rot}]$$

a. Give an informal English description of the behavior of `rot`.

b. What is the contents of the stack after executing the following program on zero arguments?

```
(postfix 0 1 2 3 1 2 3 rot rot rot)
```

c. Using `rot`, write a PostFix executable sequence that serves as subroutine for reversing the top three elements of a given stack.

d. List the kinds of situations in which `rot` can lead to a stuck state, and give a sample program illustrating each one. ◁

▷ **Exercise 3.4** The SOS for PostFix specifies that a configuration is stuck when the stack contains an insufficient number of values for a command. For example, $\langle$`(2 mul)`, $[]\rangle$ is stuck because multiplication requires two stack values.

a. Modify the semantics of PostFix so that, rather than becoming stuck, it uses sensible defaults for the missing values when the stack contains an insufficient number of values. For example, the default value(s) for `mul` would be `1`:

$\langle$`(2 mul)`, $[]\rangle \Rightarrow \langle$`(2)`, $[]\rangle$
$\langle$`(mul)`, $[]\rangle \Rightarrow \langle$`(1)`, $[]\rangle$

b. Do you think this modification is a good idea? Why or why not? ◁

▷ **Exercise 3.5** Suppose the Value domain in the PostFix SOS is augmented with a distinguished error value. Modify the rewrite rules for PostFix so that error configurations push this error value onto the stack. The error value should be "contagious" in the sense that any operation attempting to act on it should also push an error value onto the stack. Under the revised semantics, a program may return a non-error value even though it encounters an error along the way. E.g., `(postfix 0 1 2 add mul 3 4 sub)` should return `-1` rather than signaling an error when called on zero inputs. ◁

▷ **Exercise 3.6** An operational semantics for PostFix2 (the alternative PostFix syntax introduced in Figure 2.9) can be defined by making minor tweaks to the operational semantics for PostFix. Assume that the set of configurations remains unchanged. Then most commands from the secondary syntax can be handled with only cosmetic changes. For example, here is the rewrite rule for a PostFix2 numeral command:

$$\langle \texttt{(int } N\texttt{)} \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ N \ . \ S\rangle \qquad [\textit{numeral}']$$

a. Define an input function that maps PostFix2 programs (which have the form `(postfix2` N C`)`) into an initial configuration.

b. Give rewrite axioms for the PostFix2 commands `(exec)`, `(skip)`, and `(:` C_1 C_2`)`.

(See Exercise 3.7 for another approach to defining the semantics of PostFix2.) ◁

▷ **Exercise 3.7** A distinguishing feature of PostFix2 (the alternative PostFix syntax introduced in Figure 2.9) is that its grammar makes no use of sequence domains. It is reasonable to expect that its operational semantics can be modeled by configurations in which the code component is a single command rather than a command sequence. Based on this idea, design an SOS for PostFix2 in which $CF = \text{Command} \times \text{Stack}$. (Note: do not modify the Command domain.) ◁

▷ **Exercise 3.8** The Hugely Profitable Calculator Company has hired you to design a calculator language called RPN that is based on PostFix. RPN has the same syntax as PostFix command sequences (an RPN program is just a command sequence that is assumed to take zero arguments) and the operations are intended to work in basically the same manner. However, instead of providing a arbitrarily large stack, RPN limits the size of the stack to four values. Additionally, the stack is always **full** in the sense that it contains four values at all times. Initially, the stack contains four `0` values. Pushing a value onto a full stack causes the bottommost stack value to be forgotten. Popping the topmost value from a full stack has the effect of duplicating the bottommost element (i.e., it appears in the last two stack positions after the pop).

a. Develop a complete SOS for the RPN language.

b. Use your SOS to find the results of the following RPN programs:

   i. `(mul 1 add)`

   ii. `(1 20 300 4000 50000 add add add add)`

c. Although PostFix programs are guaranteed to terminate, RPN programs are not. Demonstrate this fact by writing an RPN program that loops infinitely. ◁

▷ **Exercise 3.9** A class of calculators known as *four-function* calculators supports the four usual binary arithmetic operators (`+`, `-`, `*`, `/`) in an infix notation.[7] Here we consider a language FF based on four-function calculators. The programs of FF are any parenthesized sequence of numbers and commands, where commands are `+`, `-`, `*`, `/`, and `=`. The `=` command is used to compute the result of an expression, which may be used as the first argument to another binary operator. The `=` may be elided in a string of operations.

[7] The one described here is based on the TI-1025. See [You81] for more details.

`(1 + 20 =)` $\xrightarrow[FF]{}$ `21`
`(1 + 20 = + 300 =)` $\xrightarrow[FF]{}$ `321`
`(1 + 20 + 300 =)` $\xrightarrow[FF]{}$ `321` {*Note elision of first* `=`.}
`(1 + 20)` $\xrightarrow[FF]{}$`20` {*Last number returned when no final* `=`.}

Other features supported by FF include:

- *Calculation with a constant.* Typing a number followed by `=` uses the number as the first operand in a calculation with the previous operator and second operand:

  `(2 * 5 =)` $\xrightarrow[FF]{}$ `10`
  `(2 * 5 = 7 =)` $\xrightarrow[FF]{}$ `35`
  `(2 * 5 = 7 = 11 =)` $\xrightarrow[FF]{}$ `55`

- *Implied second argument.* If no second argument is specified, the value of the second argument defaults to the first.

  `(5 * =)` $\xrightarrow[FF]{}$ `25`

- *Operator correction.* An operator key can be corrected by typing the correct one after (any number of) unintentional operators.

  `(1 * - + 2 )` $\xrightarrow[FF]{}$ `3`

a. Design an SOS for FF that is consistent with the informal description given above.

b. Use your SOS to find the final values of the following command sequences. (Note: some of the values may be floating point numbers.) Comment on the intended meaning of the unconventional command sequences.

   i. `(8 - 3 + * 4 =)`
   ii. `(3 + 5 / = =)`
   iii. `(3 + 5 / = 6 =)` ◁

3.2.3.3 Progress Rules

Introduction

The commands of PostFix programs are interpreted in a highly linear fashion in Figure 3.4. Even though executable sequences give the code a kind of tree structure, the contents of an executable sequence can only be used when they are prepended to the single stream of commands that is executed by the abstract machine. The fact that the next command to execute is always at the front of this command stream leads to a very simple structure for the rewrite rules in Figure 3.4. Transitions, which appear only in rule consequents, are all of the form

$$\langle C_{first} \ . \ Q, \ S\rangle \Rightarrow \langle Q', \ S'\rangle,$$

where Q' is either the same as Q or is the result of prepending some commands onto the front of Q. In all rules, the command C_{first} at the head of the current command sequence is consumed by the application of the rule.

These simple kinds of rules are not adequate for programming languages exhibiting a more general tree structure. Evaluating a node in an arbitrary syntax tree usually requires the recursive evaluation of its subnodes. For example, consider the evaluation of a sample numerical expression written in the EL language described in Section 2.3:

```
(+ (* (- 5 1) 2) (/ 21 7)).
```

Before the sum can be performed, the results of the product and division must be computed; before the multiplication can be performed, the subtraction must be computed. If the values of operand expressions are computed in a left-to-right order, we expect the evaluation of the expression to occur via the following transition path:

```
(+ (* (- 5 1) 2) (/ 21 7))
⇒ (+ (* 4 2) (/ 21 7))
⇒ (+ 8 (/ 21 7))
⇒ (+ 8 3)
⇒ 11.
```

In each transition, the structure of the expression tree remains unchanged except at the node where the computation is being performed. Rewrite rules for expressing such transitions need to be able to express a transition from tree to tree in terms of transitions between the subtrees. That is, the transition

```
(+ (* (- 5 1) 2) (/ 21 7)) ⇒ (+ (* 4 2) (/ 21 7))
```

is implied by the transition

```
(* (- 5 1) 2) ⇒ (* 4 2),
```

which is in turn is implied by the transition

```
(- 5 1) ⇒ 4.
```

In some sense, "real work" is only done by the last of these transitions; the other transitions just inherit the change because they define the surrounding context in which the change is embedded.

These kinds of transitions on tree-structured programs are expressed by progress rules, which are rules with antecedents. Progress rules effectively allow an evaluation process to reach inside a complicated expression to evaluate one of

its subexpressions. A one-step transition in the subexpression is then reflected as a one-step transition of the expression in which it is embedded.

Example: ELMM

To illustrate progress rules, we will develop an operational semantics for an extremely simple subset of the EL language that we will call ELMM (which stands for EL MINUS MINUS). As shown in Figure 3.6, an ELMM program is just a numerical expression, where a numerical expression is either (1) an integer numeral or (2) an arithmetic operation. There are no arguments, no conditional expressions, and no boolean expressions in ELMM.

Syntactic Domains:

$P \in$ Program
$NE \in$ NumExp
$N \in$ IntegerLiteral $= \{\texttt{-17}, \texttt{0}, \texttt{23}, \ldots\}$
$A \in$ ArithmeticOperator $= \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$

Production Rules:

P	::= `(elmm` NE_{body}`)`	[Program]
NE	::= N_{num}	[IntLit]
	| `(`A_{rator} NE_{rand1} NE_{rand2}`)`	[Arithmetic Operation]

Figure 3.6: An s-expression grammar for ELMM.

In an SOS for ELMM, configurations are just numerical expressions themselves; there are no state components. Numerical literals are the only final configurations. The input and output functions are straightforward. The interesting aspect of the ELMM SOS is the specification of the transition relation $\Rightarrow$, which is shown in Figure 3.7. The ELMM [*arithop*] axiom is similar to the same-named axiom in the POSTFIX SOS; it performs a calculation on integer numerals.

To evaluate expressions with nested subexpressions in a left-to-right order, the rules [*prog-left*] and [*prog-left*] are needed. The [*prog-left*] rule says that if the ELMM abstract machine would make a transition from NE_1 to NE_1', it should also allow a transition from `(arithop` NE_1 NE_2`)` to `(arithop` NE_1' NE_2`)`. This rule permits evaluation of the left operand of the operation while leaving the right operand unchanged. The [*prog-right*] rule is similar, except that it only permits evaluation of the right operand once the left operand has been fully evaluated to an integer numeral. This forces the operands to be evaluated

$$(A\ \ N_1\ \ N_2) \Rightarrow N_{result}, \quad \text{where } N_{result} = (calculate\ A\ N_1\ N_2) \qquad [arithop]$$

$$\frac{NE_1 \Rightarrow NE_1'}{(A\ \ NE_1\ \ NE_2) \Rightarrow (A\ \ NE_1'\ \ NE_2)} \qquad [prog\text{-}left]$$

$$\frac{NE_2 \Rightarrow NE_2'}{(A\ \ N\ \ NE_2) \Rightarrow (A\ \ N\ \ NE_2')} \qquad [prog\text{-}right]$$

Figure 3.7: Rewrite rules defining the transition relation ($\Rightarrow$) for ELMM.

in a left-to-right order. Rules like [*prog-left*] and [*prog-left*] are called **progress rules** because an evaluation step performed on a subexpression allows progress to be made on the evaluation of the whole expression.

In the case of axioms, it was easy to determine the set of transitions that were specified by a rule. But how do we determine exactly what set of transitions are specified by a progress rule? Intuitively, a transition is specified by a progress rule if it matches the consequent of the rule and it's possible to show that the antecedent transition patterns are also satisfied. For example, since the ELMM transition `(- 7 4)` $\Rightarrow$ `3` is justified by the [*arithop*] rule, the transition `(* (- 7 4) (+ 5 6)` $\Rightarrow$ `(* 3 (+ 5 6))` is justified by the [*prog-left*] rule, and the transition `(* 2 (- 7 4))` $\Rightarrow$ `(* 2 3)` is justified by the [*prog-right*] rule. Furthermore, since the above transitions themselves satisfy the antecedents of the [*prog-left*] and [*prog-right*] rules, it is possible to use these rules again to justify the following transitions:

```
(/ (* (- 7 4) (+ 5 6) (% 9 2))) ⇒ (/ (* 3 (+ 5 6) (% 9 2)))
(/ (* 2 (- 7 4)) (% 9 2)) ⇒ (/ (* 2 3) (% 9 2))
(/ 100 (* (- 7 4) (+ 5 6))) ⇒ (/ 100 (* 3 (+ 5 6)))
(/ 100 (* 2 (- 7 4))) ⇒ (/ 100 (* 2 3))
```

These examples suggest that we can justify any transition as long as we can give a proof of the transition based upon the rewrite rules. Such a proof can be visualized as a so-called **proof tree** (also known as a **derivation**) that grows upward from the bottom of the page. The root of a proof tree is the transition we are trying to prove, its intermediate nodes are instantiated progress rules, and its leaves are instantiated axioms. A proof tree is structured so that the consequent of each instantiated rule is one antecedent of its parent (below) in the tree. For example, the proof tree associated with the transition of `(/ 100 (* (- 7 4) (+ 5 6)))` appears in Figure 3.8. We can represent the

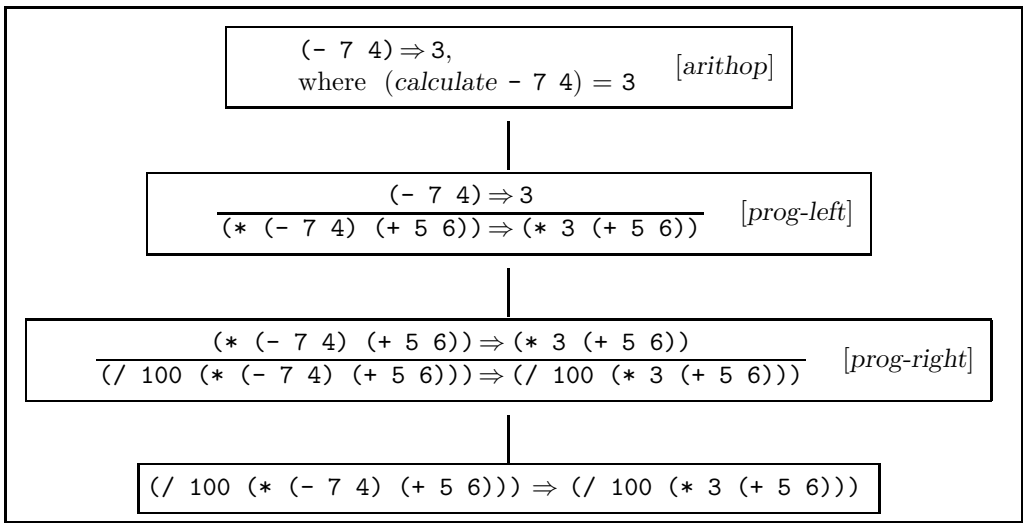


Figure 3.8: A proof tree for a ELMM transition involving nested expressions. The root of the tree is at the bottom of the page; the leaf is at the top.

proof tree in the figure much more concisely by displaying each transition only once, as shown below:

$$\dfrac{\dfrac{\dfrac{}{\texttt{(- 7 4)} \Rightarrow \texttt{3}}\,[\textit{arithop}]}{\texttt{(* (- 7 4) (+ 5 6))} \Rightarrow \texttt{(* 3 (+ 5 6))}}\,[\textit{prog-left}]}{\texttt{(/ 100 (* (- 7 4) (+ 5 6)))} \Rightarrow \texttt{(/ 100 (* 3 (+ 5 6)))}}\,[\textit{prog-right}]$$

The proof tree in this particular example is linear because each of the progress rules involved has only one antecedent transition pattern. A progress rule with n antecedent transition patterns would correspond to a tree node with a branching factor of n. For example, suppose we added the following progress rule to the ELMM SOS:

$$\dfrac{NE_1 \Rightarrow NE_1' \;;\; NE_2 \Rightarrow NE_2'}{\texttt{(arithop } NE_1 \;\; NE_2\texttt{)} \Rightarrow \texttt{(arithop } NE_1' \;\; NE_2'\texttt{)}}\,[\textit{prog-both}]$$

This rule allows simultaneous evaluation of both operands. It leads to proof trees that have branching, such as the following tree in which three arithmetic

operations are performed simultaneously:

$$\dfrac{\dfrac{}{\texttt{(+ 25 75)} \Rightarrow \texttt{100}}\,[arithop] \quad \dfrac{\dfrac{}{\texttt{(- 7 4)} \Rightarrow \texttt{3}}\,[arithop] \quad \dfrac{}{\texttt{(+ 5 6)} \Rightarrow \texttt{11}}\,[arithop]}{\texttt{(* (- 7 4) (+ 5 6))} \Rightarrow \texttt{(* 3 11)}}\,[prog\text{-}both]}{\texttt{(/ (+ 25 75) (* (- 7 4) (+ 5 6)))} \Rightarrow \texttt{(/ 100 (* 3 11))}}\,[prog\text{-}both]$$

It is possible to express any proof tree (even one with branches) in the more traditional linear textual style for a proof. In this style, a proof of a transition is a sequence of transitions where each transition is justified either by an axiom or by a progress rule whose antecedent transitions are justified by transitions earlier in the sequence. A linear textual version of the branching proof tree above would be:

#	**Transition**	**Justification**
[1]	`(+ 25 75)` ⇒`100`	[*arithop*]
[2]	`(- 7 4)` ⇒`3`	[*arithop*]
[3]	`(+ 5 6)` ⇒`11`	[*arithop*]
[4]	`(* (- 7 4) (+ 5 6))` ⇒`(* 3 11)`	[*prog-both*] & [2] & [3]
[5]	`(/ (+ 25 75) (* (- 7 4) (+ 5 6)))` ⇒ `(/ 100 (* 3 11))`	[*prog-both*] & [1] & [4]

The elements of the linear textual proof sequence have been numbered, and justifications involving progress rules include the numbers of the transitions matched by their antecedents. There are many alternative proof sequences for this example that differ in the ordering of the elements. Indeed, the legal linear textual proof sequences for this example are just topological sorts of the original proof tree. Because such linearizations involve making arbitrary choices, we prefer to use the tree based notation, whose structure highlights the essential dependencies in the proof.

When writing down a transition sequence to show the evaluation of an ELMM expresssion we will not explicitly justify every transition with a proof tree, even though such a proof tree must exist. However, if we are listing justifications for transitions, then we will list the names of the rules that would be needed to perform the proof. See Figure 3.9 for an example. (This example uses the original SOS, which does not include the [*prog-both*] rule.)

We shall see in Section 3.4 that the fact that each transition has a proof tree is key to proving properties about transitions. Transition properties are often proven by structural induction on the structure of the proof tree for the transition.

▷ **Exercise 3.10**

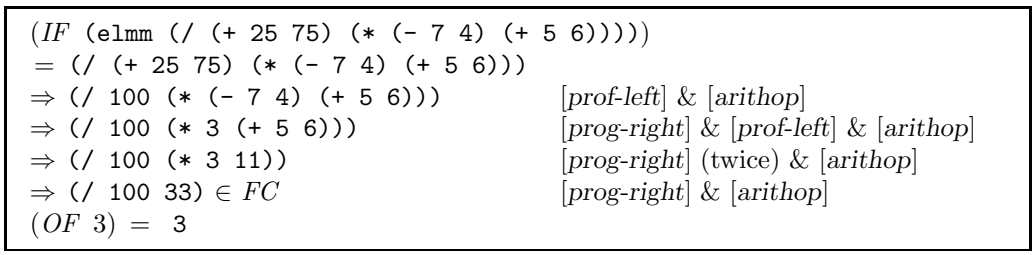


Figure 3.9: An example illustrating evaluation of ELMM expressions.

a. Consider a language ELM (short for EL MINUS) that extends ELMM with indexed references to program inputs. The syntax for ELM is like that of ELMM except that (1) ELM programs have the form `(elm` $N_{numargs}$ NE_{body}`)`, where $N_{numargs}$ specifies the number of expected program arguments and (2) numerical expressions are extended with EL's `(arg` N_{index}`)` construct, which gives the value of the argument whose index is given by N_{index} (assume indices start at 1).

   Write a complete SOS for ELM. Your configurations will need to include a state component representing the program arguments.

b. Write a complete SOS for the full EL language described in Section 2.3.2. You will need to define two kinds of configurations: one to handle numeric expressions and one to handle boolean expressions. Each kind of configuration will be a pair of an expression and a sequence of numeric arguments and will have its own transition relation. ◁

Example: POSTFIX

As another example of progress rules, we will consider an alternative approach for describing the `exec` command of POSTFIX. The [*execute*] axiom in Figure 3.4 handled `exec` by popping an executable sequence off the stack and prepending it to the command sequence following the `exec` command. Figure 3.10 presents a progress rule, [*exec-prog*], that, together with the axiom [*exec-done*], can replace the [*execute*] rule. The [*exec-prog*] rule says that if the abstract machine would make a transition from configuration $\langle Q_{exec},\ S\rangle$ to configuration $\langle Q_{exec}{}',\ S'\rangle$ then it should also allow a transition from the configuration $\langle$`exec .` $Q_{rest},\ Q_{exec}$ `.` $S\rangle$ to $\langle$`exec .` $Q_{rest},\ Q_{exec}{}'$ `.` $S'\rangle$.

Rather than prepending the commands in Q_{exec} to Q_{rest}, the [*exec-prog*] rule effectively executes the commands in Q_{exec} while it remains on the stack. Note that, unlike all the rules that we have seen before, this rule does *not* remove the `exec` command from the current command sequence. Instead, the `exec` command is left in place so that the execution of the command sequence at the

$$\frac{\langle Q_{exec},\ S\rangle \Rightarrow \langle Q_{exec}{}',\ S'\rangle}{\langle \texttt{exec . } Q_{rest},\ Q_{exec}\ \texttt{ . } S\rangle \Rightarrow \langle \texttt{exec . } Q_{rest},\ Q_{exec}{}'\ \texttt{ . } S'\rangle} \quad [\textit{exec-prog}]$$

$$\langle \texttt{exec . } Q_{rest},\ \texttt{() . } S\rangle \Rightarrow \langle Q_{rest},\ S\rangle \quad [\textit{exec-done}]$$

Figure 3.10: A pair of rules that could replace the [*execute*] axiom.

top of the stack will continue during the next transition. Since the commands are removed from Q_{exec} after being executed, the executable sequence at the top of the stack will eventually become empty. At this point, the [*exec-done*] rule takes over, and removes both the completed `exec` command and its associated empty executable sequence.

Figure 3.11 shows how the example considered earlier in Figure 3.1 and Figure 3.5 would be handled using the [*exec-prog*] and [*exec-done*] rules. Each transition is justified by a proof tree that uses the rules listed as a justification. For example, the transition

$$\begin{array}{l}\langle \texttt{(exec sub), [(exec), (mul add), 3, 2, 1, 4, 5]}\rangle \\ \quad \Rightarrow \langle \texttt{(exec sub), [(exec), (add), 6, 1, 4, 5]}\rangle\end{array}$$

is justified by the following proof tree:

$$\dfrac{\dfrac{\dfrac{}{\langle \texttt{(mul add), [3, 2, 1, 4, 5]}\rangle \Rightarrow \langle \texttt{(add), [6, 1, 4, 5]}\rangle}\ [\textit{arithop}]}{\langle \texttt{(exec), [(mul add), 3, 2, 1, 4, 5]}\rangle \Rightarrow \langle \texttt{(exec), [(add), 6, 1, 4, 5]}\rangle}\ [\textit{exec-prog}]}{\begin{array}{c}\langle \texttt{(exec sub), [(exec), (mul add), 3, 2, 1, 4, 5]}\rangle \\ \Rightarrow \langle \texttt{(exec sub), [(exec), (add), 6, 1, 4, 5]}\rangle\end{array}}\ [\textit{exec-prog}]$$

The Meaning of Progress Rules

There are some technical details about progress rules that we glossed over earlier. When we introduced progress rules, we blindly assumed that they were always reasonable. But not all progress rules make sense.

For example, suppose we extend PostFix with a `loop` command defined by the following progress rule:

$$\frac{\langle \texttt{loop . } Q,\ S\rangle \Rightarrow \langle Q,\ S\rangle}{\langle \texttt{loop . } Q,\ S\rangle \Rightarrow \langle Q,\ S\rangle} \quad [\textit{loop}]$$

$(IF\ \langle$`(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)`$, [4,5]\rangle)$
$= \langle$`((2 (3 mul add) exec) 1 swap exec sub), [4, 5]`$\rangle$
$\Rightarrow \langle$`(1 swap exec sub), [(2 (3 mul add) exec), 4, 5]`$\rangle$ [*seq*]
$\Rightarrow \langle$`(swap exec sub), [1, (2 (3 mul add) exec), 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(exec sub), [(2 (3 mul add) exec), 1, 4, 5]`$\rangle$ [*swap*]
$\Rightarrow \langle$`(exec sub), [((3 mul add) exec), 2, 1, 4, 5]`$\rangle$ [*exec-prog*] & [*num*]
$\Rightarrow \langle$`(exec sub), [(exec), (3 mul add), 2, 1, 4, 5]`$\rangle$ [*exec-prog*] & [*seq*]
$\Rightarrow \langle$`(exec sub), [(exec), (mul add), 3, 2, 1, 4, 5]`$\rangle$ [*exec-prog*] (twice) & [*num*]
$\Rightarrow \langle$`(exec sub), [(exec), (add), 6, 1, 4, 5]`$\rangle$ [*exec-prog*] (twice) & [*arithop*]
$\Rightarrow \langle$`(exec sub), [(exec), (), 7, 4, 5]`$\rangle$ [*exec-prog*] (twice) & [*arithop*]
$\Rightarrow \langle$`(exec sub), [(), 7, 4, 5]`$\rangle$ [*exec-prog*] & [*exec-done*]
$\Rightarrow \langle$`(sub), [7, 4, 5]`$\rangle$ [*exec-done*]
$\Rightarrow \langle$`(), [-3]`$\rangle \in FC$ [*arithop*]
$(OF\ \langle$`(),[-3]`$\rangle) =$ `-3`

Figure 3.11: An example illustrating the alternative rules for `exec`.

Any attempt to prove a transition involving `loop` will fail because there are no axioms involving `loop` with which to terminate the proof tree. Thus, this rule stands for no transitions whatsoever!

We'd like to ensure that all progress rules we consider make sense. We can guarantee this by restricting the form of allowable progress rules to outlaw nonsensical rules like [*loop*]. This so-called **structure restriction** guarantees that any attempt to prove a transition from a given configuration will eventually terminate. The standard structure restriction for an SOS requires the code component of the LHS of each antecedent transition to be a subphrase of the code component of the LHS of the consequent transition. Since program parse trees are necessarily finite, this guarantees that all attempts to prove a transition will have a finite proof.[8]

While simple to follow, the standard structure restriction prohibits many reasonable rules. For example, the [*exec-prog*] rule does not obey this restriction, because the code component of the LHS of the antecedent is unrelated to the code component of the LHS of the consequent. Yet, by considering the entire configuration rather than just the code component, it is possible to design a metric in which the LHS of the antecedent is "smaller" than the LHS of the

[8]This restriction accounts for the term "Structured" in Structured Operational Semantics.

consequent (see Exercise 3.11). While it is sometimes necessary to extend the standard structure restriction in this fashion, most of our rules will actually obey the standard version.

▷ **Exercise 3.11** To guarantee that a progress rule is well-defined, we must show that the antecedent configurations are smaller than the consequent configurations. Here we explore a notion of "smaller than" for the PostFix configurations that establishes the well-definedness of the [*exec-prog*] rule. (Since [*exec-prog*] is the only progress rule for PostFix, it is the only one we need to consider.)

Suppose that we define a relation $<$ on PostFix configurations such that

$$\langle Q_1, \ S\rangle < \langle \texttt{exec} \ . \ Q_2, \ \ Q_1 \ . \ S\rangle$$

for any command sequences Q_1 and Q_2 and any stack S. This is the *only* relation on PostFix configurations; two configurations not satisfying this relation are simply incomparable.

a. A sequence $[a_1, a_2, \ldots]$ is **strictly decreasing** if $a_{i+1} < a_i$ for all i. Using the relation $<$ defined above for configurations, show that every strictly decreasing sequence $[cf_1, cf_2, \ldots]$ of PostFix configurations must be finite.

b. Explain how the result of the previous part implies the well-definedness of the [*exec-prog*] rule. ◁

▷ **Exercise 3.12** The abstract machine for PostFix described thus far employs configurations with two components: a command sequence and a stack. It is possible to construct an alternative abstract machine for PostFix in which configurations consist only of a command sequence. The essence of such a machine is suggested by the transition sequence in Figure 3.12, where the primed rule names are the names of rules for the new abstract machine, not the abstract machine presented earlier.

a. The above example shows that an explicit stack component is not necessary to model PostFix evaluation. Explain how this is possible. (Is there an implicit stack somewhere?)

b. Write an SOS for PostFix in which a configuration is just a command sequence. The SOS should have the behavior exhibited above on the given example. Recall that an SOS has five components; describe all five. Use only axioms to specify your transition relation.

c. In the above example, the `exec` command is handled by replacing it and the executable sequence Q to its left by the contents of Q. This mirrors the prepending behavior of [*execute*] in the original abstract machine. Write rules for the new abstract machine that instead mirror the behavior of [*exec-prog*] and [*exec-done*].

d. Develop an appropriate notion of "smaller than" that establishes the well-definedness of your new [*exec-prog*] rule. (See Exercise 3.11.)

```
((swap exec swap exec) (1 sub) swap (2 mul) swap 3 swap exec)
⇒ ((1 sub) (swap exec swap exec) (2 mul) swap 3 swap exec)  [swap′]
⇒ ((1 sub) (2 mul) (swap exec swap exec) 3 swap exec)       [swap′]
⇒ ((1 sub) (2 mul) 3 (swap exec swap exec) exec)            [swap′]
⇒ ((1 sub) (2 mul) 3 swap exec swap exec)                   [exec′]
⇒ ((1 sub) 3 (2 mul) exec swap exec)                        [swap′]
⇒ ((1 sub) 3 2 mul swap exec)                               [exec′]
⇒ ((1 sub) 6 swap exec)                                     [arithop′]
⇒ (6 (1 sub) exec)                                          [swap′]
⇒ (6 1 sub)                                                 [exec′]
⇒ 5                                                         [arithop′]
```

Figure 3.12: Sample transition sequence for an alternative PostFix abstract machine whose configurations are command sequences.

e. Sketch how you might prove that the new SOS and the original SOS define the behavior. ◁

3.2.3.4 Context-based Semantics

Axioms and progress rules are not the only way to specify the transition relation of a small-step operational semantics. Here we introduce another approach to specifying transitions that is popular in the literature. This approach is based on a notion of **context** that specifies the position of a subphrase in a larger program phrase. Here we will explain this notion and show how it can be used to specify transitions.

In general, a context is a phrase with a single **hole** node in the abstract syntax tree for the phrase. A sample context $\mathbb{C}$ in the ELMM language is `(+ 1 (- □ 2))`, where where □ denotes the hole in the context. "Filling" this hole with any ELMM numerical expression yields another numerical expression. For example, filling $\mathbb{C}$ with `(/ (* 4 5) 3)`, written $\mathbb{C}${`(/ (* 4 5) 3)`}, yields the numerical expression `(+ 1 (- (/ (* 4 5) 3) 2))`.

Contexts are useful for specifying a particular occurrence of a phrase that may occur more than once in an expression. For example, `(+ 3 4)` appears twice in `(* (+ 3 4) (/ (+ 3 4) 2))`. The leftmost occurrence is specified by the context `(* □ (/ (+ 3 4) 2))`, while the rightmost one is specified by `(* (+ 3 4) (/ □ 2))`. Contexts are also useful for specifying the part of a phrase that remains unchanged (the **evaluation context**) when a basic computation (known as a **redex**) is performed. For example, consider the evaluation

of the ELMM expression `(/ 100 (* (- 7 4) (+ 5 6)))`. If operands are evaluated in a left-to-right order, the next redex to be performed is `(- 7 4)`. The evaluation context $\mathbb{E}$ for this redex is `(/ 100 (* □ (+ 5 6)))`. The result of performing the redex (`3` in this case) can be plugged into the evaluation context to yield the result of the transition: $\mathbb{E}\{\texttt{3}\}$ =`(/ 100 (* 3 (+ 5 6)))`.

Evaluation contexts and redexes can be defined via grammars, such as the ones for ELMM in Figure 3.13. In ELMM, a redex is an arithmetic operator applied to two integer numerals. An ELMM evaluation context is either a hole or an arithmetic operation one of whose two operands is an evaluation context. If the evaluation context is in the left operand position ([*Eval Left*]) the right operand can be an arbitrary numerical expression. But if the evaluation context is in the right operand position ([*Eval Right*]), the left operand *must* be a numeral. This structure enforces left-to-right evaluation in ELMM in a way similar to the [*prog-left*] and [*prog-right*] progress rules. Indeed, evaluation contexts are just another way of expressing the information in progress rules — namely, how to find the redex (i.e., where an axiom can be applied).

Redexes

$\mathcal{R} \in$ ElmmRedex

$\mathcal{R}$ ::= `(`$A\ N_1\ N_2$`)` [Arithmetic operation]

Reduction relation ($\rightsquigarrow$)

`(`$A\ N_1\ N_2$`)` $\rightsquigarrow N_{result}$, where N_{result} =$(calculate\ A\ N_1\ N_2)$

Evaluation Contexts

$\mathbb{E} \in$ ElmmEvalContext

$\mathbb{E}$ ::= □ [Hole]
| `(`$A\ \mathbb{E}\ NE$`)` [Eval Left]
| `(`$A\ N\ \mathbb{E}$`)` [Eval Right]

Transition relation ($\Rightarrow$)

$\mathbb{E}\{\mathcal{R}\} \Rightarrow \mathbb{E}\{\mathcal{R}'\}$, where $\mathcal{R} \rightsquigarrow \mathcal{R}'$

Figure 3.13: A context-based specification of the ELMM transition relation.

Associated with redexes is a reduction relation ($\rightsquigarrow$) that corresponds to the basic computations axioms we have seen before. The left hand side of the relation is the redex, while the right hand side is the **reduct**. The transition relation ($\Rightarrow$) is defined in terms of the reduction relation using evaluation contexts: the expression $\mathbb{E}\{\mathcal{R}\}$ rewrites to $\mathbb{E}\{\mathcal{R}'\}$ as long as there is a reduction $\mathcal{R} \rightsquigarrow \mathcal{R}'$. The transition relation is deterministic if there is at most one way to parse an

expression into a evaluation context filled with a redex (which is the case in ELMM).

The following table shows the context-based evaluation of an ELMM expression:

Expression	Evaluation Context	Redex	Reduct
`(/ (+ 25 75) (* (- 7 4) (+ 5 6)))`	`(/ □ (* (- 7 4) (+ 5 6)))`	`(+ 25 75)`	`100`
⇒ `(/ 100 (* (- 7 4) (+ 5 6)))`	`(/ 100 (* □ (+ 5 6)))`	`(- 7 4)`	`3`
⇒ `(/ 100 (* 3 (+ 5 6)))`	`(/ 100 (* 3 □))`	`(+ 5 6)`	`11`
⇒ `(/ 100 (* 3 11))`	`(/ 100 □)`	`(* 3 11)`	`33`
⇒ `(/ 100 33)`	□	`(/ 100 33)`	`3`
⇒ `3`			

Context-based semantics are most convenient in an SOS where the configurations consist solely of a code component. But they can also be adapted to configurations that have state components. For example, Figure 3.14 is a context-based semantics for ELM, the extension to ELMM that includes indexed input via the form `(arg` N_{index}`)` (see Exercise 3.10). An ELM configuration is a pair of (1) an ELM numerical expression and (2) a sequence of numerals representing the program arguments. Both the ELM reduction relation and transition relation must include the program arguments so that the `arg` form can access them.

▷ **Exercise 3.13** Starting wih Figure 3.14, develop a context-based semantics for the full EL language. ◁

▷ **Exercise 3.14** The most natural context-based semantics for PostFix is based on the approach sketched in Exercise 3.12, where configurations consist only of a command sequence. Figure 3.15 is the skeleton of a context-based semantics that defines the transition relation for these configurations. It uses a command sequence context $\mathbb{EQ}$ whose hole can be filled with a command sequence that is internally appended to other command sequences. For example, if $\mathbb{EQ} = [1, 2, \square, \texttt{sub}]$, then $\mathbb{EQ}\{[3, \texttt{swap}]\} = [1, 2, 3, \texttt{swap}, \texttt{sub}]$. Complete the semantics in Figure 3.15 by fleshing out the missing details. ◁

3.3 Big-step Operational Semantics

A small-step operational semantics is a framework for describing program execution as an iterative sequence of small computational steps. But this is not always the most natural way to view execution. We often want to evaluate a phrase

Redexes

$\mathcal{R} \in$ ElmRedex

$\mathcal{R}$ ::= `(`$A\ \ N_1\ \ N_2$`)` [Arithmetic operation]
| `(arg` N_{index}`)` [Indexed Input]

Reduction relation ($\rightsquigarrow$)

$\langle$`(`$A\ \ N_1\ \ N_2$`)`$, N^*\rangle \rightsquigarrow N_{result}$ where $N_{result} =$(*calculate* $A\ \ N_1\ \ N_2$)
$\langle$`(arg` N_{index}`)`$, [N_1, \ldots, N_{N_{size}}]\rangle \rightsquigarrow N_{N_{index}}$
where (*compare* `>` N_{index} `0`) $\wedge \neg$ (*compare* `>` $N_{index}\ \ N_{size}$)

Evaluation Contexts

$\mathbb{E} \in$ ElmEvalContext

$\mathbb{E}$ ::= $\Box$ [Hole]
| `(`$A\ \ \mathbb{E}\ \ NE$`)` [Eval Left]
| `(`$A\ \ N\ \ \mathbb{E}$`)` [Eval Right]

Transition relation ($\Rightarrow$)

$\langle \mathbb{E}\{\mathcal{R}\}, N^*\rangle \Rightarrow \langle \mathbb{E}\{\mathcal{R}'\}, N^*\rangle$ where $\langle \mathcal{R}, N^*\rangle \rightsquigarrow \mathcal{R}'$

Figure 3.14: A context-based specification of the ELM transition relation.

Redexes

$\mathcal{R} \in$ PostFixRedex

$\mathcal{R}$ ::= $[V, \texttt{pop}]$ [Pop]
| $[V_1, V_2, \texttt{swap}]$ [Swap]
| $[N_1, N_2, A]$ [Arithmetic operation]
| ... left as an exercise ...

Reduction relation ($\rightsquigarrow$)

$[V, \texttt{pop}] \rightsquigarrow [\,]$
$[V_1, V_2, \texttt{swap}] \rightsquigarrow [V_2, V_1]$
$[N_1, N_2, A] \rightsquigarrow [N_{result}]$ where $N_{result} =$(*calculate* $A\ \ N_2\ \ N_1$)
... left as an exercise ...

Evaluation Contexts

$\mathbb{EQ} \in$ PostfixEvalSequenceContext

$\mathbb{EQ}$ ::= $V^*\ @\ \Box\ @\ Q$

Transition relation ($\Rightarrow$)

$\mathbb{EQ}\{\mathcal{R}\} \Rightarrow \mathbb{EQ}\{\mathcal{R}'\}$, where $\mathcal{R} \rightsquigarrow \mathcal{R}'$

Figure 3.15: A context-based specification of the transition relation for a subset of PostFix.

by recursively evaluating its subphrases and then combining the results. This is the key idea of denotational semantics, which we shall study in Chapter 4. However, this idea also underlies an alternative form of operational semantics, called **big-step operational semantics (BOS)** (also known as **natural semantics**). Here we briefly introduce big-step semantics in the context of a few examples.

Let's begin by defining a BOS for the simple expression language ELMM, in which programs are numerical expressions that are either numerals or arithmetic operations. A BOS typically has an **evaluation relation** for each non-trivial syntactic domain that directly specifies a result for a given program phrase or configuration. The BOS in Figure 3.16 defines two evaluation relations:

1. $\rightarrow_{NE} \in$ NumExp $\times$ Intlit specifies the evaluation of an ELMM numerical expression; and

2. $\rightarrow_{P} \in$ Program $\times$ Intlit specifies the evaluation of an ELMM program.

$$\frac{NE \rightarrow_{NE} N_{ans}}{\texttt{(elmm } NE\texttt{)} \rightarrow_{P} N_{ans}} \quad [prog]$$

$$N \rightarrow_{NE} N \quad [num]$$

$$\frac{NE_1 \rightarrow_{NE} N_1 \quad ; \quad NE_2 \rightarrow_{NE} N_2}{\texttt{(}A\ NE_1\ NE_2\texttt{)} \rightarrow_{NE} N_{result}} \quad [arithop]$$

where $N_{result} = (calculate\ A\ N_1\ N_2)$

Figure 3.16: Big-step operational semantics for ELMM.

There are two rules specifying $\rightarrow_{NE}$. The [*num*] rule says that numerals evaluate to themselves. The [*arithop*] rule says that evaluating an arithmetic operation $\texttt{(}A\ N_1\ N_2\texttt{)}$ yields the result (N_{result}) of applying the operator to the results (N_1 and N_2) of evaluating the operands. The single [*prog*] rule specifying $\rightarrow_P$ just says that the result of an ELMM program is the result of evaluating its numerical expression.

As with SOS transitions, each instantiation of a BOS evaluation rule is justified by a proof tree, which we shall call an **evaluation tree**. Below is the

proof tree for the evaluation of the program `(elmm (* (- 7 4) (+ 5 6)))`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{7} \longrightarrow_{NE} \texttt{7}}[num] \quad \dfrac{}{\texttt{4} \longrightarrow_{NE} \texttt{4}}[num]}{\texttt{(- 7 4)} \longrightarrow_{NE} \texttt{3}}[arithop] \quad \dfrac{\dfrac{}{\texttt{5} \longrightarrow_{NE} \texttt{5}}[num] \quad \dfrac{}{\texttt{6} \longrightarrow_{NE} \texttt{6}}[num]}{\texttt{(+ 5 6)} \longrightarrow_{NE} \texttt{11}}[arithop]}{\texttt{(* (- 7 4) (+ 5 6))} \longrightarrow_{NE} \texttt{33}}[arithop]}{\texttt{(elmm (* (- 7 4) (+ 5 6)))} \longrightarrow_{P} \texttt{33}}[prog]$$

Unlike the proof tree for an SOS transition, which justifies a single computational step, the proof tree for a BOS transition justifies the entire evaluation! This is the sense in which the steps of a BOS are "big"; they tell how to go from a phrase to an answer (or something close to an answer). In the case of ELMM, the leaves of the proof tree are always trivial evaluations of numerals to themselves.

With BOS evaluations there is no notion of a stuck state. In the ELMM BOS, there is no proof tree for an expression like `(* (/ 7 0) (+ 5 6))` that contains an error. However, we can extend the BOS to include an explicit error token as a possible result and modify the rules to generate and propagate such a token. Since all ELMM programs terminate, a BOS with this extension completely specifies the behavior of a program. But in general, the top-level evaluation rule for a program only partially specifies its behavior, since there is no tree (not even an infinite one) asserting that a program loops. What would the answer A of such a program be in the relation $P \longrightarrow_P A$?

The ELMM BOS rules also do not specify the order in which operands are evaluated, but this is irrelevant anyway since there is no way in ELMM to detect whether one operation is performed before another. The ELMM BOS rules happen to specify a (necessarily deterministic) function, but since they can specify general relations, a BOS can describe non-deterministic evaluation as well.

In ELMM, the evaluation relation maps a code phrase to its result. In general, the LHS (and RHS) of an evaluation relation can be more complex, containing state components in addition to a code component. This is illustrated in the BOS for ELM, which extends ELMM with an indexed input construct (Figure 3.17). Here, the two evaluation relations have different domains than before: they include an integer numeral sequence to model the program arguments.

1. $\longrightarrow_{NE} \in$ (NumExp $\times$ Intlit*) $\times$ Intlit specifies the evaluation of an ELM numerical expression; and

2. $\longrightarrow_{P} \in$ (Program $\times$ Intlit*) $\times$ Intlit specifies the evaluation of an ELM program.

Each of these relations can be read as "evaluating a program phrase relative to the program arguments to yield a result". As a notational convenience, we

$$\frac{NE \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_{NE} N_{ans}}{\texttt{(elm } N_{numargs} \ NE\texttt{)} \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_{P} N_{ans}} \ [prog]$$
where $(compare \texttt{ = } N_{numargs} \ N_{size})$

$$N \xrightarrow{N^*}_{NE} N \quad [num]$$

$$\frac{NE_1 \xrightarrow{N^*}_{NE} N_1 \quad ; \quad NE_2 \xrightarrow{N^*}_{NE} N_2}{\texttt{(}A \ NE_1 \ NE_2\texttt{)} \xrightarrow{N^*}_{NE} N_{result}} \ [arithop]$$
where $N_{result} = (calculate \ A \ N_1 \ N_2)$

$$\texttt{(arg } N_{index}\texttt{)} \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_{NE} N_{N_{index}} \quad [input]$$
where $(compare \texttt{ > } N_{index} \ \texttt{0}) \wedge \neg\, (compare \texttt{ > } N_{index} \ N_{size})$

Figure 3.17: Big-step operational semantics for ELM.

abbreviate $\langle X, N_{args}{}^* \rangle \longrightarrow_X N_{ans}$ as X $\xrightarrow{N_{args}{}^*}_X N_{ans}$, where X ranges over P and NE. The [*prog*] rule is as in ELMM, except that it checks that the number of arguments is as expected and passes them to the body for its evaluation. These arguments are ignored by the [*num*] and [*arithop*] rules, but are used by the [*input*] rule to return the specified argument.

Here is a sample ELM proof tree showing the evaluation of the program `(elm 2 (* (arg 1) (+ 1 (arg 2))))` on the two arguments `7` and `5`:

$$\dfrac{\dfrac{}{\texttt{(arg 1)} \xrightarrow{[7,5]}_{NE} \texttt{7}}\,[input] \quad \dfrac{\dfrac{}{\texttt{1} \xrightarrow{[7,5]}_{NE} \texttt{1}}\,[num] \quad \dfrac{}{\texttt{(arg 2)} \xrightarrow{[7,5]}_{NE} \texttt{5}}\,[input]}{\texttt{(+ (arg 2) 1)} \xrightarrow{[7,5]}_{NE} \texttt{6}}\,[arithop]}{\texttt{(elm 2 (* (arg 1) (+ 1 (arg 2))))} \xrightarrow{[7,5]}_{P} \texttt{42}}\,[prog]$$

Can we describe PostFix execution in terms of a BOS? Yes – via the evaluation relations $\longrightarrow_P$ (for programs) and $\longrightarrow_Q$ (for command sequences) in Figure 3.18. The $\longrightarrow_Q$ relation $\in$ (Commands $\times$ Stack) $\times$ Stack treats command sequences as "stack transformers" that map an input stack to an output stack. We abbreviate $\langle Q, S \rangle \longrightarrow_Q S'$ as $Q \xrightarrow{S}_Q S'$. The [*non-exec*] rule "cheats" by using the SOS transition relation $\Rightarrow$ to specify how a non-`exec` command C transforms the stack to S'. Then $\longrightarrow_Q$ specifies how the rest of the commands transform S' into S''. The [*exec*] rule is more interesting because it uses $\longrightarrow_Q$ in both antecedents. The executable sequence commands Q_{exec} transform S to S', while the remaining commands Q_{rest} transform S' to S''. The [*exec*] rule

illustrates how evaluation order (in this case, executing Q_{exec} before Q_{rest}) can be specified in a BOS by "threading" a state component (in this case, the stack) through an evaluation.

$$\frac{Q \xrightarrow{[N_{N_{size}},\ldots,N_1]}_Q N_{ans} \ . \ S}{\texttt{(postfix } N_{numargs} \ Q\texttt{)} \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_P N_{ans}} \quad [prog]$$
$$\text{where } (compare \texttt{ = } N_{numargs} \ N_{size})$$

$$\frac{\langle C \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ S'\rangle \quad ; \quad Q \xrightarrow{S'}_Q S''}{C \ . \ Q \xrightarrow{S}_Q S''} \quad [non\text{-}exec]$$
$$\text{where } C \neq \texttt{exec}$$

$$\frac{Q_{exec} \xrightarrow{S}_Q S' \quad ; \quad Q_{rest} \xrightarrow{S'}_Q S''}{\texttt{exec} \ . \ Q_{rest} \xrightarrow{(Q_{exec}) \ . \ S}_Q S''} \quad [exec]$$

Figure 3.18: Big-step operational semantics for POSTFIX.

It is convenient to define $\rightarrow_Q$ so that it returns a stack, but stacks are not the final answer we desire. The [*prog*] rule $\in$ (Program $\times$ Intlit*) $\times$ Stack takes care of creating the initial stack from the arguments and extracting the top integer (if it exists) from the final stack.

How do small-step and big-step semantics stack up against each other? Each has its advantages and limitations. A big-step semantics is often more concise than a small-step semantics and one of its proof trees can summarize the entire execution of a program. The recursive nature of a big-step semantics also corresponds more closely to structure of interpreters for high-level languages than a small-step semantics. On the other hand, the iterative step-by-step nature of a small-step semantics corresponds more closely to the way low-level languages are implemented, and it is often a better framework for reasoning about computational resources, errors, and termination. Furthemore, infinite loops are easy to model in a small-step semantics but not in a big-step semantics.

We will use small-step semantics as our default form of operational semantics throughout the rest of this book. This is not because big-step semantics are not useful — they are — but because we will tend to use denotational semantics rather than big-step operational semantics for language specifications that compose the meanings of whole phrases from subphrases.

▷ **Exercise 3.15** Construct a BOS evaluation tree that shows the evaluation of `(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)` on arguments `4` and `5`. ◁

▷ **Exercise 3.16** Extend the BOS in Figure 3.16 to handle the full EL language. You will need a new evaluation relation, $\rightarrow_{BE}$, to handle boolean expressions. ◁

▷ **Exercise 3.17** Modify each of the BOS specifications in Figures 3.16–3.18 to generate and propagate an error token that models signalling an error. Be careful to handle all error situations. ◁

3.4 Operational Reasoning

3.4.1 Programming Language Properties

The suitability of a programming language for a given purpose largely depends on many high-level properties of the language. Important global properties of a programming language include:

- **universality**: the language can express all computable programs;
- **determinism**: the set of possible outcomes from executing a program on any particular inputs is a singleton;
- **termination**: all programs are guaranteed to terminate (i.e., it is not possible to express an infinite loop);
- **static checkability**: a class of program errors can be found by static analysis without resorting to execution;
- **referential transparency**: different occurrences of an expression within the same context always have the same meaning.

Languages often exhibit equivalence properties that allow **safe transformations**: systematic substitutions of one program phrase for another that are guaranteed not to change the behavior of the program. Finally, properties of *particular* programs are often of interest. For instance, we might want to show that a given program terminates, that it uses only bounded resources, or that it is equivalent to some other program. For these sorts of purposes, an important characteristic of a language is how easy it is to prove properties of particular programs written in a language.

A language exhibiting a desired list of properties may not always exist. For example, no language can be both universal and terminating because a universal language must be able to express infinite loops.[9]

[9] But it is often possible to carve a terminating sublanguage out of a universal language.

The properties of a programming language are important to language designers, implementers, and programmers alike. The features included in a language strongly depend on what properties the designers want the language to have. For example, designers of a language in which all programs are intended to terminate cannot include general looping constructs, while designers of a universal language must include features that allow nontermination. Compiler writers extensively use safe transformations to automatically improve the efficiency of programs. The properties of a language influence which language a programmer chooses for a task as well as what style of code the programmer writes.

An important benefit of a formal semantics is that it provides a framework that facilitates proving properties both about the entire language and about particular programs written in the language. Without a formal semantics, our understanding of such properties would be limited to intuitions and informal (and possibly incorrect) arguments. A formal semantics is a shared language for convincing both ourselves and others that some intuition that we have about a program or a language is really true. It can also help us develop new intuitions. It is useful not only to the extent that it helps us construct proofs but also to the extent that it helps us find holes in our arguments. After all, some of the things we think we can prove simply aren't true. The process of constructing a proof can give us important insight into *why* they aren't true.

Below we use operational semantics to reason about EL and PostFix. We first discuss the deterministic behavior of EL under various conditions. Then we show that all PostFix programs are guaranteed to terminate. We conclude by considering conditions under which we can transform one PostFix command sequence to another without changing the behavior of a program.

3.4.2 Deterministic Behavior of EL

Recall that a programming language is deterministic if there is exactly one possible outcome for any pair of program and inputs. In Section 3.2.1, we saw that a deterministic SOS transition relation implies that programs behave deterministically. In Section 3.2.3.1, we argued that the PostFix transition relation is deterministic, so PostFix is a deterministic language.

We can similarly argue that EL is deterministic. We will give the argument for the sublanguage ELMM, but it can be extended to full EL. We will use the SOS for ELMM given in Figure 3.7, which has just three rules: [*arithop*], [*prog-left*], and [*prog-right*]. For a given ELMM numerical expression *NE*, we argue that there is at most one proof tree justifying a transition for *NE*. The proof is by structural induction on the height of the AST for *NE*.

- (Base cases) If *NE* is a numeral, it matches no rules, so there is no transition. If *NE* has the form `(`$A\ \ N_1\ \ N_2$`)`, it can match only the [*arithop*] rule, since there are no transitions involving numerals.
- (Induction cases) *NE* must have the the form `(`$A\ \ NE_1\ \ NE_2$`)`, where at least one of NE_1 and NE_2 is not a numeral. If NE_1 is not a numeral, then *NE* can match only the [*prog-left*] rule, and only in the case where there is a proof tree justifying the transition $NE_1 \Rightarrow NE_1{}'$. By induction, there is at most one such proof tree, so there is at most one proof tree for a transition of *NE*. If NE_1 is a numeral, then NE_2 must not be a numeral, in which case *NE* can match only the [*prog-right*] rule, and similar reasoning applies.

Alternatively, we can prove the determinism of the ELMM transition relation using the context semantics in Figure 3.13. In this case, we need to show that each ELMM numerical expression can be parsed into an evaluation context and redex in at most one way. Such a proof is essentially the same as the one given above, so we omit it.

The ELMM SOS specifies that operations are performed in left-to-right order. Why does the order of evaluation matter? It turns out that it doesn't — there is no way in ELMM to detect the order in which operations are performed! Intuitively, either the evaluation is successful, in which all operations are performed anyway, leading to the same answer, or a divsion/remainder by zero is encountered somewhere along the way, in which case the evaluation is unsuccessful. Note that if we could distinguish between different kinds of errors, the story would be different. For instance, if divide-by-zero gave a different error from remainder-by-zero, then evaluating the expression `(+ (/ 1 0) (% 2 0))` would indicate which of the two subexpressions was evaluated first. The issue of evaluation order is important to implementers, because they sometimes can make program execute more efficiently by reordering operations.

How can we formally show that evaluation order in ELMM does not matter? We begin by replacing the [*prog-right*] rule in the SOS by the following [*prog-right′*] rule to yield a modified ELMM transition relation $\Rightarrow'$.

$$\frac{NE_2 \Rightarrow' NE_2{}'}{(A\ \ NE_1\ \ NE_2) \Rightarrow' (A\ \ NE_1\ \ NE_2{}')} \qquad [\textit{prog-right}']$$

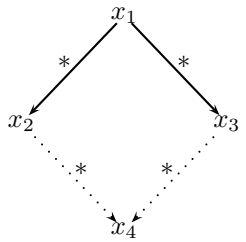
With this change, operands can be evaluated in either order, so the transition relation is no longer deterministic. For example, the expression `(* (- 7 4) (+ 5 6))` now has two transitions:

`(* (- 7 4) (+ 5 6))` $\Rightarrow'$ `(* 3 (+ 5 6))`
`(* (- 7 4) (+ 5 6))` $\Rightarrow'$ `(* (- 7 4) 11)`

Nevertheless, we would like to argue that the behavior of programs is still deterministic even though the transition relation is not.

A handy property for this purpose is called **confluence**. Informally, confluence says that if two transition paths from a configuration diverge, there must be a way to bring them back together. The formal definition is as follows:

> **Confluence:** A relation $\rightarrow \in X \times X$ is confluent if and only if for every $x_1, x_2, x_3 \in X$ such that $x_1 \xrightarrow{*} x_2$ and $x_1 \xrightarrow{*} x_3$, there exists an x_4 such that $x_2 \xrightarrow{*} x_4$ and $x_3 \xrightarrow{*} x_4$. Confluence is usually displayed via the following diagram, in which solid lines are the given relations and the dotted lines are assumed to exist when the property holds. Due to the shape of the diagram, confluence is also called the **diamond property**.



Suppose that a transition relation $\Rightarrow$ is confluent. Then if an initial configuration cf_i has transition paths to two final configurations cf_{f_1} and cf_{f_2}, these are necessarily the same configuration! Why? By confluence, there must be a configuration cf such that $cf_{f_1} \stackrel{*}{\Rightarrow} cf$ and $cf_{f_2} \stackrel{*}{\Rightarrow} cf$. But cf_{f_1} and cf_{f_2} are elements of *Irreducible*, so the only transition paths leaving them have length 0. This means $cf_{f_1} = cf = cf_{f_2}$. Thus, a confluent transition relation guarantees a unique final configuration. Indeed, it guarantees a unique irreducible configuration: it is not possible to get stuck on one path and reach a final configuration on the other.

Confluence by itself does not guarantee a single outcome. It is still possible for a confluent transition relation to have some infinite paths, in which case there is a second outcome (∞). This possibility must be ruled out to prove deterministic behavior. In the case of ELMM, it is easy to prove there are no loops (see Exercise 3.27).

We can now show that ELMM has deterministic behavior under $\Rightarrow'$ by arguing that $\Rightarrow'$ is confluent. We will actually show a stronger property, known as **one-step confluence**, in which the transitive closure stars in the diamond diagram are removed; confluence easily follows from one-step confluence.

Suppose that $NE_1 \Rightarrow' NE_2$ and $NE_1 \Rightarrow' NE_3$. Using terminology from context-based semantics, call the redex reduced in the first transition the "red"

redex and the one reduced in the second transition the "blue" redex. Either these are the same redex, in which case $NE_2 = NE_3$ trivially joins the paths, or the redexes are disjoint (i.e., one does not occur as a subexpression of another). In the latter case, there must be an expression NE_4 that is a copy of NE_1 in which both the red and blue redexes have been reduced. Then $NE_2 \Rightarrow' NE_4$ by reducing the blue redex and $NE_3 \Rightarrow' NE_4$ by reducing the red redex. So NE_4 joins the diverging transitions

We have shown that ELMM has deterministic behavior even when its operations are performed in a non-deterministic order. A similar approach can be used to show that ELM and EL have the same property. Confluence in these languages is fairly straightforward. It becomes much trickier in languages where redexes overlap or performing one redex can copy another.

We emphasize that confluence is a sufficient but not necessary condition for a non-deterministic transition relation to give rise to deterministic behavior. In general, many distinct final configurations might map to the same outcome.

▷ **Exercise 3.18** Suppose that in addition to changing the ELMM SOS by replacing [*prog-right*] with [*prog-right′*], the rule [*prog-both*] introduced on page 58 is added to the SOS.

a. In this modified SOS, how many different transition paths lead from the expression `(/ (+ 25 75) (* (- 7 4) (+ 5 6)))` to the result `3`?

b. Does the modified SOS still have deterministic behavior? Explain your answer ◁

▷ **Exercise 3.19** Consider extending ELMM with a construct `(either` NE_1 NE_2`)` that returns the result of evaluating either NE_1 or NE_2.

a. What are the possible behaviors of the following program?

```
(elmm (* (- (either 1 2) (either 3 4)) (either 5 6)))
```

b. The informal specification of `either` given above is ambiguous. For example, must the expression `(+ (either 1 (/ 2 0)) (either (% 3 0) 4))` return the result `5`, or can it get stuck? The semantics of `either` can be defined either way. Give formal specifications for each interpretation of `either` that is consistent with the informal description. ◁

▷ **Exercise 3.20**

a. Show that the two transition relations (one for NumExp, one for BoolExp) in an EL SOS can be deterministic,

b. Suppose that both transition relations in an EL SOS allow operations to be performed in any order, so that they are non-deterministic. Argue that the behavior of EL programs is still deterministic. ◁

3.4.3 Termination of PostFix Programs

An important property of PostFix is expressed by the following theorem:

> PostFix **Termination Theorem:** All PostFix programs are guaranteed to terminate. That is, executing a PostFix program always either returns a numeral or signals an error.[10]

This theorem is based on the following intuition: existing commands are consumed by execution, but no new commands are ever created, so the commands must eventually "run out." This intuition is essentially correct, but an intuition does not a proof make. After all, PostFix is complex enough to harbor a subtlety that invalidates the intuition. The `nget` command allows the duplication of numerals — is this problematic with regards to termination? Executable sequences are moved to the stack, but their contents can later be prepended to the code component. How can we be certain that this shuffling between code and stack doesn't go on forever? And how do we deal with the fact that executable sequences can be arbitrarily nested?

These questions indicate the need for a more convincing argument that termination is guaranteed. This is the kind of situation in which formal semantics comes in handy. Below we present a proof for termination based on the SOS for PostFix.

3.4.3.1 Energy

Associate with each PostFix configuration a natural number called its *energy* (so called to suggest the potential energy of a dynamical system). By considering each rewrite rule of the semantics in turn, we will prove that the energy strictly decreases with each transition. The energy of an initial configuration must then be an upper bound on the length of any path of transitions leading from the initial configuration. Since the initial energy is finite, there can be no unbounded transition sequences from the initial configuration, so the execution of a program must terminate.

[10] This theorem can fail to hold if PostFix is extended with new commands, such as a `dup` command that duplicates the top stack value. See Section 3.5 for details.

The energy of a configuration is defined by the following energy functions:

$$\begin{aligned}
\mathrm{E}_{config}[\![\langle Q,\ S\rangle]\!] &= \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![S]\!] && (3.1)\\
\mathrm{E}_{seq}[\![[\,]]\!] &= 0 && (3.2)\\
\mathrm{E}_{seq}[\![C\ .\ Q]\!] &= 1 + \mathrm{E}_{com}[\![C]\!] + \mathrm{E}_{seq}[\![Q]\!] && (3.3)\\
\mathrm{E}_{stack}[\![[\,]]\!] &= 0 && (3.4)\\
\mathrm{E}_{stack}[\![V\ .\ S]\!] &= \mathrm{E}_{com}[\![V]\!] + \mathrm{E}_{stack}[\![S]\!] && (3.5)\\
\mathrm{E}_{com}[\![(Q)]\!] &= \mathrm{E}_{seq}[\![Q]\!] && (3.6)\\
\mathrm{E}_{com}[\![C]\!] &= 1, \quad C \text{ not an executable sequence.} && (3.7)
\end{aligned}$$

These definitions embody the following intuitions:

- The energy of a configuration, sequence, or stack is greater than or equal to the sum of the energy of its components.
- Executing a command consumes at least one unit of energy (the 1 that appears in 3.3). This is true even for commands that are transferred from the code component to the stack component (i.e., numerals and executable sequences); such commands are worth one more unit of energy in the command sequence than on the stack.[11]
- An executable sequence can be worth no more energy as a sequence than as a stack value (3.6).

The following lemmas are handy for reasoning about the energy of sequences:

$$\begin{aligned}
\mathrm{E}_{com}[\![C]\!] &\geq 0 && (3.8)\\
\mathrm{E}_{seq}[\![Q_1\ @\ Q_2]\!] &= \mathrm{E}_{seq}[\![Q_1]\!] + \mathrm{E}_{seq}[\![Q_2]\!] && (3.9)
\end{aligned}$$

These can be derived from the energy definitions above. Their derivations are left as an exercise.

Equipped with the energy definitions and identity 3.9, we are ready to prove the PostFix Termination Theorem.

3.4.3.2 The Proof of Termination

Proof: We show that every transition reduces the energy of a configuration. Recall that every transition in an SOS has a proof in terms of the rewrite rules. In the case of PostFix, where all the rules are axioms, the proof is trivial: every

[11] The invocation $\mathrm{E}_{com}[\![V]\!]$ that appears in 3.5 may seem questionable because $\mathrm{E}_{com}[\![]\!]$ should be called on elements of Command, not elements of Value. But since every stack value is also a command, the invocation is well-defined.

PostFix transition is justified by one rewrite axiom. To prove a property about PostFix transitions, we just need to show that it holds for each rewrite axiom in the SOS. Here's the case analysis for the energy reduction property:

- [*num*]: $\langle N\ .\ Q,\ S\rangle \Rightarrow \langle Q,\ N\ .\ S\rangle$

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle N\ .\ Q,\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![N\ .\ Q]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![N]\!] + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3} \\ = & 1 + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![N\ .\ S]\!] & \text{by 3.5} \\ = & 1 + \mathrm{E}_{config}[\![\langle Q,\ N\ .\ S\rangle]\!] & \text{by 3.1} \end{array}$$

  The LHS has one more unit of energy than the RHS, so moving a numeral to the stack reduces the configuration energy by one unit.

- [*seq*]: $\langle Q_{exec}\ .\ Q_{rest},\ S\rangle \Rightarrow \langle Q_{rest},\ Q_{exec}\ .\ S\rangle$ Moving an executable sequence to the stack also consumes one energy unit by exactly the same argument as for [*num*].

- [*pop*]: $\langle \texttt{pop}\ .\ Q,\ V_{top}\ .\ S\rangle \Rightarrow \langle Q,\ S\rangle$ Popping V_{top} off of a stack takes at least two energy units:

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle \texttt{pop}\ .\ Q,\ V_{top}\ .\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![\texttt{pop}\ .\ Q]\!] + \mathrm{E}_{stack}[\![V_{top}\ .\ S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![\texttt{pop}]\!] + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{com}[\![V_{top}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3 and 3.5} \\ = & 2 + \mathrm{E}_{com}[\![V_{top}]\!] + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.7} \\ \geq & 2 + \mathrm{E}_{config}[\![\langle Q,\ S\rangle]\!] & \text{by 3.1 and 3.8} \end{array}$$

- [*swap*]: $\langle \texttt{swap}\ .\ Q,\ V_1\ .\ V_2\ .\ S\rangle \Rightarrow \langle Q,\ V_2\ .\ V_1\ .\ S\rangle$ Swapping the top two elements of a stack consumes two energy units:

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle \texttt{swap}\ .\ Q,\ V_1\ .\ V_2\ .\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![\texttt{swap}\ .\ Q]\!] + \mathrm{E}_{stack}[\![V_1\ .\ V_2\ .\ S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![\texttt{swap}]\!] + \mathrm{E}_{seq}[\![Q]\!] & \\ & \quad + \ \mathrm{E}_{com}[\![V_1]\!] + \mathrm{E}_{com}[\![V_2]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3 and 3.5} \\ = & 2 + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![V_2\ .\ V_1\ .\ S]\!] & \text{by 3.7 and 3.5} \\ = & 2 + \mathrm{E}_{config}[\![\langle Q,\ V_2\ .\ V_1\ .\ S\rangle]\!] & \text{by 3.1} \end{array}$$

- [*execute*]: $\langle \texttt{exec}\ .\ Q_{rest},\ Q_{exec}\ .\ S\rangle \Rightarrow \langle Q_{exec}\ @\ Q_{rest},\ S\rangle$ Executing the `exec` command consumes two energy units:

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle \texttt{exec}\ .\ Q_{rest},\ Q_{exec}\ .\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![\texttt{exec}\ .\ Q_{rest}]\!] + \mathrm{E}_{stack}[\![Q_{exec}\ .\ S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![\texttt{exec}]\!] + \mathrm{E}_{seq}[\![Q_{rest}]\!] & \\ & \quad + \ \mathrm{E}_{com}[\![(Q_{exec})]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3 and 3.5} \\ = & 2 + \mathrm{E}_{seq}[\![Q_{exec}]\!] + \mathrm{E}_{seq}[\![Q_{rest}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.6 and 3.7} \\ = & 2 + \mathrm{E}_{seq}[\![Q_{exec}\ @\ Q_{rest}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.9} \\ = & 2 + \mathrm{E}_{config}[\![\langle Q_{exec}\ @\ Q_{rest},\ S\rangle]\!] & \text{by 3.1} \end{array}$$

- [*nget*], [*arithop*], [*relop-true*], [*relop-false*], [*sel-true*], [*sel-false*]: These cases are similar to those above and are left as exercises for the reader. ◇

The approach of defining a natural number function that decreases on every iteration of a process is a common technique for proving termination. However, inventing the function can sometimes be tricky. In the case of PostFix, we have to get the relative weights of components just right to handle movements between the program and stack.

The termination proof presented above is rather complex. The difficulty is not inherent to PostFix, but is due to the particular way we have chosen to formulate its semantics. There are alternative formulations in which the termination proof is simpler (see exercise 3.25).

▷ **Exercise 3.21** Show that lemmas 3.8 and 3.9 hold. ◁

▷ **Exercise 3.22** Complete the proof of the PostFix termination theorem by showing that the following axioms reduce configuration energy: [*nget*], [*arithop*], [*relop-true*], [*relop-false*], [*sel-true*], [*sel-false*]. ◁

▷ **Exercise 3.23** Bud "eagle-eye" Lojack notices that definitions 3.2 and 3.4 do not appear as the justification for any steps in the PostFix Termination Theorem. He reasons that these definitions are arbitrary, so he could just as well use the following definitions instead:

$$\begin{array}{lcll} \mathrm{E}_{seq}[\![[]]\!] & = & 17 & (\ 3.2'\ ) \\ \mathrm{E}_{stack}[\![[]]\!] & = & 23 & (\ 3.4'\ ) \end{array}$$

Is Bud correct? Explain your answer. ◁

▷ **Exercise 3.24** Prove the termination property of PostFix based on the SOS for PostFix2 from Exercise 3.7.

a. Define an appropriate energy function on configurations in the alternative SOS.

b. Show that each transition in the alternative SOS reduces energy. ◁

3.4.3.3 Structural Induction

The above proof is based on a PostFix SOS that uses only axioms. But what if the SOS contained progress rules, like [*exec-done*] from Section 3.2.3.3? How do we prove a property like reduction in configuration energy when progress rules are involved?

Here's where we can take advantage of the fact that every transition of an SOS must be justified by a finite proof tree based on the rewrite rules. Recall that there are two types of nodes in the proof tree: the leaves, which correspond to axioms, and the intermediate nodes, which correspond to progress rules. Suppose we can show that

- the property holds at each leaf — i.e., it is true for the consequent of every axiom; and
- the property holds at each intermediate node — i.e., for every progress rule, if the property holds for all of the antecedents, then it also holds for the consequent.

Then, by induction on the height of its proof tree, the property must hold for each transition specified by the rewrite rules. This method for proving a property based on the structure of a tree (in this case the proof tree of a transition relation) is called **structural induction**.

As an example of a proof by structural induction, we consider how the previous proof of the termination property for PostFix would be modified for an SOS that uses the [*exec-done*] and [*exec-prog*] rules in place of the [*exec*] rule. It is straightforward to show that the [*exec-done*] axiom reduces configuration energy; this is left as an exercise for the reader. To show that the [*exec-prog*] rule satisfies the property, we must show that *if* its single antecedent transition reduces configuration energy, *then* its consequent transition reduces configuration energy as well.

Recall that the [*exec-prog*] rule has the form:

$$\frac{\langle Q_{exec},\ S\rangle \Rightarrow \langle Q_{exec}{}',\ S'\rangle}{\langle \texttt{exec . } Q_{rest},\ Q_{exec}\ \texttt{ . } S\rangle \Rightarrow \langle \texttt{exec . } Q_{rest},\ Q_{exec}{}'\ \texttt{ . } S'\rangle} \qquad [\textit{exec-prog}]$$

We assume that the antecedent transition,

$$\langle Q_{exec},\ S\rangle \Rightarrow \langle Q_{exec}{}',\ S'\rangle,$$

reduces configuration energy, so that the following inequality holds:

$$\mathrm{E}_{\mathit{config}}[\![\langle Q_{exec},\ S\rangle]\!] > \mathrm{E}_{\mathit{config}}[\![\langle Q_{exec}{}',\ S'\rangle]\!].$$

Then we show that the consequent transition also reduces configuration energy:

$$
\begin{array}{lll}
& \mathrm{E}_{config}[\![\langle \texttt{exec . } Q_{rest},\ Q_{exec}\ \texttt{ . }\ S\rangle]\!] & \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{stack}[\![Q_{exec}\ \texttt{ . }\ S]\!] & \text{by 3.1} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{com}[\![(\,Q_{exec})]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.5} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{seq}[\![Q_{exec}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.6} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{config}[\![\langle Q_{exec},\ S\rangle]\!] & \text{by 3.1} \\
> & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{config}[\![\langle Q_{exec}{}',\ S'\rangle]\!] & \text{by assumption} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{seq}[\![Q_{exec}{}']\!] + \mathrm{E}_{stack}[\![S']\!] & \text{by 3.1} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{com}[\![(\,Q_{exec}{}')]\!] + \mathrm{E}_{stack}[\![S']\!] & \text{by 3.6} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{stack}[\![Q_{exec}{}'\ \texttt{ . }\ S']\!] & \text{by 3.5} \\
= & \mathrm{E}_{config}[\![\langle \texttt{exec . } Q_{rest},\ Q_{exec}{}'\ \texttt{ . }\ S'\rangle]\!] & \text{by 3.1}
\end{array}
$$

The $>$ appearing in the derivation sequence guarantees that the energy specified by the first line is strictly greater than the energy specified by the last line. This completes the proof that the [*exec-prog*] rule reduces configuration energy. Together with the proofs that the axioms reduce configuration energy, this provides an alternative proof of PostFix's termination property.

▷ **Exercise 3.25** Prove the termination property of PostFix based on the alternative PostFix SOS suggested in Exercise 3.12:

a. Define an appropriate energy function on configurations in the alternative SOS.

b. Show that each transition in the alternative SOS reduces energy.

c. The termination proof for the alternative semantics should be more straightforward than the termination proofs in the text and in Exercise 3.24. What characteristic(s) of the alternative SOS simplify the proof? Does this mean the alternative SOS is a "better" one? ◁

▷ **Exercise 3.26** Prove that the rewrite rules [*exec-prog*] and [*exec-done*] presented in the text specify the same behavior as the [*execute*] rule. That is, show that for any configuration cf of the form $\langle \texttt{exec . } Q,\ S\rangle$, both sets of rules eventually rewrite cf into either (1) a stuck state or (2) the same configuration. ◁

▷ **Exercise 3.27** As in PostFix, every program in the EL language terminates. Prove this fact based on an operational semantics for EL (see Exercise 3.10). ◁

3.4.4 Safe PostFix Transformations

3.4.4.1 Observational Equivalence

One of the most important aspects of reasoning about programs is knowing when it is safe to replace one program phrase by another. Two phrases are said to be

observationally equivalent (or **behaviorally equivalent**) if an instance of one can be replaced by the other in any program without changing the behavior of the program.

Observational equivalence is important because it is the basis for a wide range of program transformation techniques. It is often possible to improve a pragmatic aspect of a program by replacing a phrase by one that is equivalent but more efficient. For example, we expect that the PostFix sequence [`1`, `add`, `2`, `add`] can always be replaced by [`3`, `add`] without changing the meaning of the surrounding program. The latter may be more desirable in practice because it performs fewer additions.

A series of simple transformations can sometimes lead to dramatic performance improvements. Consider the following three transformations on PostFix command sequences, which are just three of the many safe PostFix transformations:

Before	After	Name
$[V_1, V_2, \texttt{swap}]$	$[V_2, V_1]$	[*swap-trans*]
$[(Q), \texttt{exec}]$	Q	[*exec-trans*]
$[N_1, N_2, A]$	$[N_{result}]$ where $N_{result} = (calculate\ A\ N_1\ N_2)$	[*arith-trans*]

Applying these to our running example of a PostFix command sequence yields the following sequence of simplifications:

```
((2 (3 mul add) exec) 1 swap exec sub)
↪ ((2 3 mul add) 1 swap exec sub)        [exec-trans]
↪ ((6 add) 1 swap exec sub)              [arith-trans]
↪ (1 (6 add) exec sub)                   [swap-trans]
↪ (1 6 add sub)                          [exec-trans]
↪ (7 sub)                                [arith-trans]
```

Thus, the original command sequence is a "subtract 7" subroutine. The transformations essentially perform at compile time operations that otherwise would be performed at run time.

It is often tricky to determine whether two phrases are observationally equivalent. For example, at first glance it might seem that the PostFix sequence [`swap`, `swap`] can always be replaced by the empty sequence [ ]. While this transformation is valid in many situations, these two sequences are not observationally equivalent because they behave differently when the stack contains fewer than two elements. For instance, the PostFix program `(postfix 0 1)` returns `1` as a final answer, but the program `(postfix 0 1 swap swap)` generates an error. Two phrases are observationally equivalent only if they are interchangeable in *all* programs.

Observational equivalence can be formalized in terms of the notions of **behavior** and **context** presented earlier. Recall that the behavior of a program

(see Section 3.2.1) is specified by a function *beh* that maps a program and its inputs to a set of possible outcomes:

$$beh : \text{Program} \times \text{Inputs} \rightarrow \mathcal{P}(\text{Outcome})$$

The behavior is deterministic when the resulting set is guaranteed to be a singleton. A program context is a program with a hole in it (see Section 3.2.3.4).

> **Observational Equivalence:** Suppose that $\mathbb{P}$ ranges over program contexts and H ranges over the kinds of phrases that fill the holes in program contexts. Then H_1 and H_2 are defined to be observationally equivalent (written $H_1 =_{obs} H_2$) if and only if for all program contexts $\mathbb{P}$ and all inputs I, $beh\ \langle \mathbb{P}\{H_1\}, I\rangle = beh\ \langle \mathbb{P}\{H_2\}, I\rangle$.

We will consider PostFix as an example. An appropriate notion of program contexts for PostFix is defined in Figure 3.19. A command sequence context $\mathbb{Q}$ is one that can be filled with a sequence of commands to yield another sequence of commands. For example, if $\mathbb{Q} = [\texttt{(2 mul)}, \texttt{3}]\ @\ \Box\ @\ [\texttt{exec}]$, then $\mathbb{Q}\{[\texttt{4}, \texttt{add}, \texttt{swap}]\} = [\texttt{(2 mul)}, \texttt{3}, \texttt{4}, \texttt{add}, \texttt{swap}, \texttt{exec}]$. The [*Prefix*] and [*Suffix*] productions allow the hole to be surrounded by arbitrary command sequences, while the [*Nesting*] production allows the hole to be nested within an executable sequence command. (The notation $[(\mathbb{Q})]$ designates a sequence containing a single element. That element is an executable sequence that contains a single hole.) Due to the presence of @, the grammar for PostfixSequenceContext is ambiguous, but that will not affect our presentation, since filling the hole for any parsing of a sequence context yields exactly the same sequence.

$\mathbb{P} \in$ PostfixProgContext
$\mathbb{Q} \in$ PostfixSequenceContext

$\mathbb{P}$ ::=	`(postfix` $N_{numargs}$ $\mathbb{Q}$`)`	[Program Context]
$\mathbb{Q}$ ::=	$\Box$	[Hole]
	$\mid Q\ @\ \mathbb{Q}$	[Prefix]
	$\mid \mathbb{Q}\ @\ Q$	[Suffix]
	$\mid [(\mathbb{Q})]$	[Nesting]

Figure 3.19: Definition of PostFix contexts.

The possible outcomes of a program must be carefully defined to lead to a satisfactory notion of observational equivalence. The outcomes for PostFix defined in Section 3.2.1 are fine, but small changes can sometimes lead to surprising results. For example, suppose we allow PostFix programs to return the

top value of a non-empty stack, even if the top value is an executable sequence. If we can observe the structure of a returned executable sequence, then this change invalidates all non-trivial program transformations! To see why, take any two sequences we expect to be equivalent (say, [1, add, 2, add] and [3, add]) and plug them into the context (postfix 0 (□)). In the modified semantics, the two outcomes are the executable sequences (1 add 2 add) and (3 add), which are clearly not the same, and so the two sequences are not observationally equivalent.

The problem is that the modified SOS makes distinctions between executable sequence outcomes that are too fine-grained for our purposes. We can fix the problem by instead adopting a coarser-grained notion of behavior in which there is no observable difference between outcomes that are executable sequences. For example, the outcome in this case could be the token executable, indicating that the outcome *is* an executable sequence without divulging *which* particular executable sequence it is. With this change, all the expected program transformations become valid again.

3.4.4.2 Transform Equivalence

It is possible to show the observational equivalence of two particular PostFix command sequences according to the definition on page 84. However, we will follow another route. First, we will develop an easier-to-prove notion of equivalence for PostFix sequences called **transform equivalence**. Then, after giving an example of transform equivalence, we will prove a theorem that transform equivalence implies observational equivalence for PostFix programs. This approach has the advantage that the structural induction proof on contexts needed to show observational equivalence need only be proved once (for the theorem) rather than for every pair of PostFix command sequences.

Transform equivalence is based on the intuition that PostFix command sequences can be viewed as a means of transforming one stack to another. Informally, transform equivalence is defined as follows:

> **Transform Equivalence:** Two PostFix command sequences are **transform equivalent** if they always transform equivalent input stacks to equivalent output stacks.

This definition is informal in that it doesn't say how command sequences can be viewed as transformers or pin down what it means for two stacks to be equivalent. We will now flesh these notions out.

In order to view PostFix command sequences as stack transformers, we will extend the PostFix SOS as follows:

- Modify Stack to contain a distinguished element S_{error}:

  $S \in \text{Stack} = \text{Value}^* + \text{ErrorStack}$
  $\text{ErrorStack} = \{S_{error}\}$

- Extend the transition relation, $\Rightarrow$, so that for all stuck states $cf_{stuck} \in \mathit{Stuck}$, $cf_{stuck} \Rightarrow \langle [\,],\ S_{error} \rangle$. This says that any configuration formerly considered stuck now rewrites to a final configuration with an error stack.

- Define ($\mathit{finalStack}\ Q\ S$) to be S' if $\langle Q,\ S \rangle \stackrel{*}{\Rightarrow} \langle [\,],\ S' \rangle$. The *finalStack* function is well-defined because POSTFIX is deterministic; with the extensions for handling S_{error}, *finalStack* is also a total function.

As examples of *finalStack*, consider ($\mathit{finalStack}$ [`add`, `mul`] [4, 3, 2, 1]) = [24, 1] and ($\mathit{finalStack}$ [`add`, `exec`] [4, 3, 2, 1]) = S_{error}.

The simplest notion of "stack equivalence" is that two stacks are equivalent if they are identical sequences of values. But this notion has problems similar to those discussed above with regard to outcomes in the context of observational equivalence. For example, suppose we are able to show that `(1 add 2 add)` and `(3 add)` are transform equivalent. Then we'd also like the transform equivalence of `((1 add 2 add))` and `((3 add))` to follow as a corollary. But given identical input stacks, these two sequences do *not* yield identical output stacks — the top values of the output stacks are different executable sequences!

To finesse this problem, we need a notion of stack equivalence that treats two executable sequence elements as the same if they are transform equivalent. The recursive nature of these notions prompts us to define *three* mutually recursive equivalence relations that formalize this approach: one between command sequences (transform equivalence), one between stacks (stack equivalence), and one between stack elements (value equivalence).

- Command sequences Q_1 and Q_2 are **transform equivalent** (written $Q_1 \sim_Q Q_2$) if, for all pairs of stack equivalent stacks S_1 and S_2, ($\mathit{finalStack}\ Q_1\ S_1$) is stack equivalent to ($\mathit{finalStack}\ Q_2\ S_2$). The case $S_1 = S_{error} = S_2$ can safely be ignored because S_{error} models only final configurations, not intermediate ones.

- Stacks S_1 and S_2 are **stack equivalent** (written $S_1 \sim_S S_2$) if

  - both S_1 and S_2 are the distinguished error stack, S_{error}; or
  - S_1 and S_2 are equal-length sequences of values that are elementwise value equivalent. I.e., $S_1 = [V_1, \ldots, V_n]$, $S_2 = [V_1{}', \ldots, V_n{}']$, and $V_i \sim_V V_i{}'$ for all i such that $1 \leq i \leq n$.

- Stack elements V_1 and V_2 are **value equivalent** (written $V_1 \sim_V V_2$) if V_1 and V_2 are the same integer numeral (i.e., $V_1 = N = V_2$) or if V_1 and V_2 are executable sequences whose contents are transform equivalent (i.e., $V_1 = (Q_1)$, $V_2 = (Q_2)$, and $Q_1 \sim_Q Q_2$).

Despite the mutually recursive nature of these definitions, we claim that all three are well-defined equivalence relations as long as we choose the largest relations satisfying the descriptions.

Two PostFix command sequences can be proved transform equivalent by case analysis on the structure of input stacks. This is much easier than the case analysis on the structure of contexts that is implied by observational equivalence. Since (as we shall show below) observational equivalence follows from transform equivalence, transform equivalence is a practical technique for demonstrating observational equivalence.

As a simple example of transform equivalence, we show that $[1, \texttt{add}, 2, \texttt{add}] \sim_Q [3, \texttt{add}]$. Consider two non-error stacks S_1 and S_2 such that $S_1 \sim_S S_2$. We proceed by case analysis on the structure of the stacks:

- S_1 and S_2 are both $[\,]$, in which case

$$\begin{array}{ll} & (\mathit{finalStack}\ [3, \texttt{add}]\ [\,]) \\ = & (\mathit{finalStack}\ [\texttt{add}]\ [3]) \\ = & S_{error} \\ = & (\mathit{finalStack}\ [\texttt{add}, 2, \texttt{add}]\ [1]) \\ = & (\mathit{finalStack}\ [1, \texttt{add}, 2, \texttt{add}]\ [\,]) \end{array}$$

- S_1 and S_2 are non-empty sequences whose heads are the same numeric literal and whose tails are stack equivalent. I.e., $S_1 = N\ .\ S_1'$, $S_2 = N\ .\ S_2'$, and $S_1' \sim_S S_2'$.

$$\begin{array}{ll} & (\mathit{finalStack}\ [3, \texttt{add}]\ N\ .\ S_1') \\ = & (\mathit{finalStack}\ [\texttt{add}]\ \texttt{3}\ .\ N\ .\ S_1') \\ = & (\mathit{finalStack}\ [\,]\ N\texttt{+3}\ .\ S_1') \\ = & (\mathit{finalStack}\ [N\texttt{+3}]\ S_1') \\ \sim_S & (\mathit{finalStack}\ [N\texttt{+3}]\ S_2') \\ = & (\mathit{finalStack}\ [\,]\ N\texttt{+3}\ .\ S_2') \\ = & (\mathit{finalStack}\ [\texttt{add}]\ \texttt{2}\ .\ N\texttt{+1}\ .\ S_2') \\ = & (\mathit{finalStack}\ [2, \texttt{add}]\ N\texttt{+1}\ .\ S_2') \\ = & (\mathit{finalStack}\ [\texttt{add}, 2, \texttt{add}]\ \texttt{1}\ .\ N\ .\ S_2') \\ = & (\mathit{finalStack}\ [1, \texttt{add}, 2, \texttt{add}]\ N\ .\ S_2') \end{array}$$

- S_1 and S_2 are non-empty sequences whose heads are transform equivalent executable sequences and whose tails are stack equivalent. I.e., $S_1 = Q_1\ .\ S_1'$, $S_2 = Q_2\ .\ S_2'$, $Q_1 \sim_Q Q_2$, and $S_1' \sim_S S_2'$.

$$
\begin{array}{l}
(\mathit{finalStack}\ [\texttt{3},\texttt{add}]\ \ Q_1\ .\ S_1\,')\\
\quad =\ \ (\mathit{finalStack}\ [\texttt{add}]\ \ \texttt{3}\ .\ Q_1\ .\ S_1\,')\\
\quad =\ \ S_{error}\\
\quad =\ \ (\mathit{finalStack}\ [\texttt{add},\texttt{2},\texttt{add}]\ \ \texttt{1}\ .\ Q_2\ .\ S_2\,')\\
\quad =\ \ (\mathit{finalStack}\ [\texttt{1},\texttt{add},\texttt{2},\texttt{add}]\ \ Q_2\ .\ S_2\,')
\end{array}
$$

In all three cases,

$$(\mathit{finalStack}\ [\texttt{1},\texttt{add},\texttt{2},\texttt{add}]\ \ S_1)\ \sim_S\ (\mathit{finalStack}\ [\texttt{3},\texttt{add}]\ \ S_2)\,,$$

so the transform equivalence of the sequences follows by definition of $\sim_Q$.

We emphasize that stacks can be equivalent without being identical. For instance, given the result of the above example, it is easy to construct two stacks that are stack equivalent without being identical:

$$[\texttt{(1 add 2 add)},5]\ \sim_S\ [\texttt{(3 add)},5].$$

Intuitively, these stacks are equivalent because they cannot be distinguished by any PostFix command sequence. Any such sequence must either ignore both sequence elements (e.g., [pop]), attempt an illegal operation on both sequence elements (e.g., [mul]), or execute both sequence elements on equivalent stacks (via exec). But because the sequence elements are transform equivalent, executing them cannot distinguish them.

3.4.4.3 Transform Equivalence Implies Observational Equivalence

We wrap up the discussion of observational equivalence by showing that transform equivalence of PostFix command sequences implies observational equivalence. This can be explained informally as follows. Every PostFix program context consists of two parts: the commands performed before the hole and the commands performed after the hole. The commands before the hole transform the initial empty stack into S_{pre}. Suppose the hole is filled by one of two executable sequences, Q_1 and Q_2, that are transform equivalent. Then the stacks S_{post1} and S_{post2} that result from executing these sequences, respectively, on S_{pre} must be stack equivalent. The commands performed after the hole must transform S_{post1} and S_{post2} into stack equivalent stacks S_{final1} and S_{final2}. Since behavior depends only on the equivalence class of the final stack, it is impossible to construct a context that distinguishes Q_1 and Q_2. Therefore, they are observationally equivalent.

Below, we present a formal proof that transform equivalence implies observational equivalence.

> PostFix **Transform Equivalence Theorem:** $Q_1 \sim_Q Q_2$ implies $Q_1 =_{obs} Q_2$.

This theorem is useful because it is generally easier to show that two command sequences are transform equivalent than to construct a proof based directly on the definition of observational equivalence.

Proof: We will show that for all sequence contexts $\mathbb{Q}$, $Q_1 \sim_Q Q_2$ implies $\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$. The latter equivalence implies that, for all sequence contexts $\mathbb{Q}$ and initial stacks S_{init},

$$(\mathit{finalStack}\ \mathbb{Q}\{Q_1\}\ S_{init}) \sim_S (\mathit{finalStack}\ \mathbb{Q}\{Q_2\}\ S_{init})\,.$$

This in turn implies that for all numerals N_n and arguments sequences $N_{args}{}^*$,

$$\mathit{beh}\ \langle(\texttt{program}\ N_n\ \mathbb{Q}\{Q_1\}), N_{args}{}^*\rangle = \mathit{beh}\ \langle(\texttt{program}\ N_n\ \mathbb{Q}\{Q_2\}), N_{args}{}^*\rangle.$$

So $Q_1 =_{obs} Q_2$ by the definition of observational equivalence.

We will employ the following properties of transform equivalence, which are left as exercises for the reader:

$$Q_1 \sim_Q Q_1{}' \text{ and } Q_2 \sim_Q Q_2{}' \quad \text{implies} \quad Q_1\ @\ Q_2 \sim_Q Q_1{}'\ @\ Q_2{}' \tag{3.10}$$

$$Q_1 \sim_Q Q_2 \quad \text{implies} \quad [(Q_1)] \sim_Q [(Q_2)] \tag{3.11}$$

Property 3.11 is tricky to read; it says that if Q_1 and Q_2 are transform equivalent, then the sequences that result from nesting Q_1 and Q_2 in executable sequences within a singleton sequence are also transform equivalent.

We proceed by structural induction on the grammar of the PostfixSequenceContext domain:

- (Base case) For sequence contexts of the form $\Box$, $Q_1 \sim_Q Q_2$ trivially implies $\Box\{Q_1\} \sim_Q \Box\{Q_2\}$.
- (Induction cases) For each of the following compound sequence contexts, assume that $Q_1 \sim_Q Q_2$ implies $\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$ for any $\mathbb{Q}$.
  - For sequence contexts of the form $Q\ @\ \mathbb{Q}$,

    $Q_1 \sim_Q Q_2$

implies	$\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$	by assumption
implies	$Q\ @\ (\mathbb{Q}\{Q_1\}) \sim_Q Q\ @\ (\mathbb{Q}\{Q_2\})$	by reflexivity of $\sim_Q$ and 3.10
implies	$(Q\ @\ \mathbb{Q})\{Q_1\} \sim_Q (Q\ @\ \mathbb{Q})\{Q_2\}$	by definition of $\mathbb{Q}$

  - Sequence contexts of the form $\mathbb{Q}\ @\ Q$ are handled similarly to those of the form $Q\ @\ \mathbb{Q}$.
  - For sequence contexts of the form $[(\mathbb{Q})]$,

    $Q_1 \sim_Q Q_2$

implies	$\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$	by assumption
implies	$[(\mathbb{Q}\{Q_1\})] \sim_Q [(\mathbb{Q}\{Q_2\})]$	by 3.11
implies	$[(\mathbb{Q})]\{Q_1\} \sim_Q [(\mathbb{Q})]\{Q_2\}$	by definition of $\mathbb{Q}$

    $\diamond$

▷ **Exercise 3.28** For each of the following purported observational equivalences, either prove that the observational equivalence is valid (via transform equivalence), or give a counterexample to show that it is not.

a. $[N, \texttt{pop}] =_{obs} [\,]$

b. $[\texttt{add}, N, \texttt{add}] =_{obs} [N, \texttt{add}, \texttt{add}]$

c. $[N_1, N_2, A] =_{obs} [N_{result}]$, where $N_{result} =$(*calculate* A N_2 N_1)

d. $[\texttt{(}Q\texttt{)}, \texttt{exec}] =_{obs} Q$

e. $[\texttt{(}Q\texttt{)}, \texttt{(}Q\texttt{)}, \texttt{sel}, \texttt{exec}] =_{obs} \texttt{pop . } Q$

f. $[N_1, \texttt{(}N_2\ \texttt{(}Q_a\texttt{)}\ \texttt{(}Q_b\texttt{)}\ \texttt{sel exec)}, \texttt{(}N_2\ \texttt{(}Q_c\texttt{)}\ \texttt{(}Q_d\texttt{)}\ \texttt{sel exec)}, \texttt{sel}, \texttt{exec}]$
$=_{obs} [N_2, \texttt{(}N_1\ \texttt{(}Q_a\texttt{)}\ \texttt{(}Q_c\texttt{)}\ \texttt{sel exec)}, \texttt{(}N_1\ \texttt{(}Q_b\texttt{)}\ \texttt{(}Q_d\texttt{)}\ \texttt{sel exec)}, \texttt{sel}, \texttt{exec}]$

g. $[C_1, C_2, \texttt{swap}] =_{obs} [C_2, C_1]$

h. $[\texttt{swap}, \texttt{swap}, \texttt{swap}] =_{obs} [\texttt{swap}]$ ◁

▷ **Exercise 3.29** Prove lemmas 3.10 and 3.11, which are used to show that transform equivalence implies operational equivalence. ◁

▷ **Exercise 3.30**

a. Modify the PostFix semantics in Figure 3.3 so that the outcome of a PostFix program whose final configuration has an executable sequence at the top is the token `executable`.

b. In your modified semantics, show that transform equivalence still implies observational equivalence. ◁

▷ **Exercise 3.31** Prove the following composition theorem for observationally equivalent PostFix sequences:

$Q_1 =_{obs} Q_1\,'$ and $Q_2 =_{obs} Q_2\,'$ implies Q_1 @ $Q_2 =_{obs} Q_1\,'$ @ $Q_2\,'$ ◁

▷ **Exercise 3.32** Which of the following transformations on EL numerical expressions are safe? Explain your answers. Be sure to consider stuck expressions like `(/ 1 0)`.

a. `(+ 1 2)` $\hookrightarrow$ `3`

b. `(+ 0` *NE*`)` $\hookrightarrow$ *NE*

c. `(* 0` *NE*`)` $\hookrightarrow$ `0`

d. `(+ 1 (+ 2` *NE*`))` $\hookrightarrow$ `(+ 3` *NE*`)`

e. `(+` NE NE`)` $\hookrightarrow$ `(* 2` NE`)`

f. `(if (=` N N`)` NE_1 NE_2`)` $\hookrightarrow$ NE_1

g. `(if (=` NE_1 NE_1`)` NE_2 NE_3`)` $\hookrightarrow$ NE_2

h. `(if` BE NE NE`)` $\hookrightarrow$ NE ◁

▷ **Exercise 3.33†** Develop a notion of transform equivalence for EL that is powerful enough to formally prove that the transformations in Exercise 3.32 that you think are safe are really safe. You will need to design appropriate contexts for EL programs, numerical expressions, and boolean expressions. ◁

▷ **Exercise 3.34‡** Given that transform equivalence implies observational equivalence, it is natural to wonder whether the converse is true. That is, does the following implication hold?

$$Q_1 =_{obs} Q_2 \text{ implies } Q_1 \sim_Q Q_2$$

If so, prove it; if not, explain why. ◁

▷ **Exercise 3.35†** Consider the following $\mathcal{T}_\mathcal{P}$ function, which translates an ELMM program to a PostFix program:

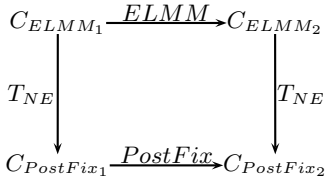
$\mathcal{T}_\mathcal{P}$: $\text{Program}_{ELMM} \rightarrow \text{Program}_{PostFix}$
$\mathcal{T}_\mathcal{P}$ $[\![$`(elmm` NE_{body}`)`$]\!]$ = `(postfix 0` $\mathcal{T}_{\mathcal{NE}}[\![NE_{body}]\!]$ `)`

$\mathcal{T}_{\mathcal{NE}}$: NumExp $\rightarrow$ Commands
$\mathcal{T}_{\mathcal{NE}}[\![N]\!]$ = $[N]$
$\mathcal{T}_{\mathcal{NE}}[\![$`(`$A$ NE_1 NE_2`)`$]\!]$ = $\mathcal{T}_{\mathcal{NE}}[\![NE_1]\!]$ @ $\mathcal{T}_{\mathcal{NE}}[\![NE_2]\!]$ @ $[\mathcal{T}_\mathcal{A}[\![A]\!]$ $]$

$\mathcal{T}_\mathcal{A}$: $\text{ArithmeticOperator}_{ELMM} \rightarrow \text{ArithmeticOperator}_{PostFix}$
$\mathcal{T}_\mathcal{A}[\![$`+`$]\!]$ = `add`
$\mathcal{T}_\mathcal{A}[\![$`-`$]\!]$ = `sub`, *etc.*

a. What is $\mathcal{T}_\mathcal{P}$ $[\![$`(elmm (/ (+ 25 75) (* (- 7 4) (+ 5 6))))`$]\!]$?

b. Intuitively, $\mathcal{T}_\mathcal{P}$ maps an ELMM program to a PostFix program with the same behavior. Develop a proof that formalizes this intuition. As part of your proof, show that the following diagram commutes:



The nodes C_{ELMM_1} and C_{ELMM_2} represent ELMM configurations, and the nodes $C_{PostFix_1}$ and $C_{PostFix_2}$ represent PostFix configurations of the form introduced

in Exercise 3.12. The horizontal arrows are transitions in the respective systems, while the vertical arrows are applications of $\mathcal{T}_{\mathcal{NE}}$. It may help to think in terms of a context-based semantics.

c. Extend the translator to translate (1) ELM programs and (2) EL programs. In each case, prove that the program resulting from your translation has the same behavior as the original program. ◁

3.5 Extending PostFix

We close this chapter on operational semantics by illustrating that slight perturbations to a language can have extensive repercussions for the properties of the language.

You have probably noticed that PostFix has a very limited expressive power. The fact that all programs terminate gives us a hint why. Any language in which all programs terminate can't be universal, because any universal language must allow nonterminating computations to be expressed. Even if we don't care about universality (maybe we just want a good calculator language), PostFix suffers from numerous drawbacks. For example, `nget` allows us to "name" numerals by their position relative to the top of the stack, but these positions change as values are pushed and popped, leading to programs that are challenging to read and write. It would be nicer to give unchanging names to values. Furthermore, `nget` only accesses numerals, and there are situations where we need to access executable sequences and use them more than once.

We could address these problems by allowing executable sequences to be copied from any position on the stack and by introducing a general way to name any value; these extensions are explored in exercises. For now, we will consider extending PostFix with a command that just copies the top value on a stack. Since the top value might be an executable sequence, this at least gives us a way to copy executable sequences — something we could not do before.

Consider a new command, `dup`, which duplicates the value at the top of the stack. After execution of this command, the top two values of the stack will be the same. The rewrite rule for `dup` is given below:

$$\langle \texttt{dup} \ . \ Q, V \ . \ S \rangle \Rightarrow \langle Q, V \ . \ V \ . \ S \rangle \qquad [dup]$$

As a simple example of using `dup`, consider the executable sequence `(dup mul)`, which behaves as a squaring subroutine:

```
(postfix 1 (dup mul) exec) -[12]-> 144
(posttfix 2 (dup mul) dup 3 nget swap exec swap 4 nget swap exec)
  -[5,12]-> 169
```

The introduction of `dup` clearly enhances the expressive power of POSTFIX. But adding this innocent little command has a tremendous consequence for the language: it destroys the termination property! Consider the program `(postfix 0 (dup exec) dup exec)`. Executing this program on zero arguments yields the following transition sequence:

$$\begin{aligned}
&\langle \texttt{((dup exec) dup exec)}, [] \rangle \\
\Rightarrow\ &\langle \texttt{(dup exec)}, [\texttt{(dup exec)}] \rangle \\
\Rightarrow\ &\langle \texttt{(exec)}, [\texttt{(dup exec)}, \texttt{(dup exec)}] \rangle \\
\Rightarrow\ &\langle \texttt{(dup exec)}, [\texttt{(dup exec)}] \rangle \\
\Rightarrow\ &\ldots
\end{aligned}$$

Because the rewrite process returns to a previously visited configuration, it is clear that the execution of this program never terminates.

It is not difficult to see why `dup` invalidates the termination proof from Section 3.4.3. The problem is that `dup` can increase the energy of a configuration in the case where the top element of the stack is an executable sequence. Because `dup` effectively creates new commands in this situation, the number of commands executed can be unbounded.

It turns out that extending POSTFIX with `dup` not only invalidates the termination property, but also results in a language that is universal![12] That is, any computable function can be expressed in POSTFIX+{`dup`}.

This simple example underscores that minor changes to a language can have major consequences. Without careful thought, it is never safe to assume that adding or removing a simple language feature or tweaking a rewrite rule will change a language in only minor ways.

We conclude this chapter with numerous exercises that explore various extensions to the POSTFIX language.

▷ **Exercise 3.36** Extend the POSTFIX SOS so that it handles the following commands:

- `pair`: Let V_1 be the top value on the stack and V_2 be the next to top value. Pop both values off of the stack and push onto the stack a pair object $\langle V_2, V_1 \rangle$.
- `fst`: If the top stack value is a pair $\langle V_{fst}, V_{snd} \rangle$, then replace it with V_{fst}. Otherwise signal an error.
- `right`: If the top stack value is a pair $\langle V_{fst}, V_{snd} \rangle$, then replace it with V_{snd}. Otherwise signal an error. ◁

[12] We are indebted to Carl Witty and Michael Frank for showing us that POSTFIX+{`dup`} is universal.

▷ **Exercise 3.37** Extend the POSTFIX SOS so that it handles the following commands:

- `get`: Call the top stack value v_{index} and the remaining stack values (from top down) v_1, v_2, ..., v_n. Pop v_{index} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$, push v_i onto the stack. Signal an error if the stack does not contain at least one value, if v_{index} is not a numeral, or if i is not in the range $[1, n]$. (`get` is like `nget` except that it can copy any value, not just a numeral.)
- `put`: Call the top stack value v_{index}, the next-to-top stack value v_{val}, the remaining stack values (from top down) v_1, v_2, ..., v_n. Pop v_{index} and v_{val} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$, change the slot holding v_i on the stack to hold v_{val}. Signal an error if the stack does not contain at least two values, if v_{index} is not a numeral, or if i is not in the range $[1, n]$. ◁

▷ **Exercise 3.38** Write the following programs in POSTFIX+{`dup`}. You may also use the pair commands from Exercise 3.36 and/or the `get`/`put` commands from Exercise 3.37 in your solution, but they are not necessary — for an extra challenge, program purely in POSTFIX+{`dup`}.

a. A program that takes a single argument (call it n) and returns the nth factorial. The factorial f of an integer is a function such that $(f\ 0) = 1$ and $(f\ n) = (n \times (f\ (n-1)))$ for $n \geq 1$.

b. A program that takes a single argument (call it n) and returns the nth Fibonacci number,. The Fibonacci function f is such that $(f\ 0) = 0$, $(f\ 1) = 1$, and $(f\ n) = ((f\ (n-1)) + (f\ (n-2)))$ for $n \geq 2$. ◁

▷ **Exercise 3.39** Abby Stracksen wishes to extend POSTFIX with a simple means of iteration. She suggests that POSTFIX should have a new command of the form `(for` N `(`Q`))`. Abby describes the behavior of her command with the following rewrite axioms:

$$\begin{array}{c} \langle \texttt{(for}\ N\ \texttt{(}Q_{for}\texttt{))}\ .\ Q_{rest},\ S\rangle \\ \Rightarrow \langle N\ .\ Q_{for}\ @\ [\texttt{(for}\ N_{dec}\ \texttt{(}Q_{for}\texttt{))}]\ @\ Q_{rest},\ S\rangle, \\ \text{where}\ \ N_{dec} = (\textit{calculate}\ \texttt{sub}\ N\ \texttt{1}) \\ \text{and}\ \ (\textit{compare}\ \texttt{gt}\ N\ \texttt{0}) \end{array} \qquad [\textit{for-once}]$$

$$\begin{array}{c} \langle \texttt{(for}\ N\ \texttt{(}Q_{for}\texttt{))}\ .\ Q_{rest},\ S\rangle \Rightarrow \langle Q_{rest},\ S\rangle, \\ \text{where}\ \ \neg\,(\textit{compare}\ \texttt{gt}\ N\ \texttt{0}) \end{array} \qquad [\textit{for-done}]$$

Abby calls her extended language POSTLOOP.

a. Give an informal specification of Abby's `for` command that would be appropriate for a reference manual.

b. Using Abby's `for` semantics, what are the results of executing the following POSTLOOP programs when called on zero arguments?

i. `(postloop 0 1 (for 5 (mul)))`

ii. `(postloop 0 1 (for 5 (2 mul)))`

iii. `(postloop 0 1 (for 5 (add mul)))`

iv. `(postloop 0 0 (for 17 (pop 2 add)))`

v. `(postloop 0 0 (for 6 (pop (for 7 (pop 1 add)))))`

c. Extending POSTFIX with the `for` command does not change its termination property. Show this by extending the termination proof described in the notes in the following way:

i. Define the energy of the `for` command.

ii. Show that the transitions in the [*for-once*] and [*for-done*] rules decrease configuration energy.

d. Bud Lojack has developed a `repeat` command of the form `(repeat` N `(`Q`))` that is similar to Abby's `for` command. Bud defines the semantics of his command by the following rewrite rules:

$$\begin{array}{c}\langle \texttt{(repeat } N \texttt{ (}Q_{rpt}\texttt{))} \texttt{ . } Q_{rest},\ S\rangle \\ \Rightarrow \langle N \texttt{ . (repeat } N_{dec} \texttt{ (}Q_{rpt}\texttt{))} \texttt{ . } Q_{rpt} \texttt{ @ } Q_{rest},\ S\rangle, \\ \text{where } N_{dec} = (\textit{calculate } \texttt{sub } N \texttt{ 1}) \\ \text{and } (\textit{compare } \texttt{gt } N \texttt{ 0})\end{array} \qquad [\textit{repeat-once}]$$

$$\begin{array}{c}\langle \texttt{(repeat } N \texttt{ (}Q_{rpt}\texttt{))} \texttt{ . } Q_{rest},\ S\rangle \Rightarrow \langle Q_{rest},\ S\rangle, \\ \text{where } \neg\, (\textit{compare } \texttt{gt } N \texttt{ 0})\end{array} \qquad [\textit{repeat-done}]$$

Does Bud's `repeat` command have the same behavior as Abby's `for` command? That is, does the following observational equivalence hold?

$$[\texttt{(repeat } N \texttt{ (}Q\texttt{))}] =_{obs} [\texttt{(for } N \texttt{ (}Q\texttt{))}]$$

Justify your answer. ◁

▷ **Exercise 3.40** Alyssa P. Hacker has created POSTSAFE, an extension to POSTFIX with a new command called `sdup`: safe `dup`. The `sdup` command is a restricted form of `dup` that does not violoate the termination property of POSTFIX. The informal semantics for `sdup` is as follows: if the top of the stack is a number or a command sequence that doesn't contain `sdup`, duplicate it; otherwise, signal an error.

As a new graduate student in Alyssa's AHRG (Advanced Hacking Research Group), you are assigned to give an operational semantics for `sdup`, and a proof that all POSTSAFE programs terminate. Alyssa set up several intermediate steps to make your life easier.

a. Write the operational semantics rules that describe the behavior of `sdup`. Model the errors through stuck states. You can use the auxiliary function

$$\mathit{contains\_sdup} : \text{Commands} \to \mathit{Bool}$$

that takes a sequence of commands and checks whether it contains `sdup` or not.

b. Consider the product domain $P = \mathbb{N} \times \mathbb{N}$ (recall that $\mathbb{N}$ is the set of natural numbers, starting with 0). On this domain, Alyssa defined the ordering $<_P$ as follows:

Definition 1 (lexicographic order) $\langle a_1, b_1\rangle <_P \langle a_2, b_2\rangle$ *iff*

i. $a_1 < a_2$ *or*

ii. $a_1 = a_2$ *and* $b_1 < b_2$.

E.g., $\langle 3, 10000\rangle <_P \langle 4, 0\rangle$, $\langle 5, 2\rangle <_P \langle 5, 3\rangle$.

Definition 2 *A* strictly decreasing chain *in P is a sequence of elements $p_1, p_2, \ldots$ such that $\forall i\ .\ p_i \in P$ and $\forall i\ .\ p_{i+1} <_P p_i$.*

i. Consider a finite strictly decreasing chain $p_1, p_2, \ldots, p_k$, where $\forall i\ .\ p_i = \langle a_i, b_i\rangle \in P$, such that $k > b_1 + 1$ (i.e., the chain has more than $b_1 + 1$ elements). Prove that $a_k < a_1$.

ii. Show that there is no infinite strictly decreasing chain in P.

c. Prove that each PostSafe program terminates by defining an appropriate energy function $\mathcal{ES}_{\mathit{config}}$. *Note:* If you need to use some helper functions that are intuitively easy to describe but tedious to define (e.g., *contains_sdup*), just give an informal description of them. ◁

▷ **Exercise 3.41** Sam Antix extends the PostFix language to allow programmers to directly manipulate stacks as first-class values. He calls the resulting language StackFix. StackFix adds three commands to the PostFix collection.

- `package`: This command packages a copy of the stack as a first-class value, S. It then clears the stack, leaving S as the only value on the stack.
- `unpackage`: This command pops the top of the stack, which must be a stack-value S, and replaces the stack with an "unpackaged" version of S.
- `switch`: This command pops the top of the stack, which must be a stack-value, S. Then the rest of the stack is packaged (as if by the `package` command); this results in a new stack-value, S_{rest}. Finally, the stack is completely replaced with an "unpackaged" version of S, and the stack-value S_{rest} is pushed on top of the resulting stack. Thus, `switch` effectively switches the roles of the stack-value on top of the stack and the rest of the stack.

As a warm-up, Sam has written some simple StackFix programs. First-class stack values may be returned as the final result of a program execution; in the case, the outcome is the token `stack-value`, which hides the details of the stack value.

`(stackfix 0 1 2 package)` $\xrightarrow{[]}$ `stack-value`
`(stackfix 0 1 2 package unpackage)` $\xrightarrow{[]}$ `2`
`(stackfix 0 1 2 package 3 switch)` $\xrightarrow{[]}$ {*error: top of stack not stack-value*}
`(stackfix 0 1 2 package 3 swap switch)` $\xrightarrow{[]}$ `stack-value`
`(stackfix 0 2 package 3 swap switch pop)` $\xrightarrow{[]}$ `2`
`(stackfix 0 1 2 package 3 swap switch unpackage)` $\xrightarrow{[]}$ `3`

a. Write a definition of the Value domain for the StackFix language.

b. Give transition rules for the `package`, `unpackage`, and `switch` commands.

c. Does `unpackage` add new expressive power to StackFix? If yes, argue why. If no, provide an equivalent sequence of commands from PostFix+{`package`,`switch`}.

d. Does every StackFix program terminate? Give a short, intuitive description of your reasoning. ◁

▷ **Exercise 3.42** Rhea Storr introduces a new PostFix command called `execs` that permits executing a sequence of commands while saving the old stack. She calls her extended language PostSave.

Rhea asks you to help her define transition rules for PostSave that in several steps move $\langle$`execs` . Q, Q_{exec} . $S\rangle$ to the configuration $\langle Q,\ V\ .\ S\rangle$. This sequence of transformations assumes that the configuration $\langle Q_{exec},\ S\rangle$ will eventually result in a final configuration $\langle[]_{\text{Command}},\ V\ .\ S'\rangle$.

Here are some examples that contrast `exec` with `execs`:

`(postsave 0 1 2 (3 mul) exec add)` $\xrightarrow{[]}$ `7`
`(postsave 0 1 2 (3 mul) execs add)` $\xrightarrow{[]}$ `8`
`(postsave 0 (1) execs)` $\xrightarrow{[]}$ `1`
`(postsave 0 2 3 (mul) execs add add)` $\xrightarrow{[]}$ `11`

To implement the SOS for PostSave, Rhea modifies the configuration space:

$cf \in CF = Layer^*$
$L \in Layer = \text{Commands} \times Stack$

Rhea's transition rule for `execs` is:

$$\langle\texttt{execs}\ .\ Q,\ Q_{exec}\ .\ S\rangle\ .\ L^* \Rightarrow \langle Q_{exec},\ S\rangle\ .\ \langle Q,\ S\rangle\ .\ L^* \qquad [execs]$$

Note that the entire stack is copied into the new layer!

a. If $\langle Q,\ S\rangle \xrightarrow{PF} \langle Q',\ S'\rangle$ is a transition rule in PostFix, provide the corresponding rule in PostSave.

b. Provide the rule for an empty command sequence in the top layer.

c. Show that programs in PostSave are no longer guaranteed to terminate by giving a command sequence that is equivalent to `dup`. ◁

▷ **Exercise 3.43** One of the chief limitations of the PostFix language is that there is no way to name values. In this problem, we consider extending PostFix with a simple naming system. We will call the resulting language PostText.

The grammar for PostText is the same as that for PostFix except that there are three new commands:

C ::= ...
| I [Name]
| `def` [Definition]
| `ref` [Name-reference]

Here, I is an element of the syntactic domain Identifier, which includes all alphabetic names except for the PostText command names (`pop`, `exec`, `def`, etc.), which are treated as reserved words of the language.

The model of the PostText language extends the model of PostFix by including a current dictionary as well as a current stack. A dictionary is an object that maintains bindings between names and values. The commands inherited from PostFix have no effect on the dictionary. The informal behavior of the new commands is as follows:

- I: I is a literal name that is similar to an immutable string literal in other languages. Executing this command simply pushes I on the stack. The Value domain must be extended to include identifiers in addition to numerals and executable sequences.
- `def`: Let v_1 be the top stack value and v_2 be the next to top value. The `def` command pops both values off of the stack and updates the current dictionary to include a binding between v_2 and v_1. v_2 should be a name, but v_1 can be any value (including an executable sequence or name literal). It is an error if v_2 is not a name.
- `ref`: The `ref` command pops the top element v_{name} off of the stack, where v_{name} should be a name I. It looks up the value v_{val} associated with I in the current dictionary and pushes v_{val} on top of the stack. It is an error if there is no binding for I in the current dictionary or if v_{name} is not a name.

For example:

```
(posttext 0 average (add 2 div) def 3 7 average ref exec) -[]-> 5
(posttext 0 a 3 def dbl (2 mul) def a ref
   dbl ref exec 4 dbl ref exec add) -[]-> 14
(posttext 0 a b def a ref 7 def b ref) -[]-> 7
(posttext 0 a 5 def a ref 7 def b ref) -[]-> error {5 is not a name.}
(posttext 0 c 4 def d ref 1 add) -[]-> error {d is unbound.}
```

In an SOS for PostText, the usual PostFix configuration space must be extended to include a dictionary object as a new state component:

$$CF_{PostText} \quad = \quad \text{Commands} \times \text{Stack} \times \text{Dictionary}$$

a. Suppose that a dictionary is represented as a sequence of identifier/value pairs:

$$D \in \text{Dictionary} \ = \ (\text{Identifier} \times \text{Value})^*$$

i. Define the final configurations, input function, and output function for the PostText SOS.

ii. Give the rewrite rules for the I, `def`, and `ref` commands.

b. Redo the above problem, assuming that dictionaries are instead represented as functions from identifiers to values, i.e.,

$$D \in \text{Dictionary} \ = \ \text{Identifier} \rightarrow (\text{Value} \ + \ \{\texttt{unbound}\})$$

where `unbound` is a distinguished token indicating an identifier is unbound in the dictionary.

You may find the following *bind* function helpful:

$$\begin{array}{l} \mathit{bind} : \text{Identifier} \rightarrow \text{Value} \rightarrow \text{Dictionary} \rightarrow \text{Dictionary} \\ = \lambda I_{bind}\, VD\, .\ \ \lambda I_{ref}\, .\ \ \textbf{if}\ I_{bind} = I_{ref}\ \ \textbf{then}\ V\ \ \textbf{else}\ (D\ \ I_{ref})\ \ \textbf{fi} \end{array}$$

bind takes a name, a value, and dictionary, and returns a new dictionary in which there is a binding between the name and value in addition to the existing bindings. (If the name was already bound in the given dictionary, the new binding effectively replaces the old.) ◁

▷ **Exercise 3.44** After several focus-group studies, Ben Bitdiddle has decided that PostFix needs a macro facility. Below is Ben's sketch of the informal semantics of the facility for his extended language, which he dubs PostMac.

Macros are specified at the beginning of a PostMac program, as follows:

$$\texttt{(postmac}\ N_{numargs}\ \texttt{((}I_1\ \ V_1\texttt{)}\ \ldots\ \ \texttt{(}I_n\ \ V_n\texttt{))}\ Q\texttt{)}$$

Each macro $\texttt{(}I_i\ \ V_i\texttt{)}$ creates a command, called $I_i \in$ Identifier, that, when executed, pushes the value V_i (which can be an integer or a command sequence) onto the stack. It is illegal to give macros the names of existing PostFix commands, or to use an identifier more than once in a list of macros. The behavior of programs that do so is undefined. Here are some examples Ben has come up with:

`(postmac 0 ((inc (1 add))) (0 inc exec inc exec))` $\xrightarrow{[]}$ `2`

`(postmac 0 ((A 1) (B (2 mul))) (A B exec))` $\xrightarrow{[]}$ `2`

`(postmac 0 ((A 1) (B (2 mul))) (A C exec))` $\xrightarrow{[]}$ `error`
{*undefined macro C*}

`(postmac 0 ((A 1) (B (C mul)) (C 2)) (A B exec))` $\xrightarrow{[]}$ `2`

`(postmac 0 ((A pop)) (1 A))` $\xrightarrow{[]}$ `error`
{*Ill-formed program: macro bodies must be values, not commands*}

Ben started writing an SOS for PostMac, but had to go make a presentation for some venture capitalists. It is your job to complete the SOS.

Before leaving, Ben made the following changes/additions to the domain definitions:

$M \in \text{MacroList} = (\text{Identifier} \times \text{Value})^*$
$P \in \text{Program} = \text{Commands} \times \text{Intlit} \times \text{MacroList}$
$CF = \text{Commands} \times \text{Stack} \times \text{MacroList}$

$C \in \text{Commands} ::= \ldots \mid I$ [Macro Reference]

He also introduced an auxiliary partial function, *lookup*, with the following signature:

$$lookup : \text{Identifier} \times \text{MacroList} \rightharpoonup \text{Value}$$

If *lookup* is given an identifier and a macro list, it returns the value that the identifier is bound to in the macro list. If there is no such value, *lookup* gets stuck.

a. Ben's notes begin the SOS for PostMac as follows:

$$\frac{\langle Q,\ S\rangle \overset{PF}{\Longrightarrow} \langle Q',\ S'\rangle}{\langle Q, S, M\rangle \overset{PM}{\Longrightarrow} \langle Q', S', M\rangle} \qquad \text{[PostFix } \textit{commands}\text{]}$$

where $\overset{PF}{\Longrightarrow}$ is the original transition relation for PostFix and $\overset{PM}{\Longrightarrow}$ is the new transition relation for PostMac. Complete the SOS for PostMac. Your completed SOS should handle the first four of Ben's examples. Don't worry about ill-formed programs. Model errors as stuck states.

b. Louis Reasoner finds out that your SOS handles macros that depend on other macros. He wants to launch a new advertising campaign with the slogan: "Guaranteed to terminate: PostFix with mutually recursive macros!" Show that Louis' new campaign is a bad idea by writing a nonterminating program in PostMac.

c. When Ben returns from his presentation, he finds out you've written a nonterminating program in PostMac. He decides to restrict the language so nonterminating programs are no longer possible. Ben's restriction is that the body (or value) of a macro cannot use any macros. Ben wants you to prove that this restricted language terminates.

   i. Extend the PostFix energy function so that it assigns an energy to configurations that include macros. Fill in the blanks in Ben's definitions of the

functions $\mathrm{E}_{com}[\![C, M]\!]$, $\mathrm{E}_{seq}[\![Q, M]\!]$ and $\mathrm{E}_{stack}[\![S, M]\!]$ and use these functions to define the configuration energy function $\mathrm{E}_{config}[\![\langle Q, S, M\rangle]\!]$.

$$
\begin{aligned}
\mathrm{E}_{com}[\![\texttt{(}Q\texttt{)}, M]\!] &= \mathrm{E}_{seq}[\![Q,\ M]\!] \\
\mathrm{E}_{com}[\![C, M]\!] &= 1 \text{ (C is not an identifier or an executable sequence)} \\
\mathrm{E}_{com}[\![I, M]\!] &= \\
\mathrm{E}_{seq}[\![[\,]_{\text{Command}},\ M]\!] &= 0 \\
\mathrm{E}_{seq}[\![C\ \texttt{.}\ Q,\ M]\!] &= \\
\mathrm{E}_{stack}[\![[\,]_{\text{Value}},\ M]\!] &= 0 \\
\mathrm{E}_{stack}[\![V\ \texttt{.}\ S, M]\!] &= \\
\mathrm{E}_{config}[\![\langle Q, S, M\rangle]\!] &=
\end{aligned}
$$

ii. Use the extended energy function (for the restricted form of POSTMAC) to show that executing a macro decreases the energy of a configuration. Since it is possible to show all the other commands decrease the energy of a configuration (by adapting the termination proof for PostFix without macros), this will show that the restricted form of POSTMAC terminates. ◁

▷ **Exercise 3.45** Dan M. X. Cope, a Lisp hacker, is unsatisfied with POSTTEXT, the name binding extension of POSTFIX introduced in Exercise 3.43. He claims that there is a better way to add name binding to POSTFIX, and creates a brand new language, POSTLISP, to test out his ideas.

The grammar for POSTLISP is the same as that for POSTFIX except that there are four new commands:

C ::= ...
- | I [Name]
- | `bind` [Push new binding]
- | `unbind` [Remove binding]
- | `lookup` [Name lookup]

Here, I is an element of the syntactic domain Identifier, which includes all alphabetic names except for the POSTLISP command names (`pop`, `exec`, `bind`, etc.), which are treated as reserved words of the language.

The model of the POSTLISP language extends the model of POSTFIX by including a *name stack* for each name. A name stack is a stack of values associated with a name that can be manipulated with the `bind`, `unbind`, and `lookup` commands as described below. The commands inherited from POSTFIX have no effect on the name stacks. The informal behavior of the new commands is as follows:

- I: I is a literal name that is similar to an immutable string literal in other languages. Executing this command simply pushes I onto the stack. The Value domain is extended to include names in addition to numerals and executable sequences.

- `bind`: Let v_1 be the top stack value and v_2 be the next-to-top value. The `bind` command pops both values off of the stack and pushes v_1 onto the name stack associated with v_2. Thus v_2 is required to be a name, but v_1 can be any value (including an executable sequence or name literal). It is an error if v_2 is not a name.

- `lookup`: The command `lookup` pops the top element v_{name} off of the stack, where v_{name} should be a name I. If v_{val} is the value at the top of the name stack associated with I, then v_{val} is pushed onto the stack. (v_{val} is *not* popped off of the name stack.) It is an error if the name stack of I is empty, or if v_{name} is not a name.

- `unbind`: The command `unbind` pops the top element v_{name} off of the stack, where v_{name} should be a name I. It then pops the top value off of the name stack associated with I. It is an error if the name stack of I is empty, or if v_{name} is not a name.

- In the initial state, each name is associated with the empty name stack.

For example:

`(postlisp 0 a 3 bind a lookup)` $\xrightarrow{[]}$ `3`
`(postlisp 0 a 8 bind a lookup a lookup add)` $\xrightarrow{[]}$ `16`
`(postlisp 0 a 4 bind a 9 bind a lookup a unbind a lookup add)` $\xrightarrow{[]}$ `13`
`(postlisp 0 19 a bind a lookup)` $\xrightarrow{[]}$ `error` {`19` *is not a name.*}
`(postlisp 0 average (add 2 div) bind 3 7 average lookup exec)` $\xrightarrow{[]}$ `5`
`(postlisp 0 a b bind a lookup 23 bind b lookup)` $\xrightarrow{[]}$ `23`
`(postlisp 0 c 4 bind d lookup 1 add)` $\xrightarrow{[]}$ `error` {`d` *name stack is empty.*}
`(postlisp 0 b unbind)` $\xrightarrow{[]}$ `error` {`b` *name stack is empty*}

In an SOS for PostLisp, the usual PostFix configuration space must be extended to include the name stacks as a new state component. Name stacks are bundled up into an object called a *name file*.

$$CF_{PostLisp} = \text{Commands} \times \text{Stack} \times \text{NameFile}$$
$$F \in \text{NameFile} = \text{Name} \rightarrow \text{Stack}$$

A NameFile is a function mapping a name to the stack of values bound to the name. If F is a name file, then $(F\ I)$ is the stack associated with I in F. The notation $F[I = S]$ denotes a name file that is identical to F except that I is mapped to S.

a. Define the final configurations, input function, and output function for the PostLisp SOS.

b. Give the rewrite rules for the I, `bind`, `unbind`, and `lookup` commands. ◁

▷ **Exercise 3.46** Abby Stracksen is bored with vanilla PostFix (it's not even universal!) and decides to add a new feature, which she calls the *heap*. A heap maps locations to elements from the Value domain, where locations are simply integers:

Location = Intlit

Note that a location can be any integer, including a negative one. Furthermore, integers and locations can be used interchangeably in Abby's language, very much like pointers in pre-ANSI C.

Abby christens her new language PostHeap. The grammar for PostHeap is the same as that for PostFix except that there are three new commands:

C ::= ...
| `allocate` [Allocation]
| `store` [Store in heap location]
| `access` [Access from heap location]

The commands inherited from PostFix have no effect on the heap. The informal behavior of the new commands is as follows:

- `allocate`: Executing this command pushes onto the stack a location that is not used in the heap.
- `store`: Let v_1 be the top stack value and v_2 be the next-to-top value. The `store` command pops v_1 off the stack and writes it into the heap at location v_2. Thus v_1 can be any element from the Value domain and v_2 has to be an Intlit. It is an error if v_2 is not an Intlit. Note that v_2 remains on the stack.
- `access`: Let v_1 be the top stack value. The `access` command reads from the heap at location v_1 and pushes the result onto the stack. Thus v_1 has to be an Intlit. It is an error if v_1 is not an Intlit or if the heap at location v_1 has not been written with `store` before. Note that v_1 remains on the stack.

For example:

`(postheap 0 allocate)` $\xrightarrow{[]}$ *N* {*implementation dependent*}
`(postheap 0 allocate 5 store access)` $\xrightarrow{[]}$ `5`
`(postheap 0 allocate 5 store 4 swap access swap pop add)` $\xrightarrow{[]}$ `9`
`(postheap 0 4 5 store)` $\xrightarrow{[]}$ `4`
`(postheap 0 4 5 store access)` $\xrightarrow{[]}$ `5`
`(postheap 0 access)` $\xrightarrow{[]}$ `error` {*no location given*}
`(postheap 0 allocate access)` $\xrightarrow{[]}$ `error` {*location has not been written*}
`(postheap 0 5 store)` $\xrightarrow{[]}$ `error` {*no location given*}

After sketching this initial description of the heap, Abby realizes that it is already 8:55 on a Friday night and she goes off to watch the X-Files. It is your task to flesh out her initial draft:

a. Give the definition of the Heap domain and the configuration domain *CF*.

b. Let *access-from-heap* be a partial function that, given a Location and a Heap in which Location has been bound, returns an element from the Value domain. In other words, *access-from-heap* has the following signature and definition:

$$\textit{access-from-heap}\text{: Location} \to \text{Heap} \rightharpoonup \text{Value}$$

$(\textit{access-from-heap}\ N\ \langle N, V\rangle\ .\ H) = V$

$(\textit{access-from-heap}\ N_1\ \langle N_2, V\rangle\ .\ H) = (\textit{access-from-heap}\ N_1\ H)$, where $N_1 \neq N_2$

Give the rewrite rules for the `allocate`, `store`, and `access` commands. You may use *access-from-heap*.

c. Is PostHeap a universal programming language? Explain your answer.

d. Abby is concerned about security because PostHeap treats integers and locations interchangeably. Since her programs don't use this "feature", she decides to restrict the language by disallowing pointer arithmetic. She wants to use *tags* to distinguish locations from integers. Abby redefines the Value domain as follows:

$$V \in \text{Value} = (\text{Intlit} \times \text{Tag}) + \text{Command}$$
$$\text{Tag} = \{\texttt{integer}, \texttt{pointer}\}$$

Informally, integers and locations are represented as pairs on the stack: integers are paired with the `integer` tag, while locations are paired with the `pointer` tag.

Give the revised rewrite rules for integers, `add`, `allocate`, `store`, and `access`. ◁

▷ **Exercise 3.47‡** Prove that PostFix+{`dup`} is universal. This can be done by showing how to translate any Turing machine program into a PostFix+{`dup`} program. Assume that integer numerals may be arbitrarily large in magnitude. ◁

Reading

Early approaches to operational semantics defined the semantics of programming languages by translating them to standard abstract machines. Landin's SECD machine [Lan64] is a classic example of such an abstract machine. Plotkin [Plo75] used it to study the semantics of the lambda calculus.

Later, Plotkin introduced Structured Operational Semantics [Plo81] as a more direct approach to specifying an operational semantics. The context-based approach to specifying transition relations for small-step operational semantics was invented by Felleisen and Friedman in [FF86] and explored in a series of papers culminating in [FH92]. Big-step (natural) semantics was introduced by

Kahn in [Kah87]. A concise overview of various approaches to semantics, including several forms of operational semantics, can be found in the first chapter of [Gun92]. The early chapters of [Win93] present an introduction to operational semantics in the context of a simple imperative language.

Other popular forms of operational semantics include **term rewriting systems** ([DJ90, BN98]) and **graph rewriting systems** ([Cou90]).

Chapter 4

Denotational Semantics

But this denoted a foregone conclusion

— *Othello, William Shakespeare*

4.1 The Denotational Semantics Game

We have seen how an operational semantics is a natural tool for evaluating programs and proving properties like termination. However, it is less than ideal for many purposes. A framework based on transitions between configurations of an abstract machine is usually better suited for reasoning about complete programs than program fragments. In PostFix, for instance, we had to extend the operational semantics with elaborate notions of observational equivalence and transform equivalence in order to effectively demonstrate the interchangeability of command sequences. Additionally, the emphasis on syntactic entities in an operational semantics can complicate reasoning. For example, syntactically distinct executable sequence answers in PostFix must be treated as the same observable value in order to support a non-trivial notion of observational equivalence for command sequences. Finally, the step-by-step nature of an operational semantics can suggest notions of time and dependency that are not essential to the language being defined. For example, an operational semantics for the expression language EL might specify that the left operand of a binary operator is evaluated before the right even though this order may be impossible to detect in practice.

An alternative framework for reasoning about programs is suggested by the notion of transform equivalence developed for PostFix. According to this notion, each PostFix command sequence is associated with a stack transform that

describes how the sequence maps an input stack to an output stack. It is natural to view these stack transforms as functions. For example, the stack transform associated with the command sequence [3, add] would be an *add3* function with the following graph:[1]

$$\{\langle errorStack, errorStack\rangle,\ \langle[\,], errorStack\rangle, \ldots,$$
$$\langle[-1],[2]\rangle,\ \langle[0],[3]\rangle,\ \langle[1],[4]\rangle, \ldots,$$
$$\langle[add3], errorStack\rangle,\ \langle[mul2], errorStack\rangle, \ldots,$$
$$\langle[5,\ 23],[8,\ 23]\rangle,\ \langle[5,\ mul2,\ 17,\ add3],[8,\ mul2,\ 17,\ add3]\rangle, \ldots\}.$$

Here, *errorStack* stands for a distinguished error stack analogous to S_{error} in the extended PostFix SOS. Stack elements that are executable sequences are represented by their stack transforms (e.g., *add3* and *mul2*) rather than some syntactic phrase.

Associating stack transform functions with command sequences has several benefits. First, this perspective directly supports a notion of equivalence for program phrases. For example, the *add3* function is the stack transform associated with the sequence [1, add, 2, add] as well as the sequence [3, add]. This implies that the two sequences are behaviorally indistinguishable and can be safely interchanged in any PostFix context. The fact that stack elements that are executable sequences are represented by functions rather than syntactic entities greatly simplifies this kind of reasoning.

The other major benefit of this approach is that the stack transform associated with the concatenation of two sequences is easily composed from the stack transforms of the component sequences. For example, suppose that the sequence [2, mul] is modeled by the *mul2* function, whose graph is sketched below:

$$\{\langle errorStack, errorStack\rangle,\ \langle[\,], errorStack\rangle, \ldots,$$
$$\langle[-1],[-2]\rangle,\ \langle[0],[0]\rangle,\ \langle[1],[2]\rangle, \ldots,$$
$$\langle[add3], errorStack\rangle,\ \langle[mul2], errorStack\rangle, \ldots,$$
$$\langle[5,\ 23],[10,\ 23]\rangle,\ \langle[5,\ mul2,\ 17,\ add3],[10,\ mul2,\ 17,\ add3]\rangle, \ldots\}.$$

Then the stack transform of [3, add, 2, mul] = [3, add] @ [2, mul] is simply the function $mul2 \circ add3$, whose graph is:

$$\{\langle errorStack, errorStack\rangle,\ \langle[\,], errorStack\rangle, \ldots,$$
$$\langle[-1],[4]\rangle,\ \langle[0],[6]\rangle,\ \langle[1],[8]\rangle, \ldots,$$
$$\langle[add3], errorStack\rangle,\ \langle[mul2], errorStack\rangle, \ldots,$$
$$\langle[5,\ 23],[16,\ 23]\rangle,\ \langle[5,\ mul2,\ 17,\ add3],[16,\ mul2,\ 17,\ add3]\rangle, \ldots\}.$$

[1] Here, and for the rest of this chapter, we rely heavily on the metalanguage concepts and notations described in Appendix A. Consult this appendix as necessary to unravel the formalism.

Similarly the stack transform of $[2, \texttt{mul}, 3, \texttt{add}] = [2, \texttt{mul}]$ @ $[3, \texttt{add}]$ is the function *add3* ∘ *mul2*, whose graph is:

$$\{\langle errorStack, errorStack\rangle,\ \langle[\,], errorStack\rangle, \ldots,$$
$$\langle[-1], [1]\rangle,\ \langle[0], [3]\rangle,\ \langle[1], [5]\rangle, \ldots,$$
$$\langle[add3], errorStack\rangle,\ \langle[mul2], errorStack\rangle, \ldots,$$
$$\langle[5,\ 23], [13,\ 23]\rangle,\ \langle[5,\ mul2,\ 17,\ add3], [13,\ mul2,\ 17,\ add3]\rangle, \ldots\}.$$

The notion that the meaning of a program phrase can be determined from the meaning of its parts is the essence of a framework called **denotational semantics**. A denotational semantics determines the meaning of a phrase in a compositional way based on its static structure rather than on some sort of dynamically changing configuration. Unlike an operational semantics, a denotational semantics emphasizes *what* the meaning of a phrase is, not *how* the phrase is evaluated. The name "denotational semantics" is derived from its focus on the mathematical values that phrases "denote."

The basic structure of the denotational framework is illustrated in Figure 4.1. A denotational semantics consists of three parts:

1. A **syntactic algebra** that describes the abstract syntax of the language under study. This can be specified by the s-expression grammar approach introduced in Chapter 2.

2. A **semantic algebra** that models the meaning of program phrases. A semantic algebra consists of a collection of semantic domains along with functions that manipulate these domains. The meaning of a program may be something as simple as an element of a primitive semantic domain like *Int*, the domain of integers. More typically, the meaning of a program is an element of a function domain that maps **context domains** to an **answer domain**, where

   - Context domains are the denotational analog of state components in an SOS configuration. They model such entities as name/value associations, the current contents of memory, and control information.
   - An answer domain represents the possible meanings of programs. In addition to a component that models what we would normally think of as being the result of a program phrase, the answer domain may also include components that model context information that was transformed by the program.

3. A **meaning function** that maps elements of the syntactic algebra (i.e., nodes in the abstract syntax trees) to their meanings in the semantic algebra. Each phrase is said to **denote** its image under the meaning function.

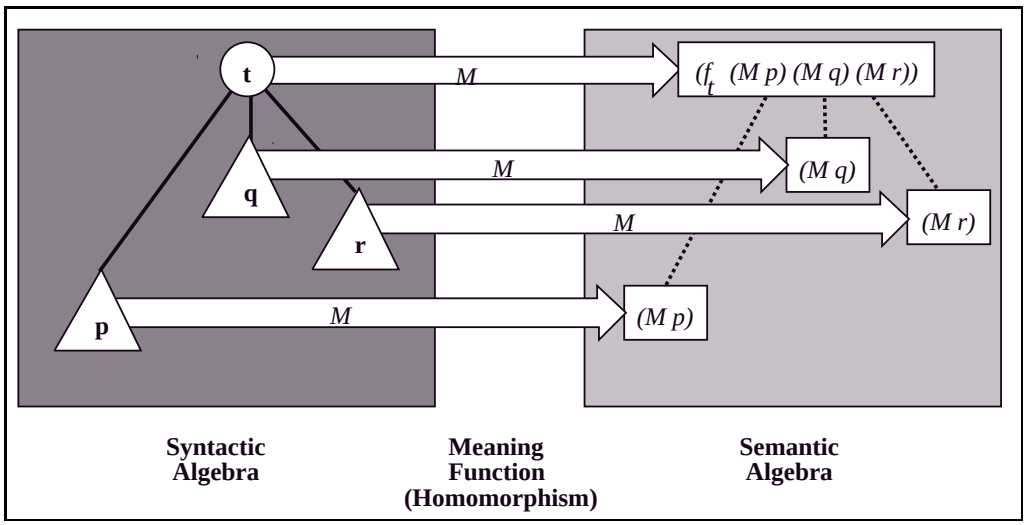


Figure 4.1: The denotational semantics "game board."

Not any function can serve as a meaning function; the function must be a **homomorphism** between the syntactic algebra and the semantic algebra. This is just the technical condition that constrains the meaning of an abstract syntax tree node to be determined from the meaning of its subnodes. It can be stated more formally as follows:

> Suppose M is a meaning function and t is a node in an abstract syntax tree, with children $t_1, \ldots, t_k$. Then
>
> $$(M\ t) \text{ must equal } (f_t\ (M\ t_1)\ \ldots\ (M\ t_k))$$
>
> where f_t is a function that is determined by the syntactic class of t.

The advantage of restricting meaning functions to homomorphisms is that their structure-preserving behavior greatly simplifies reasoning. This design choice accounts for the compositional nature of denotational semantics, whose essence is summarized by the motto "the meaning of the whole is composed out of the meaning of the parts".

4.2 A Denotational Semantics for EL

As our first example, we will develop a denotational semantics for the EL expression language. We begin with a pared-down version of the language and show how the semantics changes when adding features to yield full EL.

4.2.1 Step 1: Restricted ELMM

Recall that ELMM (Figure 3.6) is a simple expression language in which programs are expressions, and expressions are trees of binary operators (`+`,`-`,`*`,`/`,`%`) whose leaves are integer numerals. For the moment, let's ignore the `/` and `%` operations, because removing the possibility of divide-by-zero and remainder-by-zero errors simplifies the semantics. In a version of ELMM without `/` and `%`, the meaning of each numeral, expression, and program is just an integer.

This meaning is formalized in Figure 4.2. There is only one semantic domain: the domain *Int* of integers. The meaning of an ELMM program is specified by a collection of so-called **valuation functions**, one for each syntactic domain defined by the abstract syntax for the language. For each syntactic domain, the name of the associated valuation function is usually a script version of the metavariable that ranges over that domain. For example, $\mathcal{P}$ is the valuation function for $P \in$ Program, $\mathcal{NE}$ is the valuation function for $NE \in$ NumExp, and so on.

The meaning $\mathcal{P}[\![\texttt{(elmm } NE_{body}\texttt{)}]\!]$ of an ELMM program `(elmm` NE_{body}`)` is simply the integer $\mathcal{NE}[\![NE_{body}]\!]$ denoted by its body expression NE_{body}. Since an ELMM numerical expression may be either an integer numeral or an arithmetic operation, the definition of $\mathcal{NE}$ has a clause for each of these two cases. In the integer numeral case, the $\mathcal{N}$ function maps the syntactic representation of an integer numeral into a mathematical integer. We will treat integer numerals as atomic entities, but their meaning could be determined in a denotational fashion from their component signs and digits (see Exercise 4.1). In the arithmetic operation case, the $\mathcal{A}$ function maps the operator (one of `+`, `-`, and `*`) into a binary integer function that determines the meaning of the operation from the meanings of the operands.

Figure 4.3 illustrates how the denotational semantics for the restricted version of ELMM can be used to determine the meaning of the sample ELMM program `(elmm (* (+ 1 2) (- 9 5)))`. Because $\mathcal{P}$ maps programs to their meanings, $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$ *is* the meaning of this program. However, this fact is not very useful as stated because the element of *Int* denoted by the program is not immediately apparent from the form of the metalanguage expression $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$. We would like to massage the metalanguage expression for the meaning of a program into another metalanguage expression more recognizable as an element of the answer domain. We do this by using **equational reasoning** to simplify the metalanguage expression. That is, we are allowed to make any simplifications that are allowed by usual mathematical reasoning about the entities denoted by the metalanguage expressions. Equational reasoning allows such manipulations as:

Semantic Domain

$i \in Int = \{\ldots\ ,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\}$

Valuation Functions

$\mathcal{P}$: Program $\rightarrow$ *Int*
$\mathcal{P}[\![\texttt{(elmm}\ NE_{body}\texttt{)}]\!] = \mathcal{NE}[\![NE_{body}]\!]$

$\mathcal{NE}$: NumExp $\rightarrow$ *Int*
$\mathcal{NE}[\![N]\!] = (\mathcal{N}[\![N]\!])$
$\mathcal{NE}[\![\texttt{(}A\ \ NE_1\ \ NE_2\texttt{)}]\!] = (\mathcal{A}[\![A]\!]\ \ (\mathcal{NE}[\![NE_1]\!])\ \ (\mathcal{NE}[\![NE_2]\!]))$

$\mathcal{A}$: ArithmeticOperator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Int*)
$\mathcal{A}[\![\texttt{+}]\!] = +_{Int}$
$\mathcal{A}[\![\texttt{-}]\!] = -_{Int}$
$\mathcal{A}[\![\texttt{*}]\!] = \times_{Int}$

$\mathcal{N}$: Intlit $\rightarrow$ *Int*
$\mathcal{N}$ maps integer numerals to the integer numbers they denote.

Figure 4.2: Denotational semantics for a version of ELMM without `/` and `%`.

$\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$
$= \mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!]$
$= (\mathcal{A}[\![\texttt{*}]\!]\ \ (\mathcal{NE}[\![\texttt{(+ 1 2)}]\!])\ \ (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]))$
$= ((\mathcal{NE}[\![\texttt{(+ 1 2)}]\!]) \times_{Int} (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]))$
$= ((\mathcal{A}[\![\texttt{+}]\!]\ \ (\mathcal{NE}[\![\texttt{1}]\!])\ \ (\mathcal{NE}[\![\texttt{2}]\!])) \times_{Int} (\mathcal{A}[\![\texttt{-}]\!]\ \ (\mathcal{NE}[\![\texttt{9}]\!])\ \ (\mathcal{NE}[\![\texttt{5}]\!])))$
$= ((1 +_{Int} 2) \times_{Int} (9 -_{Int} 5))$
$= (3 \times_{Int} 4)$
$= 12$

Figure 4.3: Meaning of a sample program in restricted ELMM.

- substituting equals for equals;
- applying functions to arguments;
- equating two function-denoting expressions when, for each argument, they map that argument to the same result (this is called **extensionality**).

Instances of equational reasoning are organized into **equational proofs** that contain a series of equalities. Figure 4.3 presents an equational proof that $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$ is equal to the integer 12. Each equality in the proof is justified by familiar mathematical rules. For example, the equality

$$\mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!] = (\mathcal{A}[\![\texttt{*}]\!]\ (\mathcal{NE}[\![\texttt{(+ 1 2)}]\!])\ (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]))$$

is justified by the arithmetic operation clause in the definition of $\mathcal{NE}$, while the equality

$$((1 +_{Int} 2) \times_{Int} (9 -_{Int} 5)) = (3 \times_{Int} 4)$$

is justified by algebraic rules for manipulating integers. We emphasize that $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$, as well as every other line in Figure 4.3, denotes exactly the same integer. The whole purpose of the equational proof is to simplify the original expression into another metalanguage expression whose form more directly expresses the meaning of the program.

4.2.2 Step 2: Full ELMM

What happens to the denotational semantics for ELMM if we add back in the `/` and `%` operators? We now have to worry about the meaning of expressions like `(/ 1 0)` and `(% 2 0)`. We will model the meaning of such expressions by the distinguished token *error*. Since ELMM programs, numerical expressions, and arithmetic operators can now return errors in addition to integers, we invent an *Answer* domain with both of these kinds of entities to represent their meanings and change the valuation functions $\mathcal{P}$, $\mathcal{NE}$, and $\mathcal{A}$ accordingly (Figure 4.4). The integer numeral clause for $\mathcal{NE}$ now needs the injection $Int \mapsto Answer$, and the arithmetic operation clause must now propagate any errors found in the operands. The $\mathcal{A}$ clauses for `/` and `%` handle specially the case where the second operand is zero, and $Int \mapsto Answer$ injections must be used in the "regular" cases for all operators.

In full ELMM, the sample program `(elmm (* (+ 1 2) (- 9 5)))` has the meaning $(Int \mapsto Answer\ 12)$. Figure 4.5 presents an equational proof of this fact. All the pattern matching clauses appearing in the proof are there to handle the propagation of errors. The sample program has no errors, but we could

Semantic Domains

$i \in Int = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$Error = \{error\}$
$a \in Answer = Int + Error$

Valuation Functions

$\mathcal{P}$: Program $\rightarrow$ *Answer*

$\mathcal{P}[\![\texttt{(elmm } NE\texttt{)}]\!] = \mathcal{NE}[\![NE]\!]$

$\mathcal{NE}$: NumExp $\rightarrow$ *Answer*

$\mathcal{NE}[\![N]\!] = (Int \mapsto Answer\ (\mathcal{N}[\![N]\!]))$

$\mathcal{NE}[\![\texttt{(}A\ NE_1\ NE_2\texttt{)}]\!] =$ **matching** $\langle \mathcal{NE}[\![NE_1]\!], \mathcal{NE}[\![NE_2]\!] \rangle$
$\triangleright \langle (Int \mapsto Answer\ i_1), (Int \mapsto Answer\ i_2) \rangle\ [\![\ (\mathcal{A}[\![A]\!]\ i_1\ i_2)$
$\triangleright$ **else** $(Error \mapsto Answer\ error)$ **endmatching**

$\mathcal{A}$: ArithmeticOperator $\rightarrow (Int \rightarrow Int \rightarrow Answer)$

$\mathcal{A}[\![\texttt{+}]\!] = \lambda i_1\, i_2\, .\ (Int \mapsto Answer\ (i_1 +_{Int} i_2))$
`-` and `*` are handled similarly.

$\mathcal{A}[\![\texttt{/}]\!] = \lambda i_1\, i_2\, .$ **if** $i_2 = 0$
then $(Error \mapsto Answer\ error)$
else $(Int \mapsto Answer\ (i_1\ /_{Int}\ i_2))$ **fi**
`%` is handled similarly.

$\mathcal{N}$: Intlit $\rightarrow$ *Int*
$\mathcal{N}$ maps integer numerals to the integer numbers they denote.

Figure 4.4: Denotational semantics for a version of ELMM with `/` and `%`.

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!] \\
= \mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!] \\
= \textbf{matching}\ \langle \mathcal{NE}[\![\texttt{(+ 1 2)}]\!], \mathcal{NE}[\![\texttt{(- 9 5)}]\!] \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle \textbf{matching}\ \langle \mathcal{NE}[\![\texttt{1}]\!], \mathcal{NE}[\![\texttt{2}]\!] \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_3), (\mathit{Int} \mapsto \mathit{Answer}\ i_4) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{+}]\!]\ i_3\ i_4) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\,, \\
\qquad\qquad \textbf{matching}\ \langle \mathcal{NE}[\![\texttt{9}]\!], \mathcal{NE}[\![\texttt{5}]\!] \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_5), (\mathit{Int} \mapsto \mathit{Answer}\ i_6) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{-}]\!]\ i_5\ i_6) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\ \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ 1), (\mathit{Int} \mapsto \mathit{Answer}\ 2) \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_3), (\mathit{Int} \mapsto \mathit{Answer}\ i_4) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{+}]\!]\ i_3\ i_4) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\,, \\
\qquad\qquad \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ 9), (\mathit{Int} \mapsto \mathit{Answer}\ 5) \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_5), (\mathit{Int} \mapsto \mathit{Answer}\ i_6) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{-}]\!]\ i_5\ i_6) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\ \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle (\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2)\,, (\mathcal{A}[\![\texttt{-}]\!]\ 9\ 5) \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ (1 +_{\mathit{Int}} 2)), (\mathit{Int} \mapsto \mathit{Answer}\ (9 +_{\mathit{Int}} 5)) \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ 3), (\mathit{Int} \mapsto \mathit{Answer}\ 4) \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= (\mathcal{A}[\![\texttt{*}]\!]\ 3\ 4) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ (3 \times_{\mathit{Int}} 4)) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ 12)
\end{array}
$$

Figure 4.5: Meaning of a sample program in full ELMM.

introduce one by replacing the subexpression `(- 9 5)` by `(/ 9 0)`. Then the part of the proof beginning

$$= \textbf{matching}\ \langle(\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2), (\mathcal{A}[\![\texttt{-}]\!]\ 9\ 5)\rangle\ \ldots$$

would become:

$$\begin{aligned}
&= \textbf{matching}\ \langle(\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2), (\mathcal{A}[\![\texttt{/}]\!]\ 9\ 0)\rangle \\
&\quad \triangleright \langle(\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2)\rangle\ [\![\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
&\quad \triangleright \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
&= \textbf{matching}\ \langle(\mathit{Int} \mapsto \mathit{Answer}\ (1 +_{\mathit{Int}} 2)), (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\rangle \\
&\quad \triangleright \langle(\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2)\rangle\ [\![\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
&\quad \triangleright \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
&= (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error}).
\end{aligned}$$

Expressing error propagation via explicit pattern matching makes the equational proof in Figure 4.5 rather messy. As in programming, in denotational semantics it is good practice to create abstractions that capture common patterns of behavior and hide messy details. This can improve the clarity of the definitions and proofs while at the same time making them more compact.

We illustrate this kind of abstraction by introducing the following higher-order function for simplifying error handling in ELMM:

$$\begin{aligned}
&\mathit{with\text{-}int} : \mathit{Answer} \to (\mathit{Int} \to \mathit{Answer}) \to \mathit{Answer} \\
&= \lambda a f\ .\ \textbf{matching}\ \ a \\
&\qquad \triangleright (\mathit{Int} \mapsto \mathit{Answer}\ i)\ [\![\ (f\ \ i) \\
&\qquad \triangleright \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\ .
\end{aligned}$$

with-int takes an answer a and a function f from integers to answers and returns an answer. It automatically propagates errors, in the sense that it maps an input error answer to an output error answer. The function f specifies what is done for inputs that are integer answers. Thus, *with-int* hides details of error handling and extracting integers from integer answers.

A metalanguage expression of the form $(\mathit{with\text{-}int}\ \ a\ \ (\lambda i\ .\ E))$ serves as a kind of binding construct, i.e., a construct that introduces a name for a value. One way to pronounce this is:

> "If a is an integer answer, then let i name the integer in E and return the value of E. Otherwise, a must be an error, in which case an error should be returned."

The following equalities involving *with-int* are useful:

$$(\mathit{with\text{-}int}\ \ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ f) = (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})$$

$$(\mathit{with\text{-}int}\ \ (\mathit{Int} \mapsto \mathit{Answer}\ i)\ f) = (f\ \ i)$$

$$(\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![N]\!])\ \ f) = (f\ \ (\mathcal{N}[\![N]\!]))$$

$$
\begin{aligned}
&\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!] \\
&= \mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!] \\
&= \textit{with-int}\ (\mathcal{NE}[\![\texttt{(+ 1 2)}]\!]) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]) \\
&\qquad (\lambda i_2\ .\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2))) \\
&= \textit{with-int}\ (\textit{with-int}\ (\mathcal{NE}[\![\texttt{1}]\!]) \\
&\qquad (\lambda i_3\ .\ \textit{with-int}\ (\mathcal{NE}[\![\texttt{2}]\!]) \\
&\qquad\quad (\lambda i_4\ .\ (\mathcal{A}[\![\texttt{+}]\!]\ i_3\ i_4)))) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\textit{with-int}\ (\mathcal{NE}[\![\texttt{9}]\!]) \\
&\qquad\quad (\lambda i_5\ .\ \textit{with-int}\ (\mathcal{NE}[\![\texttt{5}]\!]) \\
&\qquad\qquad (\lambda i_6\ .\ (\mathcal{A}[\![\texttt{-}]\!]\ i_5\ i_6)))) \\
&\qquad (\lambda i_2\ .\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2))) \\
&= \textit{with-int}\ (\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\mathcal{A}[\![\texttt{-}]\!]\ 9\ 5) \\
&\qquad (\lambda i_2\ .\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2))) \\
&= \textit{with-int}\ (\textit{Int} \mapsto \textit{Answer}\ (1 +_{\textit{Int}} 2)) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\textit{Int} \mapsto \textit{Answer}\ (9 -_{\textit{Int}} 5)) \\
&\qquad (\lambda i_2\ .\ (\textit{Int} \mapsto \textit{Answer}\ (i_1 \times_{\textit{Int}} i_2)))) \\
&= (\textit{Int} \mapsto \textit{Answer}\ ((1 +_{\textit{Int}} 2) \times_{\textit{Int}} (9 -_{\textit{Int}} 5))) \\
&= (\textit{Int} \mapsto \textit{Answer}\ (3 \times_{\textit{Int}} 4)) \\
&= (\textit{Int} \mapsto \textit{Answer}\ 12)
\end{aligned}
$$

Figure 4.6: Example illustrating how *with-int* hides error propagation.

Using *with-int*, the $\mathcal{NE}$ valuation clause for arithmetic expressions can be redefined as:

$$
\begin{aligned}
&\mathcal{NE}[\![\texttt{(}A\ \ NE_1\ \ NE_2\texttt{)}]\!] \\
&= \textit{with-int}\ \ (\mathcal{NE}[\![NE_1]\!])\ \ (\lambda i_1\ .\ (\textit{with-int}\ \ (\mathcal{NE}[\![NE_2]\!])\ \ (\lambda i_2\ .\ (\mathcal{A}[\![A]\!]\ \ i_1\ \ i_2)))).
\end{aligned}
$$

With this modified definition and the above *with-int* equalities, details of error propagation can be hidden in equational proofs for ELMM meanings (see Figure 4.6).

One of the powers of lambda notation is that it supports the invention of new binding constructs like *with-int* via higher-order functions without requiring any new syntactic extensions to the metalanguage. We will make extensive use of this power to simplify our future denotational definitions. Later we will see how this idea appears in practical programming under as **monadic style** (Chapter 8) and **continuation passing style** (Chapters 9 and 17).

4.2.3 Step 3: ELM

The ELM language (Exercise 3.10) is obtained from ELMM by adding indexed input via the expression `(arg` N_{index}`)`, where N_{index} specifies the index (starting at 1) of a program argument. The program form is `(elm` $N_{numargs}$ NE_{body}`)`, where N_{numarg} indicates the number of integer arguments expected by the program when it is executed. Intuitively, the meaning of ELM programs and numerical expressions must now be extended to include the program arguments. In Figure 4.7, this is expressed by modeling the meaning of programs and expressions as functions with signature $Int^* \rightarrow Answer$ that map a sequence of integers (the program arguments) to an answer (either an integer or an error). The program argument sequence i^* must be "passed down" the syntax tree to the body of a program and the operands of an arithmetic operation so that they can eventually be referenced in an `arg` form at a leaf of the syntax tree. The `elm` program form must check that the number of supplied arguments matches the expected number of arguments $i_{numargs}$, and the `arg` form must check that the index i_{index} is between 1 and the number of arguments, inclusive.

Figure 4.8 uses denotational definitions to find the result of applying the ELM program `(elm 2 (+ (arg 2) (* (arg 1) 3)))` to the argument sequence $[4, 5]$. The equational proof assumes the following equalities, which are easy to verify:

$$(\textit{with-int}\ \ (\mathcal{NE}[\![N]\!]\ \ i^*)\ \ f) = (f\ \ (\mathcal{N}[\![N]\!]))$$

$$(\textit{with-int}\ \ (\mathcal{NE}[\![\texttt{(arg}\ N\texttt{)}]\!]\ \ [i_1, \ldots, i_k, \ldots, i_n])\ \ f) = (f\ \ i_k)\text{, where } \mathcal{N}[\![N]\!] = \mathrm{k}$$

$$(\textit{with-int}\ \ (\mathcal{A}[\![A]\!]\ \ i_1\ \ i_2)\ \ f) = (f\ \ i_{res})\text{, where } (\mathcal{A}[\![A]\!]\ \ i_1\ \ i_2) = (\textit{Int} \mapsto \textit{Result}\ \ i_{res})$$

In Figure 4.8, if we replace the concrete argument integers 4 and 5 by abstract integers i_{arg1} and i_{arg2}, respectively, then the result would be

$$(\textit{Int} \mapsto \textit{Answer}\ \ (i_{arg2} +_{\textit{Int}}\ (i_{arg1} \times_{\textit{Int}} 3))).$$

Based on this observation, we can give a meaning to the sample program itself (i.e., without applying it to particular arguments). Such a meaning must be abstracted over an arbitrary argument sequence:

$$\begin{aligned}
&\mathcal{P}[\![\texttt{(elmm 2 (+ (arg 2) (* (arg 1) 3)))}]\!] \\
&= \lambda i^*.\ \textbf{matching}\ \ i^* \\
&\qquad \triangleright\, [i_{arg1}, i_{arg2}]\ [\![\ (\textit{Int} \mapsto \textit{Answer}\ \ (i_{arg2} +_{\textit{Int}}\ (i_{arg1} \times_{\textit{Int}} 3))) \\
&\qquad \triangleright\, \textbf{else}\ (\textit{Error} \mapsto \textit{Answer}\ \ \textit{error})\ \textbf{endmatching}\,.
\end{aligned}$$

Here we have translated the **if** that appears in the $\mathcal{P}$ definition in Figure 4.7 into an equivalent **matching** construct that gives the names i_{arg1} and i_{arg2} to

Semantic Domains

$i \in \mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathit{Error} = \{\mathit{error}\}$
$a \in \mathit{Answer} = \mathit{Int} + \mathit{Error}$

Valuation Functions

$\mathcal{P} : \text{Program} \to \mathit{Int}^* \to \mathit{Answer}$

$\mathcal{P}[\![(\texttt{elm}\ N_{numargs}\ NE_{body})]\!]$
$= \lambda i^* .\ \textbf{if}\ (\mathit{length}\ i^*) = \mathcal{N}[\![N_{numargs}]\!]$
$\textbf{then}\ \mathcal{NE}[\![NE]\!]\ i^*$
$\textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{fi}$

$\mathcal{NE} : \text{NumExp} \to \mathit{Int}^* \to \mathit{Answer}$

$\mathcal{NE}[\![N_{num}]\!] = \lambda i^* .\ (\mathit{Int} \mapsto \mathit{Answer}\ \mathcal{N}[\![N_{num}]\!])$

$\mathcal{NE}[\![(\texttt{arg}\ N_{index})]\!] = \lambda i^* .\ \textbf{if}\ (1 \leq \mathcal{N}[\![N_{index}]\!])\ \textbf{and}\ (\mathcal{N}[\![N_{index}]\!] \leq (\mathit{length}\ i^*))$
$\textbf{then}\ (\mathit{Int} \mapsto \mathit{Answer}\ (\mathit{nth}\ (\mathcal{N}[\![N_{index}]\!])\ i^*))$
$\textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{fi}$

$\mathcal{NE}[\![(A\ NE_1\ NE_2)]\!]$
$= (\mathit{with\text{-}int}\ (\mathcal{NE}[\![NE_1]\!]\ i^*)\ (\lambda i_1 .\ (\mathit{with\text{-}int}\ (\mathcal{NE}[\![NE_2]\!]\ i^*)\ (\lambda i_2 .\ (\mathcal{A}[\![A]\!]\ i_1\ i_2)))))$

$\mathcal{A}$ and $\mathcal{N}$ are unchanged from ELMM.

Figure 4.7: Denotational semantics for ELM.

$\mathcal{P}[\![$`(elm 2 (+ (arg 2) (* (arg 1) 3)))`$]\!]$ $[4,5]$
$=$ **if** $(length\ [4,5]) = \mathcal{N}[\![$`2`$]\!]$
  then $(\mathcal{NE}[\![$`(+ (arg 2) (* (arg 1) 3))`$]\!]\ [4,5])$
  else $(Error \mapsto Answer\ error)$
$= (\mathcal{NE}[\![$`(+ (arg 2) (* (arg 1) 3))`$]\!]\ [4,5])$
$=$ *with-int* $(\mathcal{NE}[\![$`(arg 2)`$]\!]\ [4,5])$
  $(\lambda i_1\ .$ *with-int* $(\mathcal{NE}[\![$`(* (arg 1) 3)`$]\!]\ [4,5])$
    $(\lambda i_2\ .\ (\mathcal{A}[\![$`+`$]\!]\ i_1\ i_2)))$
$=$ *with-int* (*with-int* $(\mathcal{NE}[\![$`(arg 1)`$]\!]\ [4,5])$
    $(\lambda i_3\ .$ *with-int* $(\mathcal{NE}[\![$`3`$]\!]\ [4,5])$
      $(\lambda i_4\ .\ (\mathcal{A}[\![$`*`$]\!]\ i_3\ i_4)))$
  $(\lambda i_2\ .\ (\mathcal{A}[\![$`+`$]\!]\ 5\ i_2))$
$=$ (*with-int* $(\mathcal{A}[\![$`*`$]\!]\ 4\ 3)\ (\lambda i_2\ .\ (\mathcal{A}[\![$`+`$]\!]\ 5\ i_2)))$
$= (\mathcal{A}[\![$`+`$]\!]\ 5\ 12)$
$(Int \mapsto Answer\ 17)$

Figure 4.8: Meaning of an ELM program applied to two arguments.

the two integer arguments in the case where the argument sequence i^* has two elements. We showed above that the result in this case is correct, and we know that an error is returned for any other length.

4.2.4 Step 4: EL

Full EL (Figure 2.4) is obtained from ELM by adding a numerical `if` expression and boolean expressions for controlling these expressions. Boolean expressions BE include the truth literals `true` and `false`, relational expressions like `(<` NE_1 NE_2`)`, and logical expressions like `(and` BE_1 BE_2`)`. Since boolean expressions can include numerical expressions as subexpressions and such subexpressions can denote errors, boolean expressions can also denote errors (e.g. `(< 1 (/ 2 0))`). In Figure 4.9, we model this by having the valuation function $\mathcal{BE}$ for boolean expressions return an element in the domain *BoolAnswer* of "boolean answers" that is distinct from the domain *Answer* of "integer answers". Since a numerical subexpression of a relational expression could be an `arg` expression, the meaning of a boolean expression is a function with signature $Int^* \rightarrow BoolAnswer$ that maps implicit program arguments to a boolean answer. The error handling for relational and logical operations is handled by $\mathcal{BE}$, so the $\mathcal{R}$ and $\mathcal{L}$ valuation functions manipulate only non-error values.

Note that the error-handling in $\mathcal{BE}[\![$`(`R_{rator} NE_1 NE_2`)`$]\!]$ is performed by

Semantic Domains

$i \in Int = \{\ldots\ ,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\}$
$b \in Bool = \{true, false\}$
$Error = \{error\}$
$a \in Answer = Int\ +\ Error$
$ba \in BoolAnswer = Bool\ +\ Error$

Valuation Functions

$\mathcal{P}$: Program $\rightarrow$ $Int^* \rightarrow Answer$

The $\mathcal{P}$ clause is unchanged from ELM (except the keyword `elm` becomes `el`).

$\mathcal{NE}$: NumExp $\rightarrow$ $Int^* \rightarrow Answer$

$\mathcal{NE}[\![\texttt{(if}\ BE_{test}\ NE_{then}\ NE_{else}\texttt{)}]\!]$
$= \lambda i^*\ .\ \textbf{matching}\ (\mathcal{BE}[\![BE_{test}]\!]\ i^*)$
$\triangleright\ (Bool \mapsto BoolAnswer\ b)\ [\![$
$\textbf{if}\ b\ \textbf{then}\ \mathcal{NE}[\![NE_{then}]\!]\ i^*\ \textbf{else}\ \mathcal{NE}[\![NE_{else}]\!]\ i^*\ \textbf{fi}$
$\triangleright\ \textbf{else}\ (Error \mapsto Answer\ error)\ \textbf{endmatching}$

The other $\mathcal{NE}$ clauses are unchanged from ELM.

$\mathcal{BE}$: BoolExp $\rightarrow$ $Int^* \rightarrow BoolAnswer$

$\mathcal{BE}[\![\texttt{true}]\!] = \lambda i^*\ .\ (Bool \mapsto BoolAnswer\ true)$

$\mathcal{BE}[\![\texttt{false}]\!] = \lambda i^*\ .\ (Bool \mapsto BoolAnswer\ false)$

$\mathcal{BE}[\![\texttt{(}R_{rator}\ NE_1\ NE_2\texttt{)}]\!]$
$= \lambda i^*\ .\ \textbf{matching}\ \langle \mathcal{NE}[\![NE_1]\!]\ i^*, \mathcal{NE}[\![NE_2]\!]\ i^* \rangle$
$\triangleright\ \langle (Int \mapsto Answer\ i_1), (Int \mapsto Answer\ i_2) \rangle\ [\![$
$(Bool \mapsto BoolAnswer\ (\mathcal{R}[\![R]\!]\ i_1\ i_2))$
$\triangleright\ \textbf{else}\ (Error \mapsto BoolAnswer\ error)\ \textbf{endmatching}$

$\mathcal{BE}[\![\texttt{(}L_{rator}\ BE_1\ BE_2\texttt{)}]\!]$
$= \lambda i^*\ .\ \textbf{matching}\ \langle \mathcal{BE}[\![BE_1]\!]\ i^*, \mathcal{BE}[\![BE_2]\!]\ i^* \rangle$
$\triangleright\ \langle (Bool \mapsto BoolAnswer\ b_1), (Bool \mapsto BoolAnswer\ b_2) \rangle\ [\![$
$(Bool \mapsto BoolAnswer\ (\mathcal{L}[\![L]\!]\ b_1\ b_2))$
$\triangleright\ \textbf{else}\ (Error \mapsto BoolAnswer\ error)\ \textbf{endmatching}$

$\mathcal{R}$: RelationalOperator $\rightarrow$ $(Int \rightarrow Int \rightarrow Bool)$

$\mathcal{R}[\![\texttt{<}]\!] = <_{Int}$
`=` and `>` are handled similarly.

$\mathcal{L}$: LogicalOperator $\rightarrow$ $(Bool \rightarrow Bool \rightarrow Bool)$

$\mathcal{L}[\![\texttt{and}]\!] = \lambda b_1\, b_2\ .\ (b_1\ \textbf{and}\ b_2)$
`or` is handled similarly.

$\mathcal{A}$ and $\mathcal{N}$ are unchanged from ELM.

Figure 4.9: Denotational semantics for EL.

pattern matching. Could it instead be done via *with-int*? No. The final return value of *with-int* is in *Answer*, but the final return value of $\mathcal{BE}$ is in *BoolAnswer*. However, we could define and use a new auxiliary function that is like *with-int* but returns an element of *BoolAnswer*(see Exercise 4.3).

Something that stands out in our study of the denotational semantics of the EL dialects is the importance of semantic domains and the signatures of valuation functions. Studying these gives insight into the fundamental nature of a language, even if the detailed valuation clause definitions are unavailable. For example, consider the signature of the numerical expression valuation function $\mathcal{NE}$ in the various dialects we studied. In ELMM without `/` and `%`, the signature

$$\mathcal{NE} : \text{NumExp} \rightarrow \mathit{Int}$$

indicates that expressions simply stands for an integer. In full ELMM, the "unwound" signature

$$\mathcal{NE} : \text{NumExp} \rightarrow (\mathit{Int} + \mathit{Error})$$

indicates that errors may be encountered in the evaluation of some expressions. The ELM signature

$$\mathcal{NE} : \text{NumExp} \rightarrow \mathit{Int}^* \rightarrow (\mathit{Int} + \mathit{Error})$$

has a context domain Int^* representing program arguments that are passed down the abstract syntax tree. We will see many kinds of contexts in our study of other languages. Some, like ELM program arguments, only flow down to subexpressions. We shall see later that other contexts can have more complex flows, and that these flows are reflected in the valuation function signatures.

4.2.5 A Denotational Semantics is Not a Program

You may have noticed that the denotational definitions for the dialects of EL strongly resemble programs in certain programming languages. In fact, it is straightforward to write an executable EL interpreter that reflects the structure of its valuation clauses, especially in functional programming languages like ML, HASKELL, and SCHEME. Of course, an interpreter has to be explicit about many of the details suppressed in the denotational definition (parsing the concrete syntax, choosing appropriate data structures to represent domain elements, etc.). Furthermore, details of the implementation language may complicate matters. In particular, the correspondence will be much less direct if the implementation programming language does not support first-class procedures.

Although a denotational definition often suggests an approach for implementing an interpreter program, it can be misleading to think of the denotational definition itself *as* a program. Programming language procedures typically imply computation; denotational specifications do not. An interpreter specifies a process for evaluating program phrases, often one with particular operational properties. In contrast, there is no notion of process associated with a valuation function: it is simply a declarative description for a mathematical function (i.e., a triple of a source, a target, and a graph).

For example, consider the following metalanguage expression, which might arise in the context of reasoning about an ELMM program:

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\textit{with-int}\ \ (\mathcal{A}[\![-]\!]\ \ 3\ \ 3)\ \ (\lambda i_2 \,.\, (\mathcal{A}[\![*]\!]\ \ i_1\ \ i_2))))\,.$$

If we (incorrectly) view this as an expression in a programming language like ML or SCHEME, we might think that no evaluation can take place until an integer is supplied for i_0, and, after that happens, the division must be performed first, followed by the subtraction, and finally the multiplication. But there is no inherent notion of evaluation order associated with the metalanguage expression. We can perform any mathematical simplifications in any order on this expression. For example, observing that $(\mathcal{A}[\![-]\!]\ 3\ 3)$ has the same meaning as $(\mathcal{N}[\![0]\!])$ allows us to rewrite the expression to

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\textit{with-int}\ \ (\mathcal{N}[\![0]\!])\ \ (\lambda i_2 \,.\, (\mathcal{A}[\![*]\!]\ \ i_1\ \ i_2))))\,.$$

This is equivalent to

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\mathcal{A}[\![*]\!]\ \ i_1\ \ 0))\,,$$

which is in turn equivalent to

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\textit{Int} \mapsto \textit{Answer}\ \ 0))\,,$$

since the product of 0 and any integer is 0. A division result cannot be an error when the second argument is non-zero, so this can be further simplified to:

$$\lambda i_0 \,.\, (\textit{Int} \mapsto \textit{Answer}\ \ 0).$$

The moral of this example is that many simplifications can be done with metalanguage expressions that would be difficult to justify with expressions in most programming languages.[2]

[2]Certain real-world programming languages, particularly the purely functional language HASKELL, were designed to support the kind of mathematical reasoning that can be done with metalanguage expressions.

Despite the above warning, sometimes it *is* useful to think of denotational descriptions as programs, if only for building intuitions about what they mean. This situation is reminiscent of the dy/dx notation in calculus, which teachers and textbooks commonly warn should never be viewed as a fraction. And yet, viewing it as a fraction has many advantages for understanding its meaning as well as for remembering formulae (the chain rule in particular). Similarly, viewing denotational definitions as programs can sometimes be helpful, especially for a beginner. To avoid misleading processing intuitions from familiar programming languages, you should view the lambda notation of the metalanguage as a typed, curried, normal-order programming language.

▷ **Exercise 4.1** We have treated integer numerals atomically, but we could express them in terms of their component signs and digits via an s-expression grammar:

SN ∈ SignedNumeral
UN ∈ UnsignedNumeral
D ∈ Digit
SN ::= `(+` *UN*`)` | `(-` *UN*`)` | *UN*
UN ::= *D* | `(@` *UN* *D*`)`
D ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

For example, the numeral traditionally written as `-273` would be written in s-expression form as `(- (@ (@ 2 7) 3))`. Give a denotational semantics for numerals by providing valuation functions for each of SignedNumeral, UnsignedNumeral, and Digit. ◁

▷ **Exercise 4.2** Use the ELM semantics to determine the meaning of the following ELM program: `(elm 2 (/ (arg 1) (- (arg 1) (arg 2))))`. ◁

▷ **Exercise 4.3** o By analogy with the *with-int* auxiliary function in the ELM semantics, define functions with the following signatures and use them to "hide" error-handling in the EL valuation clauses for conditional expressions, relational operations, and logical operations:

$$\textit{with-bool} : \textit{BoolAnswer} \to (\textit{Bool} \to \textit{Answer}) \to \textit{Answer}$$
$$\textit{with-int}_{BA} : \textit{Answer} \to (\textit{Int} \to \textit{BoolAnswer}) \to \textit{BoolAnswer}$$
$$\textit{with-bool}_{BA} : \textit{BoolAnswer} \to (\textit{Bool} \to \textit{BoolAnswer}) \to \textit{BoolAnswer}$$ ◁

4.3 A Denotational Semantics for PostFix

We are now ready to flesh out the details of the denotational description of PostFix that were sketched in Section 4.1. The abstract syntax for PostFix was already provided in Figure 2.8, so the syntactic algebra is already taken care of. We therefore need to construct the semantic algebra and the meaning function.

$$
\begin{array}{l}
t \in \mathit{StackTransform} = \mathit{Stack} \rightarrow \mathit{Stack} \\
s \in \mathit{Stack} = \mathit{Value}^* + \mathit{Error} \\
v \in \mathit{Value} = \mathit{Int} + \mathit{StackTransform} \\
r \in \mathit{Result} = \mathit{Value} + \mathit{Error} \\
a \in \mathit{Answer} = \mathit{Int} + \mathit{Error} \\
\quad \mathit{Error} = \{\mathit{error}\} \\
i \in \mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
b \in \mathit{Bool} = \{\mathit{true}, \mathit{false}\}
\end{array}
$$

Figure 4.10: Semantic domains for the PostFix denotational semantics.

4.3.1 A Semantic Algebra for PostFix

What kind of mathematical entities should we use to model PostFix programs? Suppose that we have some sort of entity representing stacks. Then it's natural to model both PostFix commands and command sequences as functions that transform one stack entity into another. For example, the `swap` command could be modeled by a function that takes a stack as an argument, and returns a stack in which the top two elements have been swapped.

We need to make some provision for the case where the stack contains an insufficient number of elements or the wrong type of elements. For this purpose we will assume that there is a distinguished stack, *errorStack*, that indicates that an error has occurred. For example, calling the transform associated with the `swap` command on a stack with fewer than two elements should return *errorStack*. All transforms should return *errorStack* when given *errorStack* as an argument.

Figure 4.10 presents domain equations that describe one implementation of this approach. The *StackTransform* domain consists of functions from stacks to stacks, where an element of the domain *Stack* is either a sequence of values or the distinguished error stack (here modeled by the single element of the unit domain *Error*). The domain *Value* of stackable values includes not only integers but also stack transforms, which model executable sequences that have been pushed on the stack. The *Result* domain models intermediate results obtained via stack manipulations or arithmetic operations. It includes an error result to model situations like popping an empty stack and dividing by zero. The *Answer* domain models the final outcome of a PostFix program. Like *Result*, *Answer* includes an error answer, but its only non-error answers are integers (because executable sequences at the top of a final stack cannot be observed and are treated as errors).

A somewhat unsettling property of the domain equations in the figure is that they are defined recursively — transforms operate on stacks, which themselves

may contain transforms. In Chapter 5 we will discuss how to understand a set of recursively defined equations. For now, we'll just assume that these equations have a sensible interpretation.

We extend the semantic domains into a semantic algebra by defining a collection of functions that manipulate the domains. Right now we'll just specify the interfaces to these functions. We'll defer the details of their definitions until after we've studied the meaning function. This will allow us to move more quickly to the core of the denotational semantics — the meaning function — without getting sidetracked by various issues concerning the definition of the semantic functions.

Figure 4.11 gives informal specifications for the functions that we will use to manipulate the semantic domains. We will defer studying the implementation of these functions until later. *errorResult*, *errorAnswer*, *errorStack*, and *errorTransform* are just names for useful constants involving errors. *push*, *pop*, and *top* are the usual stack operations. Their specifications are complicated somewhat by the details of error handling. For example, *top* returns an element of *Result* rather than just *Value* because it must return *errorResult* in the case where the given stack is empty. *push* takes its argument from *Result* rather than *Value* so that it can be composed with *top*. *intAt* is an auxiliary function that simplifies the specification of `nget`. *arithop* simplifies the specifications for arithmetic and relational commands; it serves to abstract over common behavior (replacing the top two integers on the stack by some value that depends on them) while suppressing error detail (return an error stack if any error is encountered along the way). *transform* facilitates error handling when a result that is expected to be a transform turns out to be an integer or an error result instead. *resToAns* handles the conversion from results to answers.

The signatures of the functions, especially the stack functions, may seem strange at first glance, because few of them explicitly refer to the *Stack* domain. But recall that *StackTransform* is defined to be $Stack \to Stack$, so that the signature of *push*, for instance, is really

$$Result \to (Stack \to Stack)\,.$$

From this perspective, *push* probably seems more familiar: it is a function that takes an result and stack (in curried form) and returns a stack. However, since stack transforms are the key abstraction of this semantics, we have written the signatures to emphasize this fact. Under this view, *push* is a function that takes a result and returns a stack transform. Of course, in either case *push* is exactly the same mathematical entity; the only difference is in how we think about it!

- *errorResult* : *Result*
  An error in the domain *Result*.
- *errorAnswer* : *Answer*
  An error in the domain *Answer*.
- *errorStack* : *Stack*
  The distinguished error stack.
- *errorTransform* : *StackTransform*
  A transform that maps all stacks to *errorStack*.
- *push* : *Result* → *StackTransform*
  Given the result value v, return a transform that pushes v on a stack; otherwise return *errorTransform*.
- *pop* : *StackTransform*
  For a nonempty stack s, return the stack resulting from popping the top value; otherwise return *errorStack*.
- *top* : *Stack* → *Result*
  Given a nonempty stack s, return result that is the top element of s; otherwise return *errorResult*.
- *intAt* : *Int* → *Stack* → *Result*
  Given an integer i_{index} and a stack whose i_{index}th element (starting from 1) is the integer i_{result}, return i_{result}; otherwise return *errorResult*.
- *arithop* : (*Int* → *Int* → *Result*) → *StackTransform*
  Let f : *Int* → *Int* → *Result* be the functional argument to *arithop*. Return a transform with the following behavior: if the given stack has two integers i_1 and i_2 followed by s_{rest}, then return a stack whose top value v_{result} is followed by s_{rest}, where (*Value* ↦ *Result* v_{result}) is the result of the application (f i_2 i_1). If the given stack is not of this form or if the result of applying f is *errorResult*, then return *errorStack*.
- *transform* : *Result* → *StackTransform*
  Given a result that is a stack transform, return it; otherwise return *errorTransform*.
- *resToAns* : *Result* → *Answer*
  Given a result that is an integer, return it as an answer; otherwise return *errorAnswer*.

Figure 4.11: Specifications for functions on PostFix semantic domains.

$\mathcal{P}$: Program $\rightarrow$ *Int** $\rightarrow$ *Answer*
$\mathcal{Q}$: Commands $\rightarrow$ *StackTransform*
$\mathcal{C}$: Command $\rightarrow$ *StackTransform*
$\mathcal{A}$: ArithmeticOperator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Result*)
$\mathcal{R}$: RelationalOperator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Bool*)
$\mathcal{N}$: Intlit $\rightarrow$ *Int*

Figure 4.12: Signatures of the PostFix valuation functions.

4.3.2 A Meaning Function for PostFix

Now we're ready to study the meaning function for PostFix. As in EL, we specify the meaning function by a collection of valuation functions, one for each syntactic domain defined by the abstract syntax for the language.

As we learned in studying the denotational semantics of EL, the signatures of valuation functions contain valuable information about the meaning of the language. It is always prudent to study the signatures before delving into the details of the definitions for the valuation functions.

The signatures for the PostFix valuation functions appear in Figure 4.12. In the case of PostFix, one of the things the signatures say is that a PostFix program is like an EL program: it takes a sequence of integers as arguments and either returns an integer or signals an error. If the signature of $\mathcal{P}$ were instead

$$\mathcal{P} : \text{Program} \rightarrow \mathit{Int}^* \rightarrow \mathit{Result},$$

it would indicate that some PostFix programs could return a stack transform (i.e., an executable sequence) instead of an integer. If the signature were one of

$$\mathcal{P} : \text{Program} \rightarrow \mathit{Int}^* \rightarrow \mathit{Int} \quad \text{or} \quad \mathcal{P} : \text{Program} \rightarrow \mathit{Int}^* \rightarrow \mathit{Value},$$

it would tell use that errors could not be signaled by a PostFix program.

The signatures also tell us that both commands and command sequences map to stack transforms. Since stack transforms are easily composable, this suggests that the meaning of a command sequence will be some sort of composition of the meanings of its component commands. This turns out to be the case. The return type of $\mathcal{A}$ matches the argument type of *arithop*, one of the auxiliary functions specified in Figure 4.11. This is more than coincidence; the auxiliary functions and valuation functions were designed to dovetail in a nice way.

Now we're ready to study the definitions of the PostFix valuation functions, which appear in Figure 4.13. The meaning of a program (`postfix` $N_{numargs}$ Q) is a function that transforms an initial stack consisting of the integers in the argument sequence i^* via the transform $\mathcal{Q}[\![Q]\!]$ and returns the top integer of the

$\mathcal{P}[\![(\texttt{postfix}\ N_{numargs}\ Q)]\!]$
$= \lambda i^* .\ \textbf{if}\ (length\ i^*) = \mathcal{N}[\![N_{numargs}]\!]$
$\textbf{then}\ (resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (Value^* \mapsto Stack\ (map\ Int \mapsto Value\ i^*)))))$
$\textbf{else}\ errorAnswer\ \textbf{fi}$

$\mathcal{Q}[\![C\ .\ Q]\!] = \mathcal{Q}[\![Q]\!] \circ \mathcal{C}[\![C]\!]$
$\mathcal{Q}[\![]\!] = \lambda s .\ s$

$\mathcal{C}[\![N]\!] = (push\ (Value \mapsto Result\ (Int \mapsto Value\ (\mathcal{N}[\![N]\!]))))$
$\mathcal{C}[\![(\,Q)]\!] = (push\ (Value \mapsto Result\ (StackTransform \mapsto Value\ \mathcal{Q}[\![Q]\!])))$
$\mathcal{C}[\![\texttt{pop}]\!] = pop$
$\mathcal{C}[\![\texttt{swap}]\!] = \lambda s .\ (push\ (top\ (pop\ s))\ (push\ (top\ s)\ (pop\ (pop\ s))))$
$\mathcal{C}[\![\texttt{nget}]\!] = \lambda s .\ \textbf{matching}\ (top\ s)$
$\triangleright\ (Value \mapsto Result\ (Int \mapsto Value\ i))\ [\![\ (push\ (intAt\ i\ (pop\ s))\ (pop\ s))$
$\triangleright\ \textbf{else}\ errorStack\quad \textbf{endmatching}$
$\mathcal{C}[\![\texttt{sel}]\!] = \lambda s .\ \textbf{matching}\ (top\ (pop\ (pop\ s)))$
$\triangleright\ (Value \mapsto Result\ (Int \mapsto Value\ i))\ [\![$
$(push\ (\textbf{if}\ (i =_{Int} 0)\ \textbf{then}\ (top\ s)\ \textbf{else}\ (top\ (pop\ s))\ \textbf{fi})$
$(pop\ (pop\ (pop\ s)))\ )$
$\triangleright\ \textbf{else}\ errorStack\quad \textbf{endmatching}$
$\mathcal{C}[\![\texttt{exec}]\!] = \lambda s .\ (transform\ (top\ s)\ (pop\ s))$
$\mathcal{C}[\![A]\!] = (arithop\ \mathcal{A}[\![A]\!])$
$\mathcal{C}[\![R]\!] = (arithop\ (\lambda i_1\, i_2 .\ (Value \mapsto Result$
$(Int \mapsto Value\ (\textbf{if}\ (\mathcal{R}[\![R]\!]\ i_1\ i_2)\ \textbf{then}\ 1\ \textbf{else}\ 0\ \textbf{fi})))))$

$\mathcal{A}[\![\texttt{sub}]\!] = \lambda i_1\, i_2 .\ (Value \mapsto Result\ (Int \mapsto Value\ (i_1 -_{Int} i_2)))$
Similarly for `add`, `mul`
$\mathcal{A}[\![\texttt{div}]\!] = \lambda i_1\, i_2 .\ \textbf{if}\ (i_2 =_{Int} 0)\ \textbf{then}\ errorResult$
$\textbf{else}\ (Value \mapsto Result\ (Int \mapsto Value\ (i_1 /_{Int} i_2)))\ \textbf{fi}$

$\mathcal{R}[\![\texttt{lt}]\!] = <_{Int}$
Similarly for `eq` and `gt`

$\mathcal{N}$ maps integer numerals to the integer numbers they denote.

Figure 4.13: Valuation functions for POSTFIX.

resulting stack. The definitions of *resToAns* and *top* guarantee that an error answer is returned when the stack is empty or does not have an integer as its top element. An error is also signaled when the number of arguments does not match the expected number $N_{numargs}$.

The meaning of a command sequence is the composition of the transforms of its component commands. The order of the composition

$$\mathcal{Q}[\![Q]\!] \circ \mathcal{C}[\![C]\!] = \lambda s\,.\ (\mathcal{Q}[\![Q]\!]\ \ (\mathcal{C}[\![C]\!]\ \ s))$$

is crucial, because it guarantees that the stack manipulations of the first command can be observed by the subsequent commands. Reversing the order of the composition would have the effect of executing commands in a right-to-left order instead. The stack transform associated with the empty command sequence is the identity function on stacks.

Most of the clauses for the command valuation function $\mathcal{C}$ are straightforward. The integers and transforms corresponding to numerals and executable sequences are simply pushed onto the stack after appropriate injections into the *Value* and *Result* domains.[3] The transform associated with the `pop` command is simply the *pop* auxiliary function, while the transform associated with `swap` is expressed as a composition of *push*, *top*, and *pop*. If the top stack element is an integer i, the `nget` transform replaces it by the ith element from the rest of the stack if that element is an integer; in all other cases, `nget` returns an error stack. The `sel` transform selects one of the top two stack elements based on the numeric value of the third stack element; an error is signaled if the third element is not an integer. In the `exec` transform, the top stack element is expected to be a stack transform t representing an executable sequence. Applying t to the rest of the stack yields the stack resulting from executing the executable sequence. If the top stack element is not a stack transform, an error is signaled. The meaning of arithmetic and relational commands is determined by *arithop* in conjunction with $\mathcal{A}$ and $\mathcal{R}$, valuation functions that map operator symbols like `add` and `lt` to the expected functions and predicates. $\mathcal{A}$ treats `div` specially so that division by 0 signals an error.

Before we move on, a few notes about reading the PostFix denotational definitions are in order. Valuation functions tend to be remarkably elegant and concise. But this does not mean that they are always easy to read! To the contrary, the density of information in a denotational definition often demands

[3] Whereas the operational semantics used a stack with syntactic values — integer numerals and command sequences — the denotational semantics uses a stack of semantic values — integers and stack transforms. This is because the valuation functions $\mathcal{N}$ and $\mathcal{Q}$ are readily available for translating the syntactic elements to the semantic ones. Here and elsewhere, we will follow the convention of using explicit injections in denotational descriptions.

meticulous attention from the reader. The ability to read semantic functions and valuation functions is a skill that requires patient practice to acquire. At first, unraveling such a definition may seem like solving a puzzle or doing detective work. However, the time invested in reading definitions of this sort pays off handsomely in terms of deep insights into the meanings of programming languages.

The conciseness of a denotational definition is due in large part to the liberal use of higher-order functions, i.e., functions that take other functions as arguments or return them as results. *arithop* is an excellent example of such a function: it takes an argument in the function domain $Int \rightarrow Int \rightarrow Result$, and returns a stack transform, which itself is an element of the function domain $Stack \rightarrow Stack$.

Definitions involving higher-order functions can be rather daunting to read until you acquire a knack for them. A typical problem is to think that pieces are missing. For example, a common reaction to the valuation clause for numerals,

$$\mathcal{C}[\![N]\!] = (push\ (Value \mapsto Result\ (Int \mapsto Value\ (\mathcal{N}[\![N]\!])))),$$

is that a stack is somehow missing. After all, the value has to be pushed onto *something* — where is it? Carefully considering types, however, will show that nothing is missing. Recall that the signature of *push* is $Result \rightarrow StackTransform$. Since

$$(Value \mapsto Result\ (Int \mapsto Value\ \mathcal{N}[\![N]\!]))$$

is clearly an element of *Result*, the result of the *push* application is a stack transform. Since $\mathcal{C}$ is supposed to map commands to stack transforms, the definition is well-typed. It's possible to introduce an explicit stack in this valuation clause by wrapping the right hand side in a λ of a stack argument:

$$\mathcal{C}[\![N]\!] = \lambda s\,.\,(push\ (Value \mapsto Result\ (Int \mapsto Value\ \mathcal{N}[\![N]\!]))\ s).$$

This form of the definition probably seems much more familiar, because it's more apparent that the meaning of the command is a function that takes a stack and returns a stack, and *push* is actually given a stack on which to push its value. But the two definitions are equivalent. In order to stress the power of higher-order functions, we will continue to use the more concise versions. We encourage you to type check the definitions and expand them with extra λs as ways of improving your skill at reading them.

Figure 4.14 illustrates using the PostFix denotational semantics to determine the result of applying the program `(postfix 2 3 sub swap pop)` to the argument integers $[7, 8]$. To make the figure more concise, we use the shorthand

Note: $\hat{n}$ is a shorthand for $(Int \mapsto Value\ n)$

$\mathcal{P}[\![$`(postfix 2 3 sub swap pop)`$]\!]\ [7,8]$

$= \textbf{if}\ (length\ [7,8]) = \mathcal{N}[\![2]\!]$
$\quad \textbf{then}\ resToAns\ \big(top\ \big(\mathcal{Q}[\![$`3 sub swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)$
$\quad \textbf{else}\ errorAnswer\ \textbf{fi}$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`3 sub swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![$`sub swap pop`$]\!]) \circ (\mathcal{C}[\![$`3`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`sub swap pop`$]\!]\ \big(\mathcal{C}[\![$`3`$]\!]\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)\big)$

$= resToAns\ (top\ ((\mathcal{Q}[\![$`sub swap pop`$]\!])\ (push\ (Value \mapsto Result\ \hat{3})$
$\qquad (Value^* \mapsto Stack\ [\hat{7},\hat{8}]))))$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`sub swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![$`swap pop`$]\!]) \circ (\mathcal{C}[\![$`sub`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`swap pop`$]\!]\ \big(\mathcal{C}[\![$`sub`$]\!]\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`swap pop`$]\!]\ \big(arithop\ (\mathcal{A}[\![$`sub`$]\!])\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)\big)$

$= resToAns\ (top\ (\mathcal{Q}[\![$`swap pop`$]\!]\ (push\ (Value \mapsto Result\ (7\ \widehat{-_{Int}}\ 3))$
$\qquad (Value^* \mapsto Stack\ [\hat{8}]))))$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![$`pop`$]\!]) \circ (\mathcal{C}[\![$`swap`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`pop`$]\!]\ \big(\mathcal{C}[\![$`swap`$]\!]\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)\big)$

$= resToAns\ (top\ (\mathcal{Q}[\![$`pop`$]\!]\ (push\ \big(top\ \big(pop\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)$
$\qquad (push\ \big(top\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)$
$\qquad\quad \big(pop\ \big(pop\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)\big)\big)\big)\big)\big)$

$= resToAns\ (top\ (\mathcal{Q}[\![$`pop`$]\!]\ (push\ \big(top\ (Value^* \mapsto Stack\ [\hat{8}])\big)$
$\qquad (push\ (Value \mapsto Result\ \hat{4})$
$\qquad\quad (Value^* \mapsto Stack\ [\,])\big)\big)\big)\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`pop`$]\!]\ \big(push\ (Value \mapsto Result\ \hat{8})\ (Value^* \mapsto Stack\ [\hat{4}])\big)\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{8},\hat{4}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![]\!]) \circ (\mathcal{C}[\![$`pop`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{8},\hat{4}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![]\!]\ \big(\mathcal{C}[\![$`pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{8},\hat{4}])\big)\big)\big)$

$= resToAns\ \big(top\ \big((\lambda s\,.\,s)\ (Value^* \mapsto Stack\ [\hat{4}])\big)\big)$

$= resToAns\ \big(top\ (Value^* \mapsto Stack\ [\hat{4}])\big)$

$= resToAns\ (Value \mapsto Result\ \hat{4})$

$= (Int \mapsto Answer\ 4)$

Figure 4.14: Equational proof that applying the POSTFIX program `(postfix 2 3 sub swap pop)` to the arguments $[7,8]$ yields the answer 4.

$\hat{n}$ to stand for $(\mathit{Int} \mapsto \mathit{Value}\ n)$. Each line of the equational proof is justified by simple mathematical reasoning. For example, the equality

$$\begin{aligned}&\mathit{resToAns}\ \big(\mathit{top}\ \big(((\mathcal{Q}[\![\texttt{pop}]\!])\circ(\mathcal{C}[\![\texttt{swap}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big)\\ =\ &\mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{pop}]\!]\ \big(\mathcal{C}[\![\texttt{swap}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big)\big)\end{aligned}$$

is justified by the definition of function composition, while the equality

$$\begin{aligned}&\mathit{resToAns}\ (\mathit{top}\ (\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ \widehat{(7 -_{\mathit{Int}} 3)})\\ &\qquad\qquad\qquad\qquad (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{8}])))))\\ =\ &\mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big)\end{aligned}$$

is justified by the definition of $-_{\mathit{Int}}$ and the specification for the *push* function. The proof shows that the result of the program execution is the integer 4.

Just as programs can be simplified by introducing procedural abstractions, equational proofs can often be simplified by structuring them more hierarchically. In the case of proofs, the analog of a programming language procedure is a theorem. For example, it's not difficult to prove a theorem stating that for any numeral N, any command sequence Q, and any stack s, the following equality is valid:

$$\begin{aligned}&(\mathcal{Q}[\![N\ .\ Q]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*))\\ =&(\mathcal{Q}[\![Q]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ ((\mathit{Int} \mapsto \mathit{Value}\ \mathcal{N}[\![N]\!])\ .\ v^*))).\end{aligned}$$

This theorem is analogous to the operational rewrite rule for handling integer numeral commands. It can be used to justify equalities like

$$\begin{aligned}&\big(\mathcal{Q}[\![\texttt{3 sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{7},\hat{8}])\big)\\ =\ &\big(\mathcal{Q}[\![\texttt{sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3},\hat{7},\hat{8}])\big)\end{aligned}$$

A few such theorems can greatly reduce the length of the sample proof. In fact, if we prove other theorems analogous to the operational rules, we can obtain a proof whose structure closely corresponds to the configuration sequence for an operational execution of the program (see Figure 4.15).

Figure 4.16 shows how the equational proof in Figure 4.15 can be generalized two handle two arbitrary integer arguments. Based on this result, we conclude that the meaning of the POSTFIX program `(postfix 2 3 sub swap pop)` is:

$$\begin{aligned}&\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!]\\ =\ &\lambda i^*\ .\ \textbf{matching}\ i^*\\ &\qquad \triangleright\ [i_1, i_2]\ [\!]\ (\mathit{Int} \mapsto \mathit{Answer}\ (i_1 -_{\mathit{Int}} 3))\\ &\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\ .\end{aligned}$$

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!]\ [7,8] \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{3 sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{7},\hat{8}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3},\hat{7},\hat{8}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{8},\hat{4}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4}])\big) \\
= \mathit{resToAns}\ (\mathit{Value} \mapsto \mathit{Result}\ \hat{4}) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ 4)
\end{array}
$$

Figure 4.15: Alternative equational proof with an operational flavor.

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!]\ [i_1, i_2] \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{3 sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3}, \hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{i_2}, \widehat{(i_1\ -_{\mathit{Int}}\ 3)}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}])\Big) \\
= \mathit{resToAns}\ (\mathit{Value} \mapsto \mathit{Result}\ \widehat{(i_1\ -_{\mathit{Int}}\ 3)}) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ (i_1\ -_{\mathit{Int}}\ 3))
\end{array}
$$

Figure 4.16: Version of equational proof for two arbitrary integer arguments.

▷ **Exercise 4.4** Use the PostFix denotational semantics to determine the values of the PostFix programs in Exercise 1.1. ◁

▷ **Exercise 4.5** Modify the PostFix denotational semantics to handle PostFix2. Include valuation functions for `:`, `(skip)`, and `(exec)`. ◁

▷ **Exercise 4.6** For each of the following, modify the PostFix denotational semantics to handle the specified extensions:

a. The `pair`, `left`, and `right` commands from Exercise 3.36.

b. The `for` and `repeat` commands from Exercise 3.39.

c. The *I*, `def`, and `get` commands from Exercise 3.43. ◁

4.3.3 Semantic Functions for PostFix: the Details

Now that we've studied the core of the PostFix semantics, we'll flesh out the details of the functions specified in Figure 4.11. Figure 4.17 presents one implementation of the specifications. As an exercise, you should make sure that these definitions type check, and that they satisfy the specifications in Figure 4.11.

Notice that several functions in Figure 4.17 describe similar manipulations. *push*, *pop*, and *arithop* all check to see if their input stack is an error stack. If so, they return *errorStack*; if not, they perform some manipulation on the sequence of values in the stack. We can abstract over these similarities by introducing three abstractions (Figure 4.18) similar to the *with-int* error hiding function defined in the EL denotational semantics:

- *with-stack-values* takes a function f from value sequences to stacks and returns a stack transform that (1) maps a non-error stack to the result of applying f to the value sequence in the stack, and (2) maps an error stack to an error stack.

- *with-val&stack* takes a function f from a value to a stack transform and returns a stack transform that (1) maps any stack whose value sequence consists of the value v followed by v_{rest}* to the result of applying f to v and the stack whose values are v_{rest}*, and (2) maps any stack not of this form to the error stack.

- *with-int&stack* takes a function f from an integer to a stack transform and returns a stack transform that (1) maps any stack whose value sequence consists of an integer i followed by v_{rest}* to the result of applying f to v

$$
\begin{array}{l}
\mathit{empty\text{-}stack} : \mathit{Stack} \ = (\mathit{Value}^* \mapsto \mathit{Stack}\ [\,]_{\mathit{Value}}) \\
\mathit{errorStack} : \mathit{Stack} \ = (\mathit{Error} \mapsto \mathit{Stack}\ \mathit{error}) \\
\mathit{errorTransform} : \mathit{StackTransform} \ = \lambda s\,.\,\mathit{errorStack} \\
\mathit{errorResult} : \mathit{Result} \ = (\mathit{Error} \mapsto \mathit{Result}\ \mathit{error}) \\
\mathit{errorAnswer} : \mathit{Answer} \ = (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error}) \\
\\
\mathit{push} : \mathit{Result} \to \mathit{StackTransform} \\
= \lambda rs\,.\ \textbf{matching}\ \langle r, s\rangle \\
\qquad \triangleright \langle (\mathit{Value} \mapsto \mathit{Result}\ v), (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*)\rangle \ [\!]\ (v\ .\ v^*) \\
\qquad \triangleright (\mathit{Error} \mapsto \mathit{Result}\ \mathit{error}) \ [\!]\ \mathit{errorStack}\ \ \textbf{endmatching} \\
\\
\mathit{pop} : \mathit{StackTransform} \\
= \lambda s\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ (v_{\mathit{head}}\ .\ v_{\mathit{tail}}{}^*)) \ [\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ v_{\mathit{tail}}{}^*) \\
\qquad \triangleright \textbf{else}\ \mathit{errorStack}\ \ \textbf{endmatching} \\
\\
\mathit{top} : \mathit{Stack} \to \mathit{Result} \\
= \lambda s\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ (v_{\mathit{head}}\ .\ v_{\mathit{tail}}{}^*)) \ [\!]\ (\mathit{Value} \mapsto \mathit{Result}\ v_{\mathit{head}}) \\
\qquad \triangleright \textbf{else}\ \mathit{errorResult}\ \ \textbf{endmatching} \\
\\
\mathit{intAt} : \mathit{Int} \to \mathit{Stack} \to \mathit{Result} \\
= \lambda is\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*) \ [\!] \\
\qquad\ \ \textbf{if}\ (1 \leq_{\mathit{Int}} i_{\mathit{index}})\ \ \textbf{and}\ (i_{\mathit{index}} \leq_{\mathit{Int}} (\mathit{length}\ v^*)) \\
\qquad\ \ \textbf{then matching}\ (\mathit{nth}\ i\ v^*) \\
\qquad\qquad \triangleright (\mathit{Int} \mapsto \mathit{Value}\ i_{\mathit{result}}) \ [\!]\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ i_{\mathit{result}})) \\
\qquad\qquad \triangleright \textbf{else}\ \mathit{errorResult} \\
\qquad\ \ \textbf{else}\ \mathit{errorResult}\ \textbf{fi} \\
\qquad \triangleright \textbf{else}\ \mathit{errorResult}\ \ \textbf{endmatching} \\
\\
\mathit{arithop} : (\mathit{Int} \to \mathit{Int} \to \mathit{Result}) \to \mathit{StackTransform} \\
= \lambda f\, s\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ ((\mathit{Int} \mapsto \mathit{Value}\ i_1)\ .\ (\mathit{Int} \mapsto \mathit{Value}\ i_2)\ .\ v_{\mathit{rest}}{}^*)) \ [\!] \\
\qquad\ \ (\mathit{push}\ \ (f\ \ i_2\ \ i_1)\ \ v_{\mathit{rest}}{}^*) \\
\qquad \triangleright \textbf{else}\ \mathit{errorStack}\ \ \textbf{endmatching} \\
\\
\mathit{transform} : \mathit{Result} \to \mathit{StackTransform} \\
= \lambda r\,.\ \textbf{matching}\ r \\
\qquad \triangleright (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{StackTransform} \mapsto \mathit{Value}\ t)) \ [\!]\ t \\
\qquad \triangleright \textbf{else}\ \mathit{errorTransform}\ \ \textbf{endmatching} \\
\\
\mathit{resToAns} : \mathit{Result} \to \mathit{Answer} \\
= \lambda r\,.\ \textbf{matching}\ r \\
\qquad \triangleright (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ i)) \ [\!]\ (\mathit{Int} \mapsto \mathit{Answer}\ i) \\
\qquad \triangleright \textbf{else}\ \mathit{errorAnswer}\ \ \textbf{endmatching}
\end{array}
$$

Figure 4.17: Functions manipulating the semantic domains for PostFix.

```
with-stack-values : (Value* → Stack) → StackTransform
= λf s . matching s
        ▷ (Value* ↦ Stack v*) ‖ (f v*)
        ▷ else errorStack endmatching

with-val&stack : (Value → StackTransform) → StackTransform
= λf . (with-stack-values
          (λv* . matching v*
                  ▷ v1 . v_rest* ‖ (f v1 (Value* ↦ Stack v_rest*))
                  ▷ else errorStack endmatching ))

with-int&stack : (Int → StackTransform) → StackTransform
= λf . (with-val&stack
          (λv . matching v
                  ▷ (Int ↦ Value i) ‖ (f i)
                  ▷ else errorTransform endmatching ))

push : Result → StackTransform
= λr . matching r
       ▷ (Value ↦ Result v) ‖ (with-stack-values (λv* . (Value* ↦ Stack (v . v*))))
       ▷ else errorTransform
       endmatching

pop : StackTransform = with-val&stack (λv_hd s_tl . s_tl)

arithop : (Int → Int → Result) → StackTransform
= λf . (with-int&stack (λi1 . (with-int&stack (λi2 . (push (f i2 i1)))))))
```

Figure 4.18: The auxiliary functions *with-stack-values*, *with-val&stack*, and *with-integer&stack* simplify some of the semantic functions for POSTFIX. (Only the modified functions are shown.)

and the stack whose values are v_{rest}*, and (2) maps any stack not of this form to the error stack.

The purpose of these new functions is to hide the details of error handling in order to highlight more important manipulations. As shown in Figure 4.18, rewriting *push* in terms of *with-stack-values* removes an error check from the definition. Using *with-val&stack* and *with-int&stack* greatly simplify *pop* and *arithop*; the updated versions concisely capture the essence of these functions without the distraction of case analyses and error checks.

As with the valuation functions, these highly condensed semantic functions can be challenging for the uninitiated to read. The fact that *push*, *pop*, and *arithop* are ultimately manipulating a stack is even harder to see in the new versions than it was in the original ones. As suggested before, reasoning about types and inserting extra λs can help. For example, since the result of a call to *with-int&stack* is a stack transform t, and t is equivalent to $\lambda s\,.\,(t\ s)$, the new version of *arithop* can be rewritten as:

$$\begin{array}{l}\lambda f\, s_0\ .\ ((\textit{with-int\&stack}\\ \qquad (\lambda i_1\, s_1\ .\ ((\textit{with-int\&stack}\\ \qquad\qquad (\lambda i_2\, s_2\ .\ (\textit{push}\ \ (f\ \ i_2\ \ i_1)\ \ s_2)))\\ \qquad\qquad s_1)))\\ \qquad s_0).\end{array}$$

At least in this form it's easier to see that there are stacks from which each occurrence of *with-int&stack* can extract an integer and substack.

Even more important is recognizing the pattern

$$((\textit{with-int\&stack}\ \ (\lambda i s_{rest}\ .\ E))\ \ s)$$

as a construct that binds names to values. This pattern can be pronounced as:

> "Let i be the top value of s and s_{rest} be all but the top value of s in the expression E. Return the value of E, except when s is empty or its top value isn't an integer, in which cases the error stack should be returned instead."

Some of the PostFix valuation functions can be re-expressed using the error hiding functions directly. For example, the valuation clause for `swap` can be written as:

$$\mathcal{C}[\![\texttt{swap}]\!] = \textit{with-val\&stack}\ \ (\lambda v_1\ .\ (\textit{with-val\&stack}\ \ (\lambda v_2\ .(\textit{push}\ \ v_2)\ \circ\ (\textit{push}\ \ v_1))))$$

You should convince yourself that this has the same meaning as the version written using *push*, *top*, and *pop*.

▷ **Exercise 4.7**

a. By analogy with *with-int&stack*, define a function *with-trans&stack* whose signature is $(StackTransform \to StackTransform) \to StackTransform$.

b. Rewrite the valuation clauses for the commands `nget`, `sel`, and `exec` using *with-val&stack*, *with-int&stack*, and *with-trans&stack* to eliminate all occurrences of *top*, *pop*, *transform*, and **matching**. ◁

4.4 Denotational Reasoning

The denotational definitions of EL and PostFix presented in the previous section are mathematically elegant, but how useful are they? We have already shown how they can be used to determine the meanings of particular programs. In this section we show how denotational semantics helps us to reason about program equality and safe program transformations. The compositional structure of the denotational semantics makes it more amenable to proving certain properties than the operational semantics. We also study the relationship between operational semantics and denotational semantics.

4.4.1 Program Equality

Above, we studied the PostFix program `(postfix 2 3 sub swap pop)`, which takes two integer arguments and returns three less than the first argument:

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!] \\
= \lambda i^* .\ \textbf{matching}\ i^* \\
\qquad \triangleright\ [i_1, i_2]\ [\![\ \ (Int \mapsto Answer\ (i_1 -_{Int} 3)) \\
\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\,.
\end{array}
$$

Intuitively, the purpose of the `swap pop` is to get rid of the second argument, which is ignored by the program. But in a PostFix program, only the integer at the top of the final stack can be observed and any other stack values are ignored. So we should be able to remove the `swap pop` from the program without changing its behavior.

We can formalize this reasoning using denotational semantics. Figure 4.19 shows a derivation of the meaning of the program `(postfix 2 3 sub)` when it is applied to two arguments. From this, we deduce that the meaning of `(postfix 2 3 sub)` is:

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub)}]\!] \\
= \lambda i^* .\ \textbf{matching}\ i^* \\
\qquad \triangleright\ [i_1, i_2]\ [\![\ \ (Int \mapsto Answer\ (i_1 -_{Int} 3)) \\
\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\,.
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub)}]\!]\ [i_1, i_2] \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{3 sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3}, \hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}, \hat{i_2}])\Big) \\
= \mathit{resToAns}\ (\mathit{Value} \mapsto \mathit{Result}\ \widehat{(i_1\ -_{\mathit{Int}}\ 3)}) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ (i_1\ -_{\mathit{Int}}\ 3))
\end{array}
$$

Figure 4.19: The meaning of `(postfix 2 3 sub)` on two arguments.

Since `(postfix 2 3 sub)` and `(postfix 2 3 sub swap pop)` have exactly the same meaning, they cannot be distinguished as programs.

Denotational semantics can also be used to show that programs from different languages have the same meaning. For example, it is not hard to show that the meaning of the EL program `(el 2 (- (arg 1) 3))` is:

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(el 2 (- (arg 1) 3)}]\!] \\
= \lambda i^*.\ \textbf{matching}\ i^* \\
\qquad \triangleright\ [i_1, i_2]\ ]\!]\ \ (\mathit{Int} \mapsto \mathit{Answer}\ (i_1\ -_{\mathit{Int}}\ 3)) \\
\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\,.
\end{array}
$$

If you review the semantic domains for EL and PostFix, you will see that the *Answer* domain is the same for both languages. So the above fact means that this EL program is interchangeable with the two PostFix programs whose meanings are given above.

4.4.2 Safe Transformations: A Denotational Approach

Because denotational semantics is compositional, it is a natural tool for proving that it is safe to replace one phrase by another. Recall the following three facts from the operational semantics of PostFix:

1. Two PostFix command sequences are observationally equivalent if they behave indistinguishably in all program contexts.

2. Two PostFix command sequences are transform equivalent if they map equivalent stacks to equivalent stacks.

3. Transform equivalence implies observational equivalence.

Since the PostFix denotational semantics models command sequences as stack transforms, the denotational equivalence of PostFix command sequences corresponds to transform equivalence in the observational framework. So we expect the following theorem:

> PostFix **Denotational Equivalence Theorem:**
> $\mathcal{Q}[\![Q_1]\!] = \mathcal{Q}[\![Q_2]\!]$ implies $Q_1 =_{obs} Q_2$.

This theorem is a consequence of a so-called adequacy property of PostFix, which we will study later in Section 4.4.4.2.

We can use this theorem to help us prove the behavioral equivalence of two command sequences. For instance, consider the pair of command sequences $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}]$ and $[\texttt{3}, \texttt{add}]$. Figure 4.20 shows that these are denotationally equivalent, so, by the above theorem, they must be observationally equivalent. The equational reasoning in Figure 4.20 uses the following three equalities, whose proofs are left as exercises:

$$(\mathcal{Q}[\![C_1 \;\; C_2 \;\; \ldots C_n]\!]) = (\mathcal{C}[\![C_n]\!]) \circ \ldots \circ (\mathcal{C}[\![C_2]\!]) \circ (\mathcal{C}[\![C_1]\!]) \tag{4.1}$$

$$(\mathit{with\text{-}int\&stack}\ f) \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ i))) = (f\ \ i) \tag{4.2}$$

$$t \circ (\mathit{with\text{-}int\&stack}\ f) = (\mathit{with\text{-}int\&stack}\ (\lambda i\,.\,(t \circ (f\ \ i)))) \tag{4.3}$$

where t maps *errorStack* to *errorStack*

It is worth noting that the denotational proof that $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}] =_{obs} [\texttt{3}, \texttt{add}]$ has a very different flavor than the operational proof of this fact given in Section 3.4.4. The operational proof worked by case analysis on the initial stack. The denotational proof in Figure 4.20 works purely by equational reasoning — there is no hint of case analysis here. This is because the all the case analyses are hidden within the carefully chosen abstractions *with-int&stack* and *push* and equalities (4.1)–(4.3). The case analyses would become apparent if these were expanded to show explicit **matching** expressions.

Denotational justifications for the safety of transformations are not limited to PostFix. For example, Figure 4.21 shows that EL numerical expressions `(+ NE NE)` and `(* 2 NE)` have the same meaning. So one can safely be substituted for the other in any EL program without changing the meaning of the program.

▷ **Exercise 4.8**

a. Prove equalities (4.1)–(4.3).

b. Equality (4.3) requires that t maps *errorStack* to *errorStack*. Show that the equality is not true if this requirement is violated. ◁

$$
\begin{array}{l}
(\mathcal{Q}[\![\texttt{1 add 2 add}]\!]) \\
= (\mathcal{C}[\![\texttt{add}]\!]) \circ (\mathcal{C}[\![\texttt{2}]\!]) \circ (\mathcal{C}[\![\texttt{add}]\!]) \circ (\mathcal{C}[\![\texttt{1}]\!]) \text{ , by (4.1)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_1\,' \,.\, (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2\,' \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2\,' + i_1\,')))))))) \\
\quad \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (\mathcal{N}[\![\texttt{2}]\!])))) \\
\quad \circ (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_1 \,.\, (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + i_1)))))))) \\
\quad \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (\mathcal{N}[\![\texttt{1}]\!])))) \text{ , by definition of } \mathcal{C} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2\,' \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2\,' + 2))))) \\
\quad \circ \ (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + 1)))))) \text{ , by (4.2)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2 \,.\, \ (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2\,' \,.\, \ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2\,' + 2)))))) \\
\qquad \circ \ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + 1)))))), \text{ by (4.3)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ ((i_2 + 1)\ + 2))))) \text{ , by (4.2)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + 3)))))) \text{ , by definition of } +_{\mathit{Int}} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_3 \,.\, (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + i_3))))))) \\
\quad \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (\mathcal{N}[\![\texttt{3}]\!])))) \text{ , by (4.2)} \\
= (\mathcal{C}[\![\texttt{add}]\!]) \circ (\mathcal{C}[\![\texttt{3}]\!]) \text{ , by definition of } \mathcal{C} \\
= (\mathcal{Q}[\![\texttt{3 add}]\!]) \text{ , by (4.1)}
\end{array}
$$

Figure 4.20: Proof that $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}]$ and $[\texttt{3}, \texttt{add}]$ are denotationally equivalent. This implies that the two sequences are observationally equivalent.

$$
\begin{aligned}
&\mathcal{NE}[\![\texttt{(+ } NE\ \ NE\texttt{)}]\!] \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_1\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{+}]\!]\ \ i_1\ \ i_2))) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (i_2 +_{Int} i_2) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (2 \times_{Int} i_2) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{+}]\!]\ \ i_2\ \ i_2)) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{*}]\!]\ \ 2\ \ i_2)) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![\texttt{2}]\!]\ \ i^*)\ \ (\lambda i_1\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{*}]\!]\ \ i_1\ \ i_2))) \\
&= \mathcal{NE}[\![\texttt{(* 2 } NE\texttt{)}]\!]
\end{aligned}
$$

Figure 4.21: Denotational proof that `(+ NE NE)` may safely be replaced by `(* 2 NE)` in EL.

▷ **Exercise 4.9**

a. We have seen that `(postfix 2 3 sub swap pop)` and `(postfix 2 3 sub)` are equivalent programs. But in general it is *not* safe to replace the command sequence `3 sub swap pop` by `3 sub`. Give a context in which this replacement would change the meaning of a program.

b. Use denotational reasoning to show that it is safe to replace any of the following command sequences by `3 sub swap pop`:

   i. `swap pop 3 sub`

   ii. `(3 sub) swap pop exec`

   iii. `3 2 nget swap sub swap pop swap pop` ◁

▷ **Exercise 4.10** Use the PostFix denotational semantics to either prove or disprove the purported observational equivalences in Exercise 3.28. ◁

▷ **Exercise 4.11** Use the EL denotational semantics to either prove or disprove the safety of the EL transformations in Exercise 3.32. ◁

4.4.3 Technical Difficulties

The denotational definition of PostFix depends crucially on some subtle details. As a hint of the subtlety, consider what happens to our denotational definition if we extend PostFix with our old friend `dup`. A valuation clause for `dup` seems straightforward:

$$\mathcal{C}[\![\texttt{dup}]\!] = \lambda s\,.\,(\mathit{push}\ \ (\mathit{top}\ \ s)\ \ s)\,.$$

At the same time we know that adding `dup` to the language introduces the possibility that programs may not terminate. Yet, the signature for $\mathcal{P}$ declares that programs map to the *Answer* domain, and the *Answer* domain does not include any entity that represents nontermination. What's going on here?

The source of the problem is the recursive structure of the semantic domains for PostFix. As the domain equations show, the *StackTransform*, *Stack*, and *Value* domains are mutually recursive:

$$\begin{aligned} \mathit{StackTransform} &= \mathit{Stack} \to \mathit{Stack} \\ \mathit{Stack} &= \mathit{Value}^* + \mathit{Error} \\ \mathit{Value} &= \mathit{Int} + \mathit{StackTransform} \end{aligned}$$

It turns out that solving such recursive domain equations sometimes requires extending some domains with an element that models nontermination, written $\perp$ and pronounced "bottom." We will study this element in more detail in the next chapter, where it plays a prominent role. In the case of PostFix, it turns out that both the *Stack* and *Answer* domains must include $\perp$, and this is able to model the meaning of non-terminating command sequences.

4.4.4 Relating Operational and Denotational Semantics

We have presented the operational and denotational semantics of several simple languages, but have not studied the connection between them. What is the relationship between these two forms of semantics? How can we be sure that reasoning done with one form of semantics is valid in the other?

4.4.4.1 Soundness

Assume that an operational semantics has a deterministic behavior function of the form

$$\mathit{beh}_{\mathit{det}} : (\text{Program} \times \text{Inputs}) \to \text{Outcome}$$

and that the related denotational semantics has a meaning function

$$\mathit{meaning} : (\text{Program} \times \mathit{Args}) \to \mathit{Answer},$$

where *Args* is a domain of program arguments and *Answer* is the domain of final answers. Also suppose that there is a function *in* that maps between the syntactic and semantic input domains and a function *out* that maps between the syntactic and semantic output domains:

$$\begin{aligned} \mathit{in} &: \text{Inputs} \to \mathit{Args} \\ \mathit{out} &: \text{Outcome} \to \mathit{Answer}. \end{aligned}$$

Then we define the following notion of soundness:

$I \in \text{Inputs} = \text{Intlit}^*$
$o \in \text{Outcome} = \text{Intlit} + \text{StuckOut}$
$\text{StuckOut} = \{\texttt{stuckout}\}$
$ar \in \mathit{Args} = \mathit{Int}^*$
$a \in \mathit{Answer} = \mathit{Int} + \mathit{Error}$
$\mathit{Error} = \{\mathit{error}\}$

$\mathit{in} : \text{Inputs} \to \mathit{Args}$
$\mathit{in} = \lambda N^* .\ (\mathit{map}\ \mathcal{N}\ N^*)$

$\mathit{out} : \text{Outcome} \to \mathit{Answer}$
$\mathit{out} = \lambda o .\ \textbf{matching}\ o$
$\triangleright\, (\text{Intlit} \mapsto \text{Outcome}\ N)\ [\!]\ (\mathit{Int} \mapsto \mathit{Answer}\ (\mathcal{N}[\![N]\!]))$
$\triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}$

$\mathit{beh}_{\mathit{det}} : (\text{Program} \times \text{Inputs}) \to \text{Outcome}$
$\mathit{beh}_{\mathit{det\,EL}}$ is defined in Exercise 3.10
and $\mathit{beh}_{\mathit{det\,PostFix}}$ is defined in Section 3.2.2.

$\mathit{meaning} : (\text{Program} \times \mathit{Args}) \to \mathit{Answer}$
$\mathit{meaning}_{EL} = \lambda\langle P, ar\rangle .\ (\mathcal{P}_{EL}[\![P]\!]\ ar)$, where $\mathcal{P}_{EL}$ is defined in Section 4.2.4.
$\mathit{meaning}_{PostFix} = \lambda\langle P, ar\rangle .\ (\mathcal{P}_{PostFix}[\![P]\!]\ ar)$,
where $\mathcal{P}_{PostFix}$ is defined in Section 4.3.2.

Figure 4.22: Instantiation of soundness components for EL and PostFix.

> **Denotational Soundness:** A denotational semantics is **sound with respect to (wrt)** an operational semantics if for all programs P and inputs I,
>
> $$\mathit{meaning}\ \langle P, (\mathit{in}\ I)\rangle = (\mathit{out}\ (\mathit{beh}_{\mathit{det}}\ \langle P, I\rangle)) .$$

This definition says that the denotational semantics agrees with the operational semantics on the result of executing a program on any given inputs. Figure 4.22 shows how the parts of the soundness definition can be instantiated for EL and PostFix.

We will now sketch a proof that the denotational semantics for PostFix is sound wrt the operational semantics for PostFix. The details of this proof, and a denotational soundness proof for EL, are left as exercises. The essence of the denotational soundness proof for PostFix is to define the meaning of an operational configuration, and show that each step in the PostFix SOS preserves this meaning. Recall that a configuration in the PostFix SOS has the form Commands × Stack, where

$\mathcal{V}$: Value $\rightarrow$ *Value*
‘ $= \lambda V$. **matching** V
$\triangleright$ (Intlit $\mapsto$ Value N) $[\!]$ $(Int \mapsto Value\ (\mathcal{N}[\![N]\!]))$
$\triangleright$ (Commands $\mapsto$ Value Q) $[\!]$ $(StackTransform \mapsto Value\ (\mathcal{Q}[\![Q]\!]))$
endmatching

$\mathcal{S}$: Stack $\rightarrow$ *Stack* $= \lambda V^*$. $(Value^* \mapsto Stack\ (map\ \mathcal{V}\ V^*))$

$\mathcal{CF}$: Commands $\times$ Stack $\rightarrow$ *Answer* $= \lambda\langle Q, S\rangle$. $resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (\mathcal{S}[\![S]\!])))$

Figure 4.23: Meaning of a PostFix configuration.

$S \in$ Stack $=$ Value*
$V \in$ Value $=$ Intlit $+$ Commands

Figure 4.23 defines a function $\mathcal{CF}$ that maps an operational configuration to an element of *Answer*. We establish the following lemmas:

1. For any PostFix program $P =$ `(postfix` $N_{numargs}$ Q`)` and numerals N^*,

$$(\mathcal{P}[\![P]\!]\ (in\ N^*)) = \mathcal{CF}[\![(IF\ \langle P, N^*\rangle)]\!],$$

where *IF* is the input function defined in Figure 3.3 that maps a PostFix program and inputs into an initial SOS configuration. There are two cases:

(a) When $\mathcal{N}[\![N_{numargs}]\!] = (length\ N^*)$, both the left and right hand sides of the equation denote

$$resToAns\,(top\ (\mathcal{Q}[\![Q]\!]\ (Value^* \mapsto Stack\ (map\ (Int \mapsto Value \circ \mathcal{N})\ N^*)))).$$

(b) When $\mathcal{N}[\![N_{numargs}]\!] \neq (length\ N^*)$, the left hand side of the equation denotes *errorAnswer* and the right hand side denotes

$$\begin{aligned}
&\mathcal{CF}[\![(IF\ \langle P, N^*\rangle)]\!]\\
&= \mathcal{CF}[\![\langle [\,]_{\text{Commands}}, [\,]_{\text{Stack}}\rangle]\!]\\
&= resToAns\ (top\ (\mathcal{Q}[\![[\,]_{\text{Commands}}]\!]\ (Value^* \mapsto Stack\ [\,]_{Stack})))\\
&= resToAns\ (top\ (Value^* \mapsto Stack\ [\,]_{Stack}))\\
&= errorAnswer.
\end{aligned}$$

2. For any transition $cf \Rightarrow cf'$, $\mathcal{CF}[\![cf]\!] = \mathcal{CF}[\![cf']\!]$. This can be shown by demonstrating this equality for each of the PostFix transition rules in Figure 3.4. For example, one such rule is:

$$\langle \texttt{exec}\ .\ Q_{rest},\ (\ Q_{exec}\ )\ .\ S\rangle \Rightarrow \langle Q_{exec}\ @\ Q_{rest},\ S\rangle \qquad [execute]$$

For this rule we have

$$\begin{aligned}
&\mathcal{CF}[\![\langle \texttt{exec}\ .\ Q_{rest},\ (\ Q_{exec}\ )\ .\ S\rangle]\!]\\
&= resToAns\ (top\ (\mathcal{Q}[\![\texttt{exec}\ .\ Q_{rest}]\!]\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*))))),\\
&\quad \text{where } v^* = (map\ \mathcal{V}\ S)\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q_{rest}]\!]\ (\mathcal{C}[\![\texttt{exec}]\!]\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*)))))\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q_{rest}]\!]\\
&\qquad (transform\ (top\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*)))\\
&\qquad\qquad (pop\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*)))))))\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q_{rest}]\!]\\
&\qquad (transform\ (Value \mapsto Result\ (StackTransform \mapsto Value\ (\mathcal{Q}[\![Q_{exec}]\!])))\\
&\qquad\qquad (Value^* \mapsto Stack\ v^*))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q_{rest}]\!]\ (\mathcal{Q}[\![Q_{exec}]\!]\ (Value^* \mapsto Stack\ v^*))))\\
&= resToAns\ (top\ ((\mathcal{Q}[\![Q_{rest}]\!] \circ \mathcal{Q}[\![Q_{exec}]\!])\ (Value^* \mapsto Stack\ v^*)))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q_{rest}\ @\ Q_{exec}]\!]\ (Value^* \mapsto Stack\ (map\ \mathcal{V}\ S))))\\
&= \mathcal{CF}[\![\langle Q_{exec}\ @\ Q_{rest},\ S\rangle]\!].
\end{aligned}$$

3. For any stuck configuration cf, $\mathcal{CF}[\![cf]\!] = errorAnswer$. This can be shown by enumerating the finite number of configuration patterns that stand for configurations in $Irreducible_{PFSOS}$, and showing that each denotes the error answer. For example, one such pattern is $\langle \texttt{swap}\ .\ Q,\ [\ V]\rangle$:

$$\begin{aligned}
&\mathcal{CF}[\![\langle \texttt{swap}\ .\ Q,\ [\ V]\rangle]\!]\\
&= resToAns\ (top\ (\mathcal{Q}[\![\texttt{swap}\ .\ Q]\!]\ (Value^* \mapsto Stack\ [\mathcal{V}\ V])))\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q]\!]\ (push\ (top\ (pop\ (Value^* \mapsto Stack\ [\mathcal{V}\ V])))\\
&\qquad\qquad (push\ (top\ (Value^* \mapsto Stack\ [\mathcal{V}\ V]))\\
&\qquad\qquad\quad (pop\ (pop\ (Value^* \mapsto Stack\ [\mathcal{V}\ V])))))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (push\ (top\ (Value^* \mapsto Stack\ [\,]))\\
&\qquad\qquad\qquad (push\ (Value \mapsto Result\ (\mathcal{V}[\![V]\!]))\\
&\qquad\qquad\qquad\quad (pop\ (Value^* \mapsto Stack\ [\,])))))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (push\ errorResult\\
&\qquad\qquad\qquad (push\ (\mathcal{V}[\![V]\!])\ \ errorStack))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ errorStack))\\
&= resToAns\ (top\ errorStack)\\
&= errorAnswer.
\end{aligned}$$

We're now ready to put the lemmas together to show denotational soundness for a POSTFIX program (`postfix` $N_{numargs}$ Q_{body}) executed on inputs N_{inputs}*. There are two cases:

1. $\mathcal{N}[\![N_{numargs}]\!] = (length\ N_{inputs}^*)$ and the initial program configuration has a transition path to a final configuration:

$$\langle Q_{body},\ N_{inputs}^*\rangle \stackrel{*}{\Rightarrow} \langle [\,]_{\text{Commands}},\ N_{ans}\ .\ V_{rest}^*\rangle$$

In this case,

$$\begin{aligned}
&meaning\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), (in\ N_{inputs}^*)\rangle\\
&= \mathcal{P}[\![(\texttt{postfix}\ N_{numargs}\ Q_{body})]\!]\ (in\ N_{inputs}^*)\\
&= \mathcal{CF}[\![(IF\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)]\!]\text{ , by lemma 1}\\
&= \mathcal{CF}[\![\langle Q_{body},\ (map\ \text{Intlit} \mapsto \text{Value}\ N_{inputs}^*)\rangle]\!]\\
&= \mathcal{CF}[\![\langle [\,]_{\text{Commands}},\ (\text{Intlit} \mapsto \text{Value}\ N_{ans})\ .\ V_{rest}^*\rangle]\!]\text{ , by lemma 2 on each } \Rightarrow\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![]\ (Value^* \mapsto Stack\ ((Int \mapsto Value\ (\mathcal{N}[\![N_{ans}]\!]))\ .\ (map\ \mathcal{V}\ V_{rest}^*)))))\\
&= resToAns\ (top\ (Value^* \mapsto Stack\ ((Int \mapsto Value\ (\mathcal{N}[\![N_{ans}]\!]))\ .\ (map\ \mathcal{V}\ V_{rest}^*))))\\
&= (Int \mapsto Answer\ (\mathcal{N}[\![N_{ans}]\!]))\\
&= (out\ (\text{Intlit} \mapsto \text{Outcome}\ N_{ans}))\\
&= (out\ (beh_{det\,PostFix}\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)).
\end{aligned}$$

2. $\mathcal{N}[\![N_{numargs}]\!] \neq (length\ N_{inputs}^*)$ or the initial program configuration has a transition path to a stuck configuration. In these cases,

$$IF\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N^*\rangle \stackrel{*}{\Rightarrow} cf_{stuck},$$

where cf_{stuck} is a stuck configuration. Then we have:

$$\begin{aligned}
&meaning\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), (in\ N_{inputs}^*)\rangle\\
&= \mathcal{P}[\![(\texttt{postfix}\ N_{numargs}\ Q_{body})]\!]\ (in\ N_{inputs}^*)\\
&= \mathcal{CF}[\![(IF\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)]\!]\text{ , by lemma 1}\\
&= \mathcal{CF}[\![cf_{stuck}]\!]\text{ , by lemma 2 on each } \Rightarrow\\
&= errorAnswer\text{, by lemma 3}\\
&= (out\ \texttt{stuck})\\
&= (out\ (beh_{det\,PostFix}\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)).
\end{aligned}$$

This completes the sketch of the proof that the denotational semantics for PostFix is sound with respect to the operational semantics for PostFix. The fact that all PostFix programs terminate simplifies the proof, because it is not necessary to consider the case of infinitely long transition paths (in which case $(beh_{det}\ \langle P, I\rangle) = \infty$). For languages containing nonterminating programs, a denotational soundness proof must also explicitly handle this case.

▷ **Exercise 4.12** Complete the proof that the denotational semantics for PostFix is sound with respect to its operational semantics by fleshing out the following details:

a. Show that lemma 2 holds for each transition rule in Figure 3.4.

b. Make a list of all stuck configuration patterns in the PostFix SOS and show that lemma 3 holds for each such pattern. ◁

▷ **Exercise 4.13** Show that the denotational semantics for each of the following languages is sound with respect to its operational semantics: (1) a version of ELMM whose operators include only +, -, and *; (2) full ELMM; (3) ELM; and (4) EL. ◁

4.4.4.2 Adequacy

The notion of soundness developed above works at the level of a whole program. But often we want to reason about smaller phrases within a program. In particular, we want to reason that we can substitute one phrase for another without changing the operational behavior of the program. The following adequacy property says that denotational equivalence implies the operational notion of observational equivalence:

> **Adequacy:** Suppose that $\mathbb{P}$ ranges over program contexts, H ranges over the kinds of phrases that fill the holes in program contexts, and $\mathcal{H}$ is a denotational meaning function for phrases. A denotational semantics is **adequate with respect to (wrt)** an operational semantics if the following holds:
>
> $$\mathcal{H}[\![H_1]\!] = \mathcal{H}[\![H_2]\!] \text{ implies } H_1 =_{obs} H_2.$$

Recall from page 84 that $H_1 =_{obs} H_2$ means that for all program contexts $\mathbb{P}$ and all inputs I, $beh\ \langle \mathbb{P}\{H_1\}, I\rangle = beh\ \langle \mathbb{P}\{H_2\}, I\rangle$

In the case of a deterministic behavior function, the following reasoning shows that adequacy is *almost* implied by denotational soundness:

$\mathcal{H}[\![H_1]\!] = \mathcal{H}[\![H_2]\!]$

implies $\mathcal{P}[\![\mathbb{P}\{H_1\}]\!] = \mathcal{P}[\![\mathbb{P}\{H_2\}]\!]$, by compositionality of denotational semantics

implies $\mathit{meaning}\ \langle \mathbb{P}\{H_1\}, (\mathit{in}\ I)\rangle = \mathit{meaning}\ \langle \mathbb{P}\{H_2\}, (\mathit{in}\ I)\rangle$ for any inputs I

implies $(\mathit{out}\ (\mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_1\}, I\rangle)) = (\mathit{out}\ (\mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_2\}, I\rangle))$, by soundness

But demonstrating the observational equivalence requires showing

$$\mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_1\}, I\rangle = \mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_2\}, I\rangle.$$

To conclude this from the above line of reasoning requires an additional property. Suppose that A ranges over observable answer expressions in the syntactic domain Answer. Then we need a property we shall call **denotational distinctness of observables**:

$$(\mathit{out}\ A_1) = (\mathit{out}\ A_2) \text{ implies } A_1 = A_2.$$

Recall that *out* maps syntactic answers to semantic ones. So the above property requires that syntactically distinct answers be denotationally distinct. That is, we cannot have two observationally distinct answers answers with the same meaning.

Both EL and PostFix have denotational distinctness of observables. In each language, observable answers are either integer numerals or an error token. Assuming that only canonical integer numerals are used (e.g., `17` rather than `017` or `+17`) distinct integer numerals denote distinct integers. Note that PostFix would *not* have this property if executable sequences at the top of a final stack could be returned as observable answers. For example, the syntactically distinct sequences `(1 add 2 add)` and `(3 add)` would both denote the same transformation:

$$(\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))).$$

The above discussion allows us to conclude that any language with denotational soundness and denotational distinctness of observables has the adequacy property. In turn, this property justifies the use of denotational reasoning for proving the safety of program transformations. For example, the PostFix Denotational Equivalence Theorem on page 141 is a corollary of the adequacy of PostFix.

4.4.4.3 Full Abstraction

Changing the unidirectional implication of adequacy to a bidirectional implication yields a stronger property called **full abstraction**:

> **Full Abstraction:** Suppose that $\mathbb{P}$ ranges over program contexts, H ranges over the kinds of phrases that fill the holes in program contexts, and $\mathcal{H}$ is a denotational meaning function for phrases. A denotational semantics is **adequate with respect to (wrt)** an operational semantics if the following holds:
>
> $$\mathcal{H}[\![H_1]\!] = \mathcal{H}[\![H_2]\!] \text{ iff } H_1 =_{obs} H_2.$$

In addition to adequacy, full abstraction requires that observational equivalence imply denotational equivalence. That is, program fragments that behave the same in all contexts must have the same denotational meaning.

The various dialects of EL we have considered are all fully abstract. Consider the restricted version of ELMM in which the only operations are `+`, `-`, and `*`. In this language, every numerical expression denotes an integer. We already know that this language is adequate; to prove full abstraction, we need to show that observational equivalence implies denotational equivalence. We will prove this by contradiction. Suppose that $NE_1 =_{obs} NE_2$, but $\mathcal{NE}[\![NE_1]\!] \neq \mathcal{NE}[\![NE_2]\!]$. Consider the ELMM context $\mathbb{P} =$ `(elmm` $\Box$`)`. Modeling the non-existent inputs in this case by *unit*, we have:

$(out\ \ (beh_{det}\ \langle \mathbb{P}\{NE_1\}, unit\rangle))$
$= \mathcal{P}[\![\mathbb{P}\{NE_1\}]\!]$, by soundness
$= \mathcal{NE}[\![NE_1]\!]$, by definition of $\mathcal{P}$
$\neq \mathcal{NE}[\![NE_2]\!]$, by assumption
$= \mathcal{P}[\![\mathbb{P}\{NE_2\}]\!]$, by definition of $\mathcal{P}$
$= (out\ \ (beh_{det}\ \langle \mathbb{P}\{NE_2\}, unit\rangle))$, by soundness

Because ELMM has denotational distinctness of observables, we conclude that $NE_1 \neq_{obs} NE_2$, contradicting our original assumption. A similar proof works to show full abstraction for the other dialects of EL we have studied.

Surprisingly, PostFix is *not* fully abstract! As argued in Section 4.4.3, even though all PostFix programs terminate, the denotational domains for answers and stacks in PostFix must include an entity denoting nontermination, which we will write as $\perp$. This is the denotational analog of the operational token ∞. Even though no PostFix command sequence can loop, the presence of $\perp$ in the semantics can distinguish the meanings of some observationally equivalent command sequences.

For example, consider the following two command sequences:

$Q_1 =$ `1 0 div`
$Q_2 =$ `exec 1 0 div`.

Q_1 signals an error for any stack. Q_2 first executes the top value V_{top} on the stack and then executes `1 0 div`. We argue that Q_2 is observationally equivalent to Q_1, because it will also signal an error for any stack:

- if the stack is empty or if V_{top} is not an executable sequence, the attempt to perform `exec` will fail with an error;
- if V_{top} is an executable sequence, Q_2 will execute it. Since all POSTFIX command sequences terminate, the execution of V_{top} will either signal an error, or it will terminate without an error. In the latter case, the execution continues with `1 0 div`, which necessarily signals an error.

Even though $Q_1 =_{obs} Q_2$, they do not denote the same stack transform! To see this, consider a stack transform $t_{weird} = \lambda s \,.\, \bot$ and a stack s_{weird} whose top value is $(\mathit{StackTransform} \mapsto \mathit{Value}\ t_{weird})$. Both t_{weird} and s_{weird} are "weird" in the sense that they can never arise during a POSTFIX computation, in which all stack transforms necessarily terminate. Nevertheless, t_{weird} is a legal element of the domain *StackTransform*, and it must be considered as a legal stack element in denotational reasoning. Observe that $(\mathcal{Q}[\![Q_1]\!]\ s_{weird}) = \mathit{errorStack}$, but $(\mathcal{Q}[\![Q_2]\!]\ s_{weird}) = \bot$ — i.e., the latter computation does not terminate. So Q_1 and Q_2 denote distinct stack transforms even though they are observationally equivalent.

Intuitively, full abstraction says that the semantic domains don't contain any extra "junk" that can't be expressed by phrases in the language. In the case of POSTFIX, the domains harbor $\bot$ even though it cannot be expressed in the language.

4.4.5 Operational vs. Denotational: A Comparison

We have noted in this chapter that a denotational semantics expresses the meaning of a program in a much more direct way than an operational semantics. Furthermore, the compositional nature of a denotational semantics is a real boon for proving properties of programs and languages. Why would we ever want to choose an operational semantics over a denotational semantics?

For one thing, an operational semantics is usually a more natural medium for expressing the step-by-step nature of program execution. The notion of "step" is an important one: it is at the heart of a mechanistic view of computation; it provides a measure by which computations can be compared (e.g., which takes the fewest steps); and it provides a natural way to talk about nondeterminism (choice between steps) and concurrency (interleaving the steps of more than one process). What counts as a natural step for a program is explicit in the rewrite

rules of an SOS. These notions cannot always be expressed straightforwardly in a denotational approach. Furthermore, in computer science, the bottom line is often what actually runs on a machine, and the operational approach is much closer to this bottom line.

From a mathematical perspective, the advantage of an operational semantics is that it's often much easier to construct than a denotational semantics. Since the objects manipulated by an SOS are simple syntactic entities, there are very few constraints on the form of an operational semantics. Any SOS with a deterministic set of rewrite rules specifies a well-defined behavior function from programs to answer expressions. Creating or extending a set of rewrite rules is fairly painless since it rarely requires any deep mathematical reasoning. Of course, the same emphasis on syntax that facilitates the construction of an operational semantics limits its usefulness for reasoning about programs. For example, it's difficult to see how some local change to the rewrite rules affects the global properties of a language.

Constructing a denotational semantics, on the other hand, is mathematically much more intensive. It is necessary to build consistent mathematical representations for each kind of meaning object. The difficulty of building such models in general is illustrated by the fact that there was no mathematically viable interpretation for recursive domain equations until Dana Scott invented one in the early 1970s. Since then, a variety of tools and techniques have been developed that make it easier to construct a denotational semantics that maps programs into a restricted set of meanings. Extending this set of meanings requires potentially difficult proofs that the extensions are sound, so most semanticists are content to stick with the well-understood meanings. This class of meanings is large enough, however, to facilitate a wide range of formal reasoning about programs and programming languages.

Reading

Denotational semantics was invented by Christopher Strachey and Dana Scott. For a tutorial introduction to denotational semantics, we recommend the articles [Ten76] and [Mos90]. Coverage of both operational and denotational semantics along with their use in reasoning about several simple programming languages can be found in several semantics textbooks [Gun92, Win93, Mit96]. Full-length books devoted to denotational semantics include [Gor79, Sto85, Sch86a].

Our notions of denotational soundness and adequacy are somewhat different than (but related to) those in the literature. For a discussion of (the traditional approach to) soundness, adequacy, and full abstraction, see [Gun92].

Chapter 5

Fixed Points

Bottom! O most courageous day! O most happy hour!

— A Midsummer Night's Dream, William Shakespeare

Recursive definitions are a powerful and elegant tool for specifying complex structures and processes. While such definitions are second nature to experienced programmers, novices are often mystified by recursive definitions. Their confusion often centers on the following question: "how can something be defined in terms of itself?" Sometimes there is a justifiable cause for confusion — not all recursive definitions make sense!

In this chapter, we carve out a class of recursive definitions that *do* make sense, and present a technique for assigning meaning to them. The technique involves finding a fixed point of a function derived from the recursive definition. We will make extensive use of this technique in our denotational descriptions of programming languages to define recursive valuation functions and recursive domains.

5.1 The Fixed Point Game

5.1.1 Recursive Definitions

For our purposes, a **recursive definition** is an equation of the form

$$x \;=\; \ldots\, x \,\ldots$$

where $\ldots\ x\ \ldots$ designates a mathematical expression that contains occurrences of the defined variable x. Mutually recursive definitions of the form

$$\begin{array}{rcl} x_1 & = & \ldots\ x_1\ \ldots\ x_n\ \ldots \\ & \vdots & \\ x_n & = & \ldots\ x_1\ \ldots\ x_n\ \ldots \end{array}$$

can always be rephrased as a single recursive definition

$$\begin{array}{rcl} x & = & \langle \ldots\ (Proj\,1\ \ x)\ \ldots\ (Proj\,n\ \ x)\ \ldots\ , \\ & & \vdots \\ & & \ldots\ (Proj\,1\ \ x)\ \ldots\ (Proj\,n\ \ x)\ \ldots \rangle, \end{array}$$

where x stands for the n-tuple $\langle x_1, \ldots, x_n \rangle$ and $Proj\,i$ extracts the ith element of the tuple. For this reason, it is sufficient to focus on recursive definitions involving a single variable.

A **solution** to a recursive definition is a value that makes the equation true when substituted for all occurrences of the defined variable. A recursive definition may have zero, one, or more solutions. For example, suppose that x ranges over the integers. Then:

- $x = 1 + x$ has no solutions;
- $x = 4 - x$ has exactly one solution (2);
- $x = \frac{9}{x}$ has two solutions (-3, 3);
- $x = x$ has an infinite number of solutions (each integer).

It is important to specify the domain of the defined variable in a recursive definition, since the set of solutions depends on this domain. For example, the recursive definition $x = \frac{1}{16x^3}$ has

- zero solutions over the integers[1];
- one solution over the positive rationals ($\frac{1}{2}$);
- two solutions over the rationals ($\frac{1}{2}$, $-\frac{1}{2}$);
- four solutions over the complex numbers ($\frac{1}{2}$, $-\frac{1}{2}$, $\frac{i}{2}$, $-\frac{i}{2}$).

[1] In this case, division is interpreted as a quotient function on integers.

In fact, many numerical domains were invented precisely to solve classes of equations that were insoluble with existing domains.

Although we are most familiar with equations that involve numeric variables, equations can involve variables from *any* domain, including product, sum, sequence, and function domains. For example, consider the following recursive definitions involving an element p of the sequence domain $Nat \times Nat$:

- $p = \langle (Proj\,2\ \ p)\,, (Proj\,1\ \ p)\rangle$ has an infinite number of solutions of the form $\langle n, n\rangle$, where $n : Nat$.
- $p = \langle (Proj\,2\ \ p)\,, (Proj\,1\ \ p)\ \text{-}\ 1\rangle$ has the unique solution $\langle 0, 0\rangle$.[2]
- $p = \langle (Proj\,2\ \ p)\,, (Proj\,1\ \ p)\ +1\rangle$ has no solutions in $Nat \times Nat$. The first element n of a solution $p = \langle n, \ldots\rangle$ would have to satisfy the equation $n = n + 1$, and this equation has no solutions.

We can also have recursive definitions involving an element s of the sequence domain Nat^*:

- $s = (cons\ \ 3\ \ (tail\ \ s))$ has an infinite number of solutions: all non-empty sequences s whose first element is 3.
- $s = (cons\ \ 3\ \ s)$ has no solutions in Nat^*, which includes only finite sequences of natural numbers and so does not contain an infinite sequence of 3s. However, this equation *does* have a solution in a domain that includes infinite sequences of numbers in addition to the finite ones. We shall use the notation $\overline{Nat^*}$ to designate this domain.
- $s = (cons\ \ 3\ \ (tail\ \ (tail\ \ s)))$ has the unique solution [3]. This definition requires that $(tail\ \ s) = (tail\ \ (tail\ \ s))$, and in Nat^* only a singleton sequence s satisfies this requirement.[3] However, in $\overline{Nat^*}$, this equation has an infinite number of solutions, since for any integer i, an infinite sequence of is satisfies $(tail\ \ s) = (tail\ \ (tail\ \ s))$.

We will be especially interested in recursive definitions over function domains. Suppose that f is an element of the domain $Nat \to Nat$. Consider the following recursive function definition of f:

$$f\ =\ \lambda n\,.\ \textbf{if}\ (n = 0)\ \textbf{then}\ 0\ \textbf{else}\ (2 +\ (f\ \ (n - 1)))\ \textbf{fi}.$$

[2] Recall that $(n_1 - n_2) = 0$ if $n_1, n_2 : Nat$ and $(n_1 < n_2)$.
[3] Recall that $(tail\ [\,])$ is defined to be $[\,]$.

Intuitively, this equation is solved when f is a doubling function, but how do we show this more formally? Recall that a function in $Nat \to Nat$ can be viewed as its graph, the set of input/output pairs for the function. The graph associated with the lambda expression is

$$\begin{array}{l}\{\langle 0, \textbf{if } (0 = 0) \textbf{ then } 0 \textbf{ else } (2 + (f\ 0))\rangle,\\ \ \langle 1, \textbf{if } (1 = 0) \textbf{ then } 0 \textbf{ else } (2 + (f\ 0))\rangle,\\ \ \langle 2, \textbf{if } (2 = 0) \textbf{ then } 0 \textbf{ else } (2 + (f\ 1))\rangle,\\ \ \langle 3, \textbf{if } (3 = 0) \textbf{ then } 0 \textbf{ else } (2 + (f\ 2))\rangle,\\ \ \ldots\ \}.\end{array}$$

After simplification, this becomes

$$\{\langle 0, 0\rangle, \langle 1, (2 + (f\ 0))\rangle,\ \langle 2, (2 + (f\ 1))\rangle,\ \langle 3, (2 + (f\ 2))\rangle,\ \ldots\ \}.$$

If f is a doubling function, then the graph of the right-hand side can be further simplified to

$$\{\langle 0, 0\rangle,\ \langle 1, 2\rangle,\ \langle 2, 4\rangle,\ \langle 3, 6\rangle,\ \ldots\ \}.$$

This is precisely the graph of the doubling function f on the left-hand side of the equation, so the equation holds true. It is not difficult to show that the doubling function is the *only* solution to the equation; we leave this as an exercise.

As with recursive definitions over other domains, recursive definitions of functions may have zero, one, or more solutions. Maintaining the assumption that f is in $Nat \to Nat$, the definition

$$f \ = \ \lambda n\,.\,(1 + (f\ n))$$

has zero solutions, because the result n_r for any given input would have to satisfy $n_r = n_r + 1$. On the other hand, the definition

$$f \ = \ \lambda n\,.\,(f\ (1 + n))$$

has an infinite number of solutions: for any given constant n_c, a function with the graph $\{\langle n, n_c\rangle \mid n : Nat\}$ is a solution to the equation.

5.1.2 Fixed Points

If d ranges over domain D, then a recursive definition

$$d \ = \ (\ldots\ d\ \ldots)$$

can always be encoded as the $D \to D$ function

$$\lambda d\,.\,(\ \ldots\ d\ \ldots).$$

We will call this the **generating function** for the recursive definition. For example, if $r : Real$, the numeric equation

$$r \;=\; 1 - r^{\,2}$$

can be represented by the $Real \to Real$ generating function

$$\lambda r\,.\;(1 -\; (r * r))\,.$$

Similarly, the recursive function definition

$$dbl\;:\;Nat \to Nat \;=\; \lambda n\,.\,\textbf{if}\;(n = 0)\;\textbf{then}\,0\,\textbf{else}\;(2 +\;(dbl\;\;(n-1)))\;\textbf{fi}$$

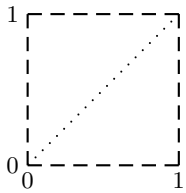
can be represented by the generating function

$$\begin{array}{l} g_{dbl} : (Nat \to Nat) \to (Nat \to Nat) \\ = \lambda f\,.\;\;\lambda n\,.\;\;\textbf{if}\,(n = 0)\;\;\textbf{then}\,0\;\;\textbf{else}\,(2 +\;(f\;\;(n-1)))\;\;\textbf{fi}, \end{array}$$

where $f : Nat \to Nat$. A generating function is not recursive, so its meaning can be straightforwardly determined from its component parts.

A solution to a recursive definition is a fixed point of its associated generating function. A **fixed point** of a function $g : D \to D$ is an element $d : D$ such that $(g\;\;d) = d$. If a function in $D \to D$ is viewed as moving elements around the space D, elements satisfying this definition are the only ones that remain stationary; hence the name "fixed point."

To build intuitions about fixed points, it is helpful to consider functions from the unit interval[4] $[0, 1]$ to itself. Such functions can be graphed in the following box:



Every point where the function graph intersects the $y = x$ diagonal is a fixed point of the function. For example, Figure 5.1 shows the graphs of functions with zero, one, two, and an infinite number of fixed points.

It is especially worthwhile to consider how a generating function like g_{dbl} moves elements around a domain of functions. Here are a few examples of how g_{dbl} maps various functions $f : Nat \to Nat$:

[4]The unit interval is the set of real numbers between 0 and 1, inclusive.

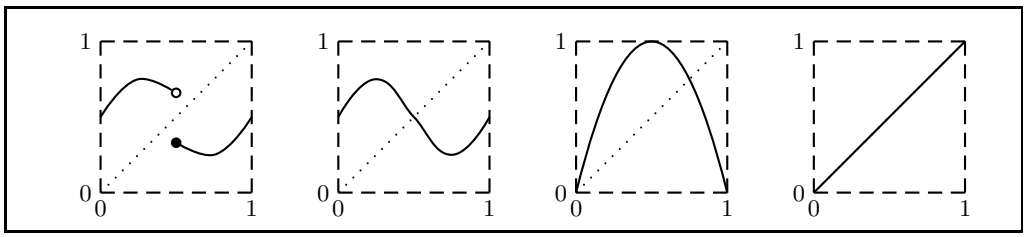


Figure 5.1: Functions on the unit interval with zero, one, two, and an infinite number of fixed points.

- If f is the identity function $\lambda n\,.\,n$, then $(g_{dbl}\ f)$ is the function that increments positive numbers and returns 0 for 0:

$$\lambda n\,.\,\textbf{if}\ (n=0)\ \textbf{then}\ 0\ \textbf{else}\ (n+1)\ \textbf{fi}$$

- If f is the function $\lambda n\,.\,\left((n+1)^2-2\right)$ then $(g_{dbl}\ f)$ is the function $\lambda n\,.\,n^2$
- If f is a doubling function, then $(g_{dbl}\ f)$ is also the doubling function, so the doubling function is a fixed point of g_{dbl}. Indeed, it is the only fixed point of g_{dbl}.

Since generating functions $D \to D$ correspond to recursive definitions, their fixed points have all the properties of solutions to recursive definitions. In particular, such a function may have zero, one, or more fixed points, and the existence and character of fixed points depends on the details of the function and the nature of the domain D.

5.1.3 The Iterative Fixed Point Technique

Above, we saw that recursive definitions can make sense over any domain. However, the methods we used to find and/or verify solutions in the examples were rather ad hoc. In the case of numeric definitions, there are many familiar techniques for manipulating equations to find solutions. Are there any techniques that will help us solve recursive definitions over more general domains?

There is a class of recursive definitions for which an **iterative fixed point technique** will find a distinguished solution of the definition. This technique finds a unique fixed point to the generating function encoding the recursive definition. The iterative fixed point technique is motivated by the observation that it is often possible to find a fixed point for a generating function by iterating the function starting with an appropriate initial value.

As a graphical example of the iteration technique, consider a transformation T on two-dimensional line drawings that is the sequential composition of the following three steps:

1. Rotate the drawing 90 degrees counter-clockwise about the origin.

2. Translate the drawing right by one unit.

3. Add a line from (0,0) to (0,1).

Figure 5.2 shows what happens when T is iterated starting with the empty drawing. Each of the first four applications of T adds a new line until the unit square is produced. Subsequent applications of T do not modify the square; it is a fixed point of T.

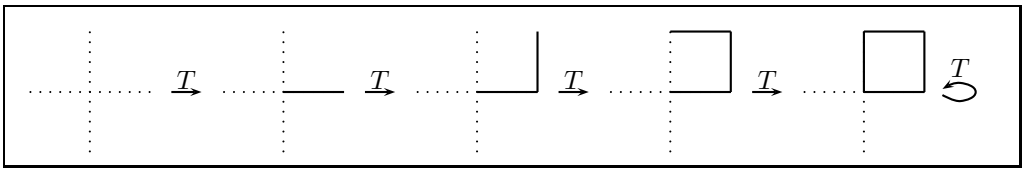


Figure 5.2: Iterating the transformation T starting with an empty line drawing leads to a fixed point in four steps.

In the line drawing example, a fixed point is reached after four iterations of the transformation. Often, iterating a generating function does not yield a fixed point in a finite number of steps, but only approaches one in the limit. A classic numerical example is finding square roots. The square root of a non-negative rational number n is a solution of the recursive definition

$$x \;=\; \frac{x + \frac{n}{x}}{2}.$$

Iterating the generating function for this definition starting with n yields a sequence of approximations that converge to $\sqrt{n}$. For example, for $n = 3$ the generating function is

$$g_{sqrt3} \;:\; Rat \to Rat \;=\; \lambda q \,.\, \frac{q + \frac{3}{q}}{2}$$

and the first few iteration steps are:

$$\begin{aligned}
\left(g^{0}_{sqrt3}\ 3\right) &= 3 \\
\left(g^{1}_{sqrt3}\ 3\right) &= 2 \\
\left(g^{2}_{sqrt3}\ 3\right) &= \frac{7}{4} = 1.75 \\
\left(g^{3}_{sqrt3}\ 3\right) &= \frac{97}{56} \approx 1.7321428571428572 \\
\left(g^{4}_{sqrt3}\ 3\right) &= \frac{18817}{10864} \approx 1.7320508100147276 \\
&\vdots
\end{aligned}$$

Since $\sqrt{3}$ is not a rational number, the fixed point clearly cannot be reached in a finite number of steps, but it is approached as the limit of the sequence of approximations.

Even in non-numeric domains, generating functions can produce sequences of values approaching a limiting fixed point. For example, consider the following recursive definition of the even natural numbers:

$$evens = \{0\} \cup \{(n+2) \mid n \in evens\}.$$

The associated generating function is

$$g_{evens} : \mathcal{P}(Nat) \to \mathcal{P}(Nat) = \lambda s\,.\,\{0\} \cup \{(n+2) \mid n \in s\},$$

where s ranges over the powerset of *Nat*. Then iterating g_{evens} starting with the empty set yields a sequence of sets that approaches the set of even numbers in the limit:

$$\begin{aligned}
\left(g^{0}_{evens}\ \{\}\right) &= \{\} \\
\left(g^{1}_{evens}\ \{\}\right) &= \{0\} \\
\left(g^{2}_{evens}\ \{\}\right) &= \{0,\ 2\} \\
\left(g^{3}_{evens}\ \{\}\right) &= \{0,\ 2,\ 4\} \\
\left(g^{4}_{evens}\ \{\}\right) &= \{0,\ 2,\ 4,\ 6\} \\
&\vdots
\end{aligned}$$

The above examples of the iterative fixed point technique involve different domains but exhibit a common structure. In each case, the generating function maps an approximation of the fixed point into a better approximation, where the notion of "better" depends on the details of the function:

- In the line drawing example, picture b is better than picture a if b contains at least as many lines of the unit square as a.
- In the square root example, number b is a better approximation to $\sqrt{n}$ than number a if $|b^2 - n| \leq |a^2 - n|$.
- In the even number example, set b is better than set a if $a \subseteq b$.

Moreover, in each of the examples, the sequence of approximations produced by the generating functions converges to a fixed point in the limit. This doesn't necessarily follow from the fact that each approximation is better than the previous one. For example, each element of the series $0, 0.9, 0.99, 0.999, \ldots$ is closer to $\sqrt{2}$ than the previous element, but the series converges to 1, not to $\sqrt{2}$. The notion of approaching a limiting value is central to the iterative fixed point technique.

The basic structure of the iterative fixed point technique is depicted in Figure 5.3. The generating function $g : D \to D$ is defined over a domain D whose values are assumed to be ordered by their information content. A line connects two values when the lower value is an approximation to the higher value. That is, the higher value contains all the information of the lower value plus some extra information. What counts as "information" and "approximation" depends on the problem domain. When values are sets, for instance, a line from a up to b might indicate that $a \subseteq b$.

In the iterative fixed point technique, iteratively applying g from an appropriate starting value d_0 yields a sequence of values with increasing information content. Intuitively, iterative applications of g climb up through the ordered values by refining the information of successive approximations. If this process reaches a value d_i such that $d_i = (g\ d_i)$, then the fixed point d_i has been found. If this process never actually reaches a fixed point, it should at least approach a fixed point as a limiting value.

We emphasize that the iterative fixed point technique does not work for every generating function. It depends on the details of the domain D, the generating function $g : D \to D$, and the the starting point d_0. The technique must certainly fail for generating functions that have no fixed points. Even when a generating function has a fixed point, the iterative technique won't necessarily find it. For example, iterating the generating function for $n = \frac{3}{n}$ starting with any non-zero rational number q yields an alternating sequence q, $\frac{3}{q}$, q, $\frac{3}{q}$, $\ldots$ that never gets any closer to the fixed point $\sqrt{3}$. presented earlier in this section. As shown in Figure 5.4, if we start with an "X" in the upper right quadrant, the iterative fixed point technique yields a different fixed point than when we start with an empty picture. Figure 5.5 shows an example in which the technique does not

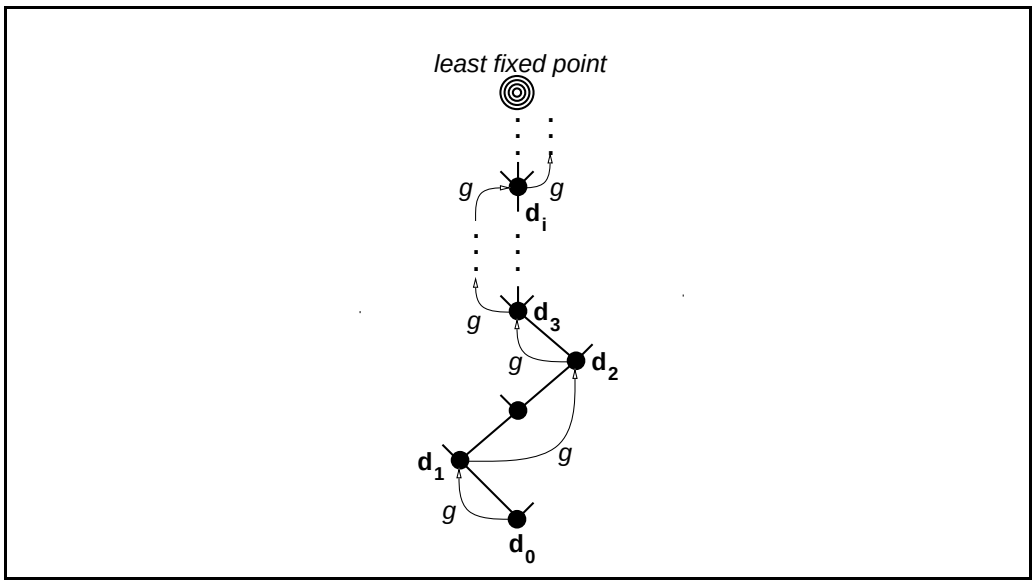


Figure 5.3: The "game board" for the iterative fixed point technique.

find a fixed point of T for an initial picture. Instead, it eventually cycles between four distinct pictures.

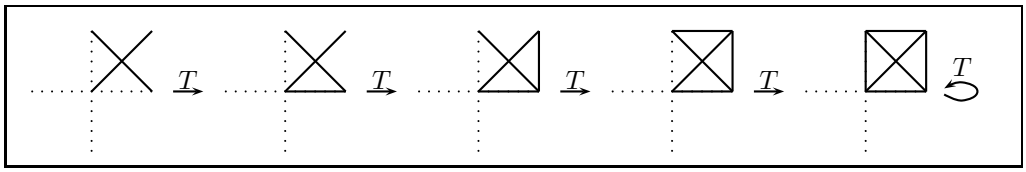


Figure 5.4: A different initial picture can lead to a different fixed point for the picture transformation T.

In the next section, we describe an important class of generating functions that are *guaranteed* to have a fixed point. A fixed point of these functions can be found by applying the iterative fixed point technique starting with a special informationless element called **bottom**. Such functions may have more than one fixed point, but the one found by iterating from bottom has less information than all the others — it is the **least fixed point**. We will choose this distinguished fixed point as *the* solution of the associated recursive definition. This solution matches our operational intuitions about what solution the computer will find when the recursive definition is expressed as a program. We are guaranteed to be able to solve any recursive definition whose generating function is in this

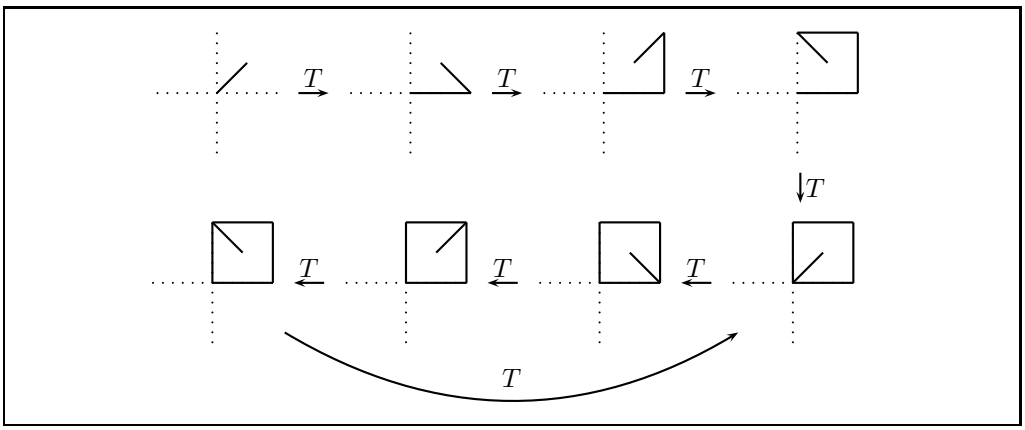


Figure 5.5: An example in which the iterative fixed point technique cannot find a fixed point of the picture transformation T for a non-empty initial picture.

special class.

▷ **Exercise 5.1** Above, we showed two fixed points of the picture transformation T.

a. Draw a third line drawing that is a fixed point of T.

b. How many fixed points does T have?

c. Characterize all the fixed points of T. That is, what properties must a picture have in order to be a fixed point of T?

d. Figure 5.5 shows an initial picture for which the iterative technique finds a cycle of four distinct pictures related by T rather than a fixed point of T. Give an initial picture for which the iterative technique finds a cycle containing only two distinct picture related by T. In the case of T, can the iterative technique find cycles of pictures with periods other than 1, 2, and 4? ◁

▷ **Exercise 5.2** For each of the following classes of functions from the unit interval to itself, indicate the minimum and maximum number of fixed points of functions in the class.

a. constant functions (i.e., functions of the form $\lambda x\,.\,a$);

b. linear functions (i.e., functions of the form $\lambda x\,.\,ax + b$);

c. quadratic functions (i.e., functions of the form $\lambda x\,.\,ax^2 + bx + c$);

d. continuous functions (i.e., functions whose graph is an unbroken curve);

e. non-decreasing functions (i.e., functions f for which $a \leq b$ implies $(f\ \ a) \leq (f\ \ b)$.

f. non-increasing functions (i.e., functions f for which $a \leq b$ implies $(f\ \ a) \geq (f\ \ b)$.

◁

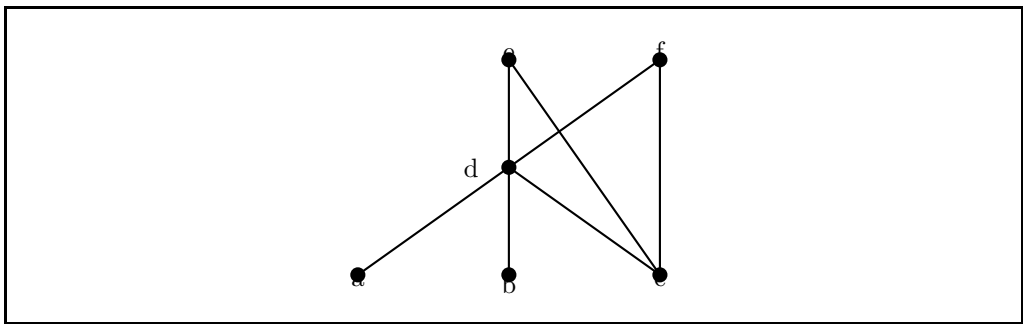


Figure 5.6: A Hasse diagram for the partial order PO.

5.2 Fixed Point Machinery

In this section we (1) present the mathematical machinery for defining a class of functions for which a distinguished fixed point always exists and (2) illustrate the use of this machinery via several examples.

5.2.1 Partial Orders

A **partial order** is a pair $\langle D, \sqsubseteq \rangle$ of a domain D and a relation $\sqsubseteq$ that is reflexive, transitive, and anti-symmetric. A relation is **anti-symmetric** if $a \sqsubseteq b$ and $b \sqsubseteq a$ together imply $a = b$. The notation $a \sqsubseteq b$ is pronounced "a is weaker than b" or "b is stronger than a." Later, we shall be ordering elements by information content, so we will also pronounce $a \sqsubseteq b$ as "a approximates b." When the relation $\sqsubseteq$ is understood from context, it is common to refer to the partial order $\langle D, \sqsubseteq \rangle$ as D.

Partial orders are commonly depicted by **Hasse diagrams** in which elements (represented by points) are connected by lines. In such a diagram, $a \sqsubseteq b$ if and only if there is a path from the point representing a to the point representing b such that each link of the path goes upward on the page. For example, Figure 5.6 shows the Hasse diagram for the partial order PO on six symbols whose relation is defined by the following graph:

$$\{\langle \mathtt{a},\mathtt{a}\rangle, \langle \mathtt{a},\mathtt{d}\rangle, \langle \mathtt{a},\mathtt{e}\rangle, \langle \mathtt{a},\mathtt{f}\rangle, \ \langle \mathtt{b},\mathtt{b}\rangle, \langle \mathtt{b},\mathtt{d}\rangle, \langle \mathtt{b},\mathtt{e}\rangle, \langle \mathtt{b},\mathtt{f}\rangle,$$
$$\langle \mathtt{c},\mathtt{c}\rangle, \langle \mathtt{c},\mathtt{e}\rangle, \langle \mathtt{c},\mathtt{f}\rangle, \ \langle \mathtt{d},\mathtt{d}\rangle, \langle \mathtt{d},\mathtt{e}\rangle, \langle \mathtt{d},\mathtt{f}\rangle, \ \langle \mathtt{e},\mathtt{e}\rangle, \ \langle \mathtt{f},\mathtt{f}\rangle\}.$$

Elements of a partial order are not necessarily related. Two elements of a partial order that are unrelated by $\sqsubseteq$ are said to be **incomparable**. For example, here is a listing of all the pairs of incomparable elements in PO: $\langle \mathtt{a},\mathtt{b}\rangle$, $\langle \mathtt{a},\mathtt{c}\rangle$, $\langle \mathtt{b},\mathtt{c}\rangle$, $\langle \mathtt{c},\mathtt{d}\rangle$, and $\langle \mathtt{e},\mathtt{f}\rangle$.

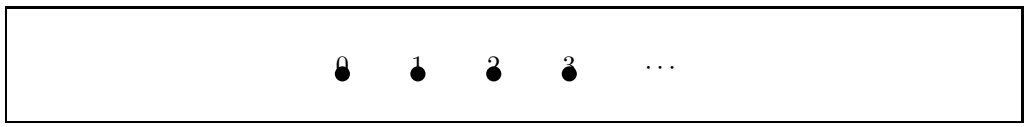


Figure 5.7: The domain *Nat* is assumed to have the discrete ordering.

An **upper bound** of a subset X of a partial order D is an element $u \in D$ that is stronger than every element of X; i.e., for every x in X, $x \sqsubseteq u$. In PO, the subset $\{\mathtt{a}, \mathtt{b}\}$ has upper bounds $\mathtt{d}$, $\mathtt{e}$, and $\mathtt{f}$; the subset $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ has upper bounds $\mathtt{e}$ and $\mathtt{f}$; and the subset $\{\mathtt{e}, \mathtt{f}\}$ has no upper bounds. The **least upper bound (lub[5])** of a subset X of D, written $\bigsqcup_D X$, is the upper bound of X that is weaker than every other upper bound of X; such an element may not exist. In PO, the lub of $\{\mathtt{a}, \mathtt{b}\}$ is $\mathtt{d}$, but neither $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ nor $\{\mathtt{e}, \mathtt{f}\}$ has a lub. There are symmetric notions of **lower bound** and **greatest lower bound (glb[6])**, but our fixed point machinery will mainly use upper bounds.

An element that is weaker than all other elements in a partial order D is called the **bottom** element and is denoted $\bot_D$. Symmetrically, an element that is stronger that all other elements in D is the **top** element (written $\top_D$). Bottom and top elements do not necessarily exist. For example, PO has neither.

Any partial order D can be **lifted** to another partial order $D_\bot$ that has all the elements and orderings of D, but includes a new element $\bot_{D_\bot}$ that is weaker than all elements of D. If D already has a bottom element $\bot_D$, then $\bot_D$ and $\bot_{D_\bot}$ are distinct, with $\bot_{D_\bot}$ being the weaker of the two. Symmetrically, the notation $D^\top$ designates the result of extending D with a new top element.

A **discrete** partial order is one in which every pair of elements is incomparable. By default, we will assume that primitive semantic domains have the discrete ordering. For example, Figure 5.7 depicts the discrete ordering for *Nat*. In this partial order, numbers are not ordered by their value, but by their information content. Each number approximates only itself.

A **flat** partial order D is a lifted discrete partial order. Flat partial orders will play an important role in our treatment of semantic domains. Figure 5.8 depicts the flat partial order $Nat_\bot$ of natural numbers. Note that $\bot_{Nat_\bot}$ acts as an "unknown natural number" that approximates every natural number.

A **total order** is a partial order in which every two elements are related (i.e., no two elements are incomparable). For example, the natural numbers under the traditional value-based ordering form a total order called ω. The elements of a total order can be arranged in a vertical line in a Hasse diagram (see Figure 5.9).

[5]The pronunciation of "lub" rhymes with "club."
[6]"glb" is pronounced "glub."

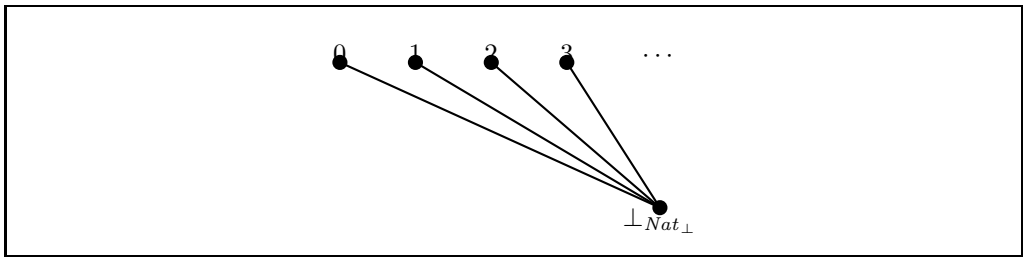


Figure 5.8: The flat partial order $Nat_\perp$.

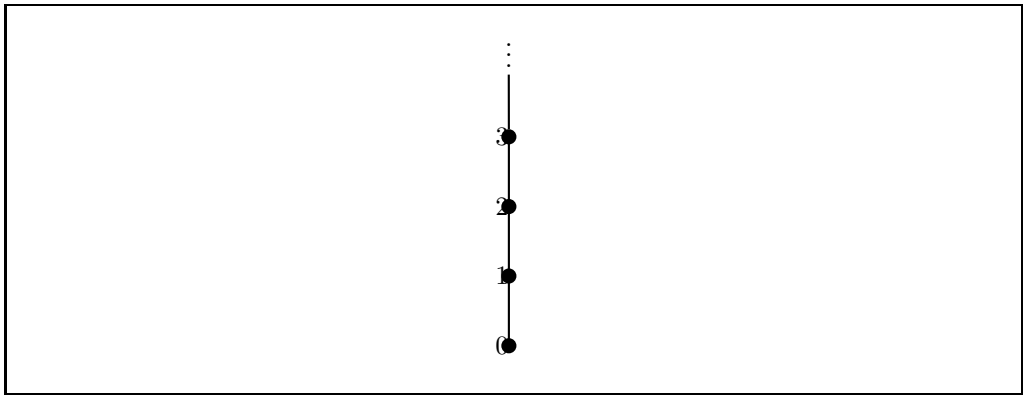


Figure 5.9: The partial order ω of natural numbers under the traditional value-based ordering.

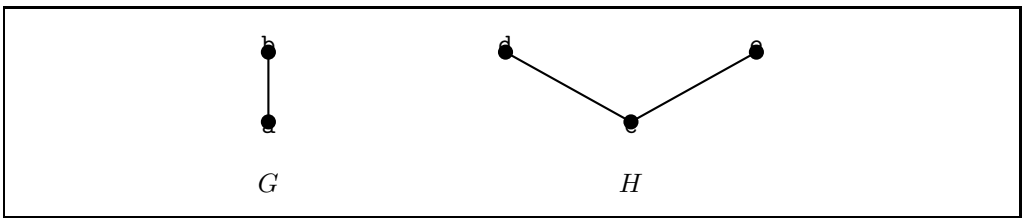


Figure 5.10: Two simple partial orders.

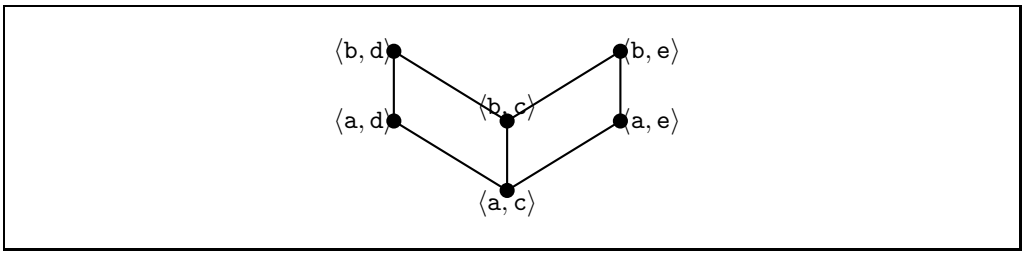


Figure 5.11: The product partial order $G \times H$.

A **chain** is a totally-ordered, non-empty subset of a partial order. The chains of PO include $\{\mathtt{a}, \mathtt{d}, \mathtt{e}\}$, $\{\mathtt{c}, \mathtt{f}\}$, $\{\mathtt{b}, \mathtt{f}\}$, and $\{\mathtt{d}\}$. In $Nat_\perp$, the only chains are (1) singleton sets and (2) doubleton sets containing $\perp_{Nat_\perp}$ and a natural number.

Given partially ordered domains, we would like to define orderings on product, sum, sequence, and function domains such that the resulting domains are also partially ordered. That way, we will be able to view all our semantic domains as partial orders. In the following definitions, assume that D and E are arbitrary partial orders ordered by $\sqsubseteq_D$ and $\sqsubseteq_E$, respectively. We will illustrate the definitions with examples involving the two concrete partial orders G and H in Figure 5.10.

5.2.1.1 Product Domains

$D \times E$ is a partial order under the following ordering:

$$\langle d_1, e_1\rangle \sqsubseteq_{D\times E} \langle d_2, e_2\rangle \text{ iff } d_1 \sqsubseteq_D d_2 \text{ and } e_1 \sqsubseteq_E e_2.$$

The partial order $G \times H$ is depicted in Figure 5.11. Note how the Hasse diagram for $G \times H$ is visually the product of the Hasse diagrams for G and H. $G \times H$ results from making a copy of G at every point of H (or, symmetrically, making a copy of H at every point of G) and adding the extra lines specified by the ordering.

5.2.1.2 Sum Domains

$D + E$ is a partial order under the following ordering:

$$\begin{aligned}(Inj\,1_{D,E}\ d_1) \sqsubseteq_{D+E} (Inj\,1_{D,E}\ d_2) \ \text{ iff } \ d_1 \sqsubseteq_D d_2\\ (Inj\,2_{D,E}\ e_1) \sqsubseteq_{D+E} (Inj\,2_{D,E}\ e_2) \ \text{ iff } \ e_1 \sqsubseteq_E e_2.\end{aligned}$$

This ordering preserves the order between elements of the same summand, but treats elements from different summands as incomparable. The Hasse diagram for a sum partial order is simply the juxtaposition of the diagrams for the summands (see Figure 5.12).

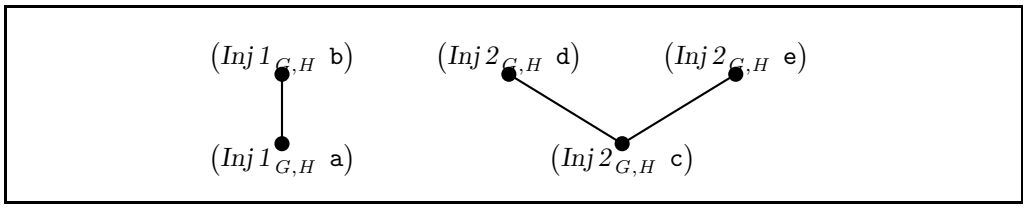


Figure 5.12: The sum partial order $G + H$.

5.2.1.3 Function Domains

$D \to E$ is a partial order under the following ordering:

$$f_1 \sqsubseteq_{D\to E} f_2 \text{ iff, for all } d \text{ in } D, (f_1\ d) \sqsubseteq_E (f_2\ d).$$

Consider using this ordering on the elements of $G \to H$. As usual, a total function from G to H can be represented by a graph of input/output pairs, but here we employ a more compact notation in which such a function is represented as a pair of the elements that `a` and `b` map to, respectively. Thus, the function with graph $\{\langle \mathtt{a}, \mathtt{c}\rangle, \langle \mathtt{b}, \mathtt{d}\rangle\}$ can be abbreviated as $\langle \mathtt{c}, \mathtt{d}\rangle$. Using this notation, the partial order $G \to H$ is isomorphic[7] to the partial order $H \times H$ (see Figure 5.13).

This sort of isomorphism holds whenever D is a finite domain. That is, if D has n elements, then $D \to E$ is isomorphic to E^n.

[7]Informally, two partial orders are isomorphic if their Hasse diagrams can be rearranged to have the same shape (ignoring the labels on the vertices). Formally, two partial orders A and B are isomorphic if there is a bijective function $f : A \to B$ that preserves ordering in both directions. That is, $a \sqsubseteq_A a'$ implies $(f\ a) \sqsubseteq_B (f\ a')$ and $b \sqsubseteq_B b'$ implies $(f^{-1}\ b) \sqsubseteq_A (f^{-1}\ b')$.

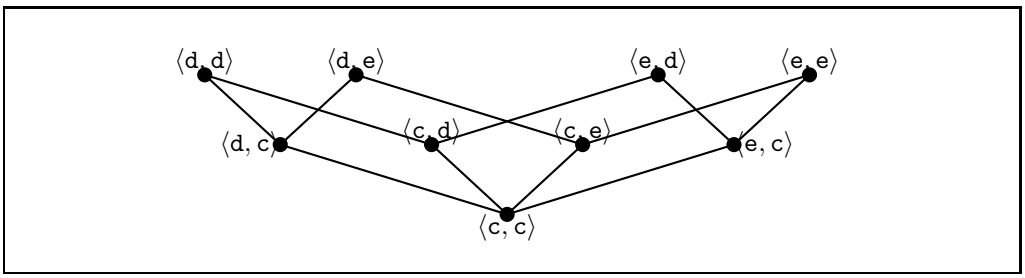


Figure 5.13: The function partial order $G \rightarrow H$. Each pair $\langle x, y\rangle$ is shorthand for a function with a graph $\{\langle \mathtt{a}, x\rangle, \langle \mathtt{b}, y\rangle\}$.

5.2.1.4 Sequence Domains

There are two common ways to order the elements of D^*. These differ in whether sequence elements of different lengths are comparable.

- Under the **prefix ordering**,

$$[d_1,\ d_2,\ \ldots,\ d_k] \sqsubseteq_{D^*} [d_1{}',\ d_2{}',\ \ldots,\ d_l{}']$$
$$\text{iff } k \leq l \text{ and } d_i \sqsubseteq_D d_i{}' \text{ for all } 1 \leq i \leq k$$

  If D is a discrete domain, this implies that a sequence s_1 is weaker than s_2 if s_1 is a prefix of s_2 — i.e., $s_2 = s_1 \ @\ s'$ for some sequence s'.

  As an example, suppose that *Bit* is the discrete partial order of the binary digits `0` and `1`. Then *Bit** under the prefix order is isomorphic to the partial order of binary numerals shown in Figure 5.14. (For example, the numeral `110` corresponds to the sequence $[\mathtt{1}, \mathtt{1}, \mathtt{0}]$.) This partial order is an infinite binary tree rooted at the empty sequence. Each element of the tree can be viewed as an approximation to all of the elements of the subtree rooted at it. For example, `110` is an approximation to `1100`, `1101`, `11000`, `11001`, `11010`, etc. In computational terms, this notion of approximation corresponds to the behavior of a computation process that produces its answer by printing out a string of `0`s and `1`s from left to right, one character at a time. At any point in time, the characters already printed are the current approximation to the final string that will be produced by the process.

  Note that if D has some non-trivial ordering relations, i.e., D is not a discrete domain, the prefix ordering of D^* is more complex than a simple tree.

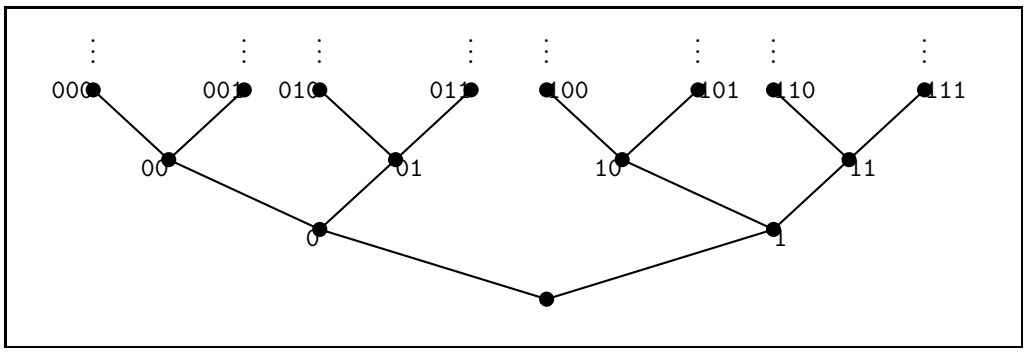


Figure 5.14: The sequence partial order Bit^* under the prefix ordering.

- Under the **sum-of-products ordering**, D^* is treated as isomorphic to the infinite sum of products

$$D^0 + D^1 + D^2 + D^3 + \cdots$$

As in the prefix ordering, sequences are ordered component-wise by their elements, but the sum-of-products ordering treats sequences of different lengths as incomparable. For example, under the sum-of-products ordering, $Bit_\perp{}^*$ is isomorphic to the partial order depicted in Figure 5.15.

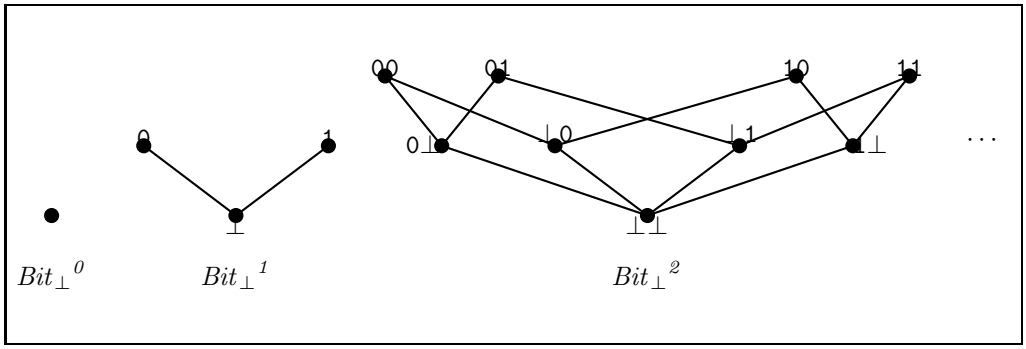


Figure 5.15: The sequence partial order $Bit_\perp{}^*$ under the sum-of-products ordering.

Although we have stated that the above definitions are partial orders, we have not argued that each ordering is in fact reflexive, transitive, and anti-symmetric. We encourage the reader to show that these properties hold for each of the definitions.

The orderings defined above are not the only ways to order compound domains, but they are relatively natural and are useful in many situations. Later, we will refine some of these orderings (particularly in the case of function domains). But, for the most part, these are the orderings that will prove useful for our study of semantic domains.

▷ **Exercise 5.3** Using the partial orders G and H in Figure 5.10, draw a Hasse diagram for each of the following compound partial orders:

a. $G \times G$

b. $H \times H$

c. $G \to G$

d. $H \to H$

e. $H \to G$

f. G^* under the prefix ordering (show the first four levels)

g. H^* under the prefix ordering (show the first four levels)

h. G^* under the sum-of-products ordering (show the first three summands)

i. H^* under the sum-of-products ordering (show the first three summands) ◁

▷ **Exercise 5.4** Suppose that A and B are finite partial orders with the same number of elements, but they are not isomorphic. Partition the following partial orders into equivalence classes based on isomorphism. That is, each class should contain all the partial orders that are isomorphic to each other.

$$\begin{array}{llll} A \times A, & A \times B, & B \times A, & B \times B, \\ A + A, & A + B, & B + A, & B + B, \\ A \to A, & A \to B, & B \to A, & B \to B \end{array}$$ ◁

▷ **Exercise 5.5** Given a discretely ordered domain D, the powerset $\mathcal{P}(D)$ is a partial order under the **subset ordering**:

$$S \sqsubseteq_{\mathcal{P}(D)} S' \text{ if } S \subseteq S'$$

Draw the Hasse diagram for the partial order $\mathcal{P}(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$ under the subset ordering.

If D is a partial order that is not discrete, it turns out that there are many "natural" ways to order the elements of the **powerdomain** $\mathcal{P}(D)$, each of which is useful for different purposes. See [Sch86a] or [GS90] for details. ◁

▷ **Exercise 5.6** For each ordering on a compound domain defined above, show that the ordering is indeed a partial order. I.e., show that the orderings defined for product, sum, function, and sequence domains are reflexive, transitive, and anti-symmetric. ◁

5.2.2 Complete Partial Orders (CPOs)

A partial order D is **complete** if every chain in D has a least upper bound in D. The term "complete partial order" is usually abbreviated **CPO**. Intuitively, completeness means that any sequence of elements visited on an upward path through a Hasse diagram must converge to a limit. Completeness is important because it guarantees that the iterative fixed point technique converges to a limiting value.

Here are some examples of CPOs:

- Any partial order with a finite number of elements is a CPO because every chain is finite and necessarily contains its lub. PO, G, and H from the previous section are all finite CPOs.

- Any flat partial order is a CPO because every chain has at most two elements, the stronger of which must be the lub. $Nat_\perp$ is a CPO with an infinite number of elements.

- $\mathcal{P}(Nat)$ is a CPO in which the elements (each of which is a subset of the naturals) are ordered by subset inclusion (see Exercise 5.5). It is complete because the lub of every chain C is the (possibly infinite) union of the elements of C. Unlike the previous examples of CPOs, this is one in which a chain may be infinite and not contain its own lub. Consider the chain C with elements c_i, where c_i is defined to be $\{n \mid n \leq i, n : Nat\}$ Then:

$$\bigsqcup_{\mathcal{P}(Nat)} C = \bigcup\{\{0\}, \{0,1\}, \{0,1,2\}, \ldots\} = Nat$$

  The lub of C is the entire set of natural numbers, but no individual c_i is equal to this set.

- The unit interval under the usual ordering of real numbers is a CPO. It is complete because the construction of the reals guarantees that it contains the least upper bound of every subset of the interval. The unit interval is another CPO in which chains do not necessarily contain their own lubs. For example, the set of all rational numbers less than $\sqrt{.5}$ does not contain $\sqrt{.5}$.

- The partial functions from Nat to Nat (denoted $Nat \rightharpoonup Nat$) form a CPO. Recall that a partial function can be represented by a graph of input/output pairs. So the function that is undefined everywhere is represented by $\{\}$, the function that returns 23 given 17 and is elsewhere

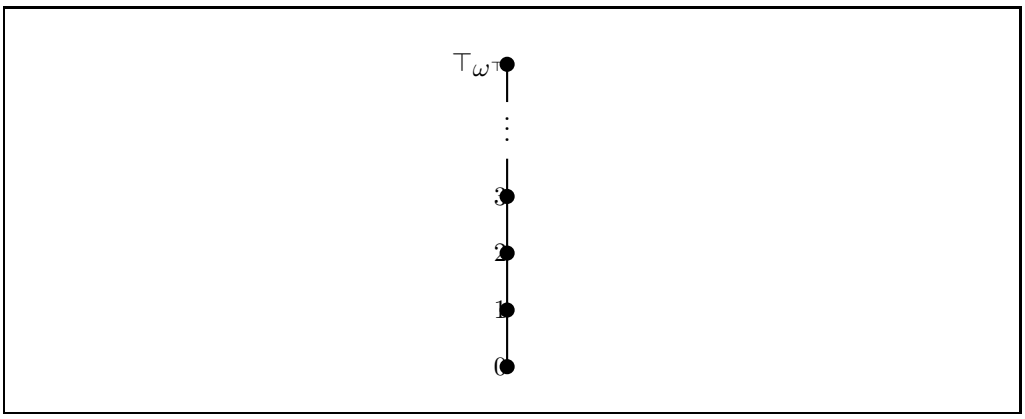


Figure 5.16: The partial order $\omega^\top$ is the partial order ω of natural numbers extended with a largest element $\top_{\omega^\top}$.

undefined is represented by $\{\langle 17, 23\rangle\}$, and so on. The ordering of elements in this CPO is just subset inclusion on the graphs of the functions. It is complete for the same reason that $\mathcal{P}(Nat)$ is complete.

It is worthwhile to consider examples of partial orders that are *not* CPOs:

- The total order ω depicted in Figure 5.9 is not a CPO because the chain consisting of the entire set has no least upper bound (i.e., there is no largest natural number). This partial order can be turned into a CPO $\omega^\top$ by extending it with a top element $\top_{\omega^\top}$ that by definition is larger than every natural number (see Figure 5.16.)

- The partial order of rational numbers (under the usual ordering) between 0 and 1, inclusive, is not complete because it does not contain irrational numbers like $\sqrt{\frac{1}{2}}$. It can be made complete by extending it with the irrationals between 0 and 1; this results in the unit interval $[0, 1]$.

- The partial order of sequences Bit^* under the prefix ordering is not a CPO. By definition, D^* is the set of *finite* sequences whose elements are taken from D. But the chain $\{[\,],\ [\mathtt{1}],\ [\mathtt{1},\mathtt{1}],\ [\mathtt{1},\mathtt{1},\mathtt{1}],\ \ldots\}$ has as its lub an infinite sequence of 1s, which is not an element of Bit^*. To make this partial order complete, it is necessary to extend it with the set of infinite sequences over 0 and 1, written Bit^∞. So the set of strings $Bit^* \cup Bit^\infty$ under the prefix ordering is a CPO.

Generalizing Bit^∞, we introduce the notation D^∞ to denote the set of all infinite sequences whose elements are taken from the domain D. We also intro-

duce the notation $\overline{D^*}$ to stand for $D^* \cup D^\infty$ under the prefix ordering. (The overbar notation is commonly used to designate the **completion** of a set, which adds to a set all of its limit points.)

As with partial orders, we are interested in combination properties of CPOs. As indicated by the following facts, we can use $\perp$, $\times$, $+$, $\rightarrow$, and * to build new CPOs out of existing CPOs. Suppose that D and E are CPOs. Then:

- $D_\perp$ is a CPO;
- $D \times E$ is a CPO under the partial order for products;
- $D + E$ is a CPO under the partial order for sums;
- $D \rightarrow E$ is a CPO under the partial order for functions;
- D^* is a CPO under the sum-of-products ordering for sequences;
- $\overline{D^*}$ is a CPO under the prefix ordering for sequences.

▷ **Exercise 5.7** For each of the compound CPOs described above, show that the compound partial order is indeed complete. That is, show that the completeness property of D and E implies that each chain of the compound domain has a lub in the compound domain. ◁

5.2.3 Pointedness

A partial order is **pointed** if it has a bottom element. Pointedness is important because the bottom element of a CPO is the natural place for the iterative fixed point technique to start. Here are some of the pointed CPOs we have studied, listed with their bottom elements:

- G, bottom $=$ `a`;
- H, bottom $=$ `c`;
- $Nat_\perp$, bottom $= \perp_{Nat}$;
- $\mathcal{P}(Nat)$, bottom $= \{\}$;
- $[0, 1]$, bottom $= 0$;
- $Nat \rightharpoonup Nat$, bottom $=$ the function whose graph is $\{\}$;
- $\omega^\top$, bottom $= 0$;

- Bit*, bottom $= [\,]$.

CPOs that we have studied that are *not* pointed include PO, $G + H$, and ${Bit_\perp}^*$ under the sum-of-products ordering.

In the iterative fixed point technique, the bottom element of a pointed CPO is treated as the element with least information — the "worst" approximation to the desired value. For example, $\perp_{Nat_\perp}$ is the unknown natural number, $[\,]$ is a (bad) approximation to any sequence of `0`s and `1`s, and $\{\}$ is a (bad) approximation to the graph of any partial function from Nat to Nat.

In computational terms, the bottom element of a CPO can informally be viewed as representing a process that **diverges** (i.e., gets caught in an infinite loop). For example, a procedure that returns a boolean for even numbers but diverges on odd numbers can be modeled as an element of the domain $Int \to Bool_\perp$ that maps every odd number to $\perp_{Bool_\perp}$.

Pointed CPOs are commonly used to encode partial functions as total functions. Any partial function f in $D \rightharpoonup E$ can be represented as a total function $f\,'$ in $D \to E_\perp$ by having $f\,'$ map to $\perp_{E_\perp}$ every element $d : D$ on which f is undefined. For example, the partial function in $PO \rightharpoonup PO$ with graph

$$\{\langle \mathtt{a}, \mathtt{d}\rangle,\ \langle \mathtt{c}, \mathtt{b}\rangle,\ \langle \mathtt{f}, \mathtt{f}\rangle\}.$$

can be represented as the total function in $PO \to PO_\perp$ with graph

$$\{\langle \mathtt{a}, \mathtt{d}\rangle,\ \langle \mathtt{b}, \perp_{PO_\perp}\rangle,\ \langle \mathtt{c}, \mathtt{b}\rangle,\ \langle \mathtt{d}, \perp_{PO_\perp}\rangle,\ \langle \mathtt{e}, \perp_{PO_\perp}\rangle,\ \langle \mathtt{f}, \mathtt{f}\rangle\}$$

Because of the isomorphism between $D \rightharpoonup E$ and $D \to E_\perp$, we casually perform implicit conversions between the two representations.

The following are handy facts about the pointedness of partial orders constructed out of parts. Suppose that D and E are arbitrary partial orders (not necessarily pointed). Then:

- $D_\perp$ is pointed.
- $D \times E$ is pointed if D and E are pointed.
- $D + E$ is never pointed.
- $D \to E$ is pointed if E is pointed.
- D^* under the sum-of-products ordering is never pointed.
- $\overline{D^*}$ and D^* under the prefix ordering are pointed.

Unpointed compound domains like $D + E$ and D^* under the sum-of-products ordering can always be made pointed by lifting them with a new bottom element or by coalescing their bottom elements if they are pointed (see Exercise 5.9).

▷ **Exercise 5.8** Prove each of the facts about pointedness claimed above. ◁

▷ **Exercise 5.9** The **smash sum** (also known as **coalesced sum**) of two pointed partial orders D and E, written $D \oplus E$, consists of the elements

$$\{\bot_{D\oplus E}\} \cup \{\big(Inj\,1_{D,E}\ d\big) \mid d \in (D - \bot_D)\} \cup \{\big(Inj\,2_{D,E}\ e\big) \mid e \in (E - \bot_E)\},$$

where $\bot_{D\oplus E}$ is a single new bottom element that combines the bottom elements $\bot_D$ and $\bot_E$. $D \oplus E$ is a partial order under the following ordering:

$$\bot_{D\oplus E} \sqsubseteq_{D\oplus E} x \quad \text{for all} \quad x \in D \oplus E$$

$$\big(Inj\,1_{D,E}\ d_1\big) \sqsubseteq_{D\oplus E} \big(Inj\,1_{D,E}\ d_2\big) \quad \text{iff} \quad d_1, d_2 \in (D - \bot_D) \quad \text{and} \quad d_1 \sqsubseteq_D d_2$$

$$\big(Inj\,2_{D,E}\ e_1\big) \sqsubseteq_{D\oplus E} \big(Inj\,2_{D,E}\ e_2\big) \quad \text{iff} \quad e_1, e_2 \in (E - \bot_E) \quad \text{and} \quad e_1 \sqsubseteq_E e_2.$$

a. Using the CPOs G and H from Figure 5.10, draw a Hasse diagram for the partial order $G \oplus H$.

b. If D and E are CPOs, show that $D \oplus E$ is a CPO.

c. What benefit does $D \oplus E$ have over $D + E$.

d. Suppose that D is a pointed CPO. Extend the notion of smash sum to a **smash sequence** $D^{\circledast}$ such that $D^{\circledast}$ is a pointed CPO under an ordering analogous to the sum-of-product ordering. What does $Bit_{\bot}{}^{\circledast}$ look like? ◁

5.2.4 Monotonicity and Continuity

Suppose that $f : D \to E$, where D and E are CPOs (not necessarily pointed). Then

- f is **monotonic** if $d_1 \sqsubseteq_D d_2$ implies $(f\ \ d_1) \sqsubseteq_E (f\ \ d_2)$.
- f is **continuous** if, for all chains C in D, $(f\ \ (\bigsqcup_D C)) = \bigsqcup_E \{(f\ \ c) \mid c \in C\}$.

A monotonic function preserves order between CPOs, while a continuous function preserves limits. In the iterative fixed point technique, monotonicity is important because when $f : D \to D$ is monotonic, the set of values

$$\{\bot,\ (f\ \ \bot),\ (f\ \ (f\ \ \bot)),\ (f\ \ (f\ \ (f\ \ \bot))),\ \ldots\}$$

is guaranteed to form a chain. Continuity guarantees that this chain approaches a limit.

As an example of these properties, consider the CPO of functions $G \to H$ depicted in Figure 5.13. Any function represented by the pair $\langle x, y \rangle$[8] is monotonic if and only if $x \sqsubseteq y$. Although there are $3^2 = 9$ total functions from G to H, only five of these are monotonic:

$$\{\langle \mathtt{c}, \mathtt{c} \rangle,\ \langle \mathtt{c}, \mathtt{d} \rangle,\ \langle \mathtt{d}, \mathtt{d} \rangle,\ \langle \mathtt{c}, \mathtt{e} \rangle,\ \langle \mathtt{e}, \mathtt{e} \rangle\}$$

The reason that there are fewer monotonic functions than total functions is that choosing the target t for a particular source element s constrains all the source elements stronger than s to map to a target stronger than t. For example, a monotonic function that maps `a` to `e` must necessarily map `b` to `e`. With larger domains, the reduction from total functions to monotonic functions can be more dramatic.

What functions from G to H are continuous? The only non-singleton chain in G is $\{\mathtt{a}, \mathtt{b}\}$. By the definition of continuity, this means that a function $f : D \to E$ is continuous if

$$\left(f\ \left(\bigsqcup_D \{\mathtt{a}, \mathtt{b}\}\right)\right) = \bigsqcup_E \{(f\ \mathtt{a}), (f\ \mathtt{b})\}.$$

In this case, this condition simplifies to $(f\ \mathtt{a}) \sqsubseteq_E (f\ \mathtt{b})$, which is equivalent to saying that f is monotonic. Thus, the continuous functions from G to H are exactly the five monotonic functions listed above.

The relationship between monotonic and continuous functions in this example is more than coincidence. Monotonicity and continuity are closely related, as indicated by the following facts:

- On finite CPOs (and even infinite CPOs with only finite chains), monotonicity implies continuity.
- On *any* CPO, continuity implies monotonicity.

We leave the proof of these facts as exercises.

Although monotonicity and continuity coincide on finite-chain CPOs, monotonicity *does not* imply continuity in general. To see this, consider the following function from $\omega^\top$ to the two-point CPO $Two = \{\bot, \top\}$:

$$\mathit{mon\text{-}not\text{-}con} : \omega^\top \to \mathit{Two} = \lambda n\,.\,\textbf{if}\ (n = \top_{\omega^\top})\ \textbf{then}\ \top\ \textbf{else}\ \bot\ \textbf{fi}$$

(See Figure 5.17 for a depiction of this function.) This function is clearly monotonic, but it is not continuous because on the subset ω of $\omega^\top$,

$$\left(f\ \left(\bigsqcup \omega\right)\right) = (f\ \top_{\omega^\top}) = \top \neq \bot = \bigsqcup_{\mathit{Two}} \{\bot\} = \bigsqcup_{\mathit{Two}} \{(f\ n) \mid n \in \omega\}$$

[8] Recall that in this compact notation from page 170, we simply record the function's value on `a` and `b`, respectively.

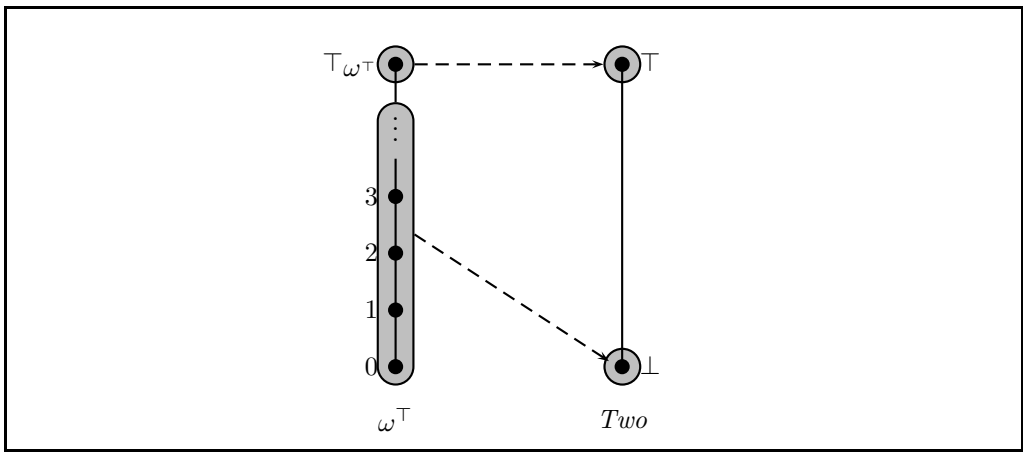


Figure 5.17: An example of a function that is monotonic but not continuous.

An important fact about continuous functions is that the set of continuous functions between CPOs D and E is itself a CPO. For example, Figure 5.18 depicts the CPO of the five continuous functions between G and H. If E is pointed, the function that maps all elements of D to $\bot_E$ is continuous and serves as the bottom element of the continuous function CPO.

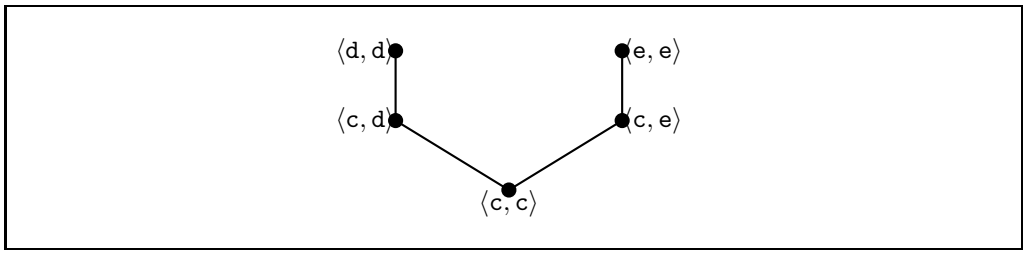


Figure 5.18: The CPO $G \xrightarrow{C} H$ of continuous functions between G and H.

Since the CPO of total functions between D and E and the CPO of continuous functions between D and E are usually distinct, it will be helpful to have a notation that distinguishes them. We will use $D \xrightarrow{T} E$ to designate the CPO of total functions from D to E and $D \xrightarrow{C} E$ to designate the CPO of continuous functions from D to E. It turns out that the CPO of continuous functions is almost always the "right thing" in semantics, so we adopt the convention that, throughout the rest of this text, any unannotated $\rightarrow$ should be interpreted as $\xrightarrow{C}$. We shall use $\xrightarrow{T}$ whenever we wish to discuss set-theoretic functions, and will explicitly use $\xrightarrow{C}$ only when we wish to emphasize the difference between $\xrightarrow{T}$ and $\xrightarrow{C}$.

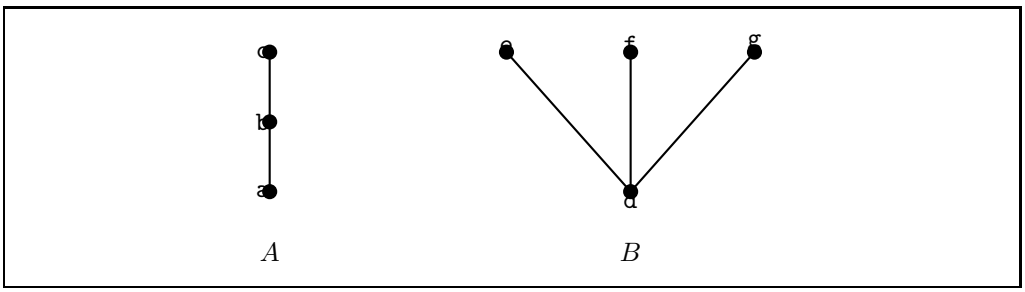


Figure 5.19: CPOs A and B.

▷ **Exercise 5.10** Using the CPOs G and H from Figure 5.10, draw Hasse diagrams for the following CPOs:

a. $G \xrightarrow{C} G$

b. $H \xrightarrow{C} H$

c. $H \xrightarrow{C} G$ ◁

▷ **Exercise 5.11** Consider the CPOs A and B pictured in Figure 5.19. For each of the following functional domains, give the number of the (1) total, (2) monotonic, and (3) continuous functions in the domain:

a. $A \to A$

b. $B \to B$

c. $A \to B$

d. $B \to A$ ◁

▷ **Exercise 5.12**

a. Show that a continuous function between CPOs is necessarily monotonic.

b. Show that a monotonic function must also be continuous if its source is a CPO all of whose chains are finite.

c. Show that if D and E are pointed CPOs then $D \xrightarrow{C} E$ is a pointed CPO. ◁

▷ **Exercise 5.13** This problem considers functions f from $[0,1]$ to itself. We will say that f is *continuous in the CPO sense* if it is a member of $[0,1] \xrightarrow{C} [0,1]$, where $[0,1]$ is assumed to have the traditional ordering. We will say that f is *continuous in the classical sense* if for all x and ϵ there exists a δ such that

$$(f\ \ [x-\delta, x+\delta]) \subseteq [(f\ \ x)-\epsilon, (f\ \ x)+\epsilon].$$

(Here we are abusing the function call notation to designate the image of all of the elements of the interval.)

a. Does classical continuity imply CPO continuity? If so, give a proof; if not, provide a counter-example of a function that is continuous in the classical sense but not in the CPO sense.

b. Does CPO continuity imply classical continuity? If so, give a proof; if not, provide a counter-example of a function that is continuous in the CPO sense but not in the classical sense. ◁

5.2.5 The Least Fixed Point Theorem

Suppose D is a domain and $f : D \to D$. Then $d : D$ is a **fixed point** of f if $(f\ d) = d$. If $\langle D, \sqsubseteq \rangle$ is a partial order, then $d : D$ is the **least fixed point** of f if it is a fixed point of f and $d \sqsubseteq d'$ for every fixed point d' of f.

Everything is now in place to prove the following fixed point theorem:

> **Least Fixed Point Theorem:** If D is a pointed CPO, then a continuous function $f : D \to D$ has a least fixed point $(\mathbf{fix}_D\ f)$ defined by $\bigsqcup_D \{(f^n\ \bot_D) \mid n \geq 0\}$.

Proof:

First we show that the above definition of $(\mathbf{fix}_D\ f)$ is a fixed point of f:

- Since $\bot_D$ is the least element in D, $\bot_D \sqsubseteq (f\ \bot_D)$.
- Since f is monotonic (recall that continuity implies monotonicity), $\bot_D \sqsubseteq (f\ \bot_D)$ implies $(f\ \bot_D) \sqsubseteq (f\ (f\ \bot_D))$. By induction, $(f^n\ \bot_D) \sqsubseteq \left(f^{n+1}\ \bot_D\right)$ for every $n \geq 0$, so $\{(f^n\ \bot_D) \mid n \geq 0\}$ is a chain in D.
- Now,

$$
\begin{array}{lll}
(f\ (\mathbf{fix}_D\ f)) & & \\
\quad = & \left(f\ \bigsqcup_D \{(f^n\ \bot_D) \mid n \geq 0\}\right) & \text{By definition of } \mathbf{fix}_D. \\
\quad = & \bigsqcup_D \{(f\ (f^n\ \bot_D)) \mid n \geq 0\} & \text{By continuity of } f. \\
\quad = & \bigsqcup_D \{(f^n\ \bot_D) \mid n \geq 1\} & \\
\quad = & \bigsqcup_D \{(f^n\ \bot_D) \mid n \geq 0\} & \left(f^0\ \bot_D\right) = \bot_D \text{ can't change lub.} \\
\quad = & (\mathbf{fix}_D\ f) & \text{By definition of } \mathbf{fix}_D.
\end{array}
$$

Thus, $(f\ (\mathbf{fix}_D\ f)) = (\mathbf{fix}_D\ f)$, showing that $(\mathbf{fix}_D\ f)$ is indeed a fixed point of f.

To see that this is the *least* fixed point of f, suppose d' is some other fixed point. Then clearly $\bot_D \sqsubseteq d'$, and by the monotonicity of f, $(f^n\ \bot_D) \sqsubseteq (f^n\ d') = d'$. So d' is an upper bound of the set $S = \{(f^n\ \bot_D) \mid n \geq 0\}$. But then, by the definition of least upper bound, $(\mathbf{fix}_D\ f) = \left(\bigsqcup_D S\right) \sqsubseteq d'$. ◇

We can treat $\mathbf{fix}_D$ as a function of type $(D \to D) \to D$. It turns out that $\mathbf{fix}_D$ is itself a continuous function, and satisfies some other properties that make it "the right thing" for many semantic purposes (see Gunter and Scott [GS90]).

The Least Fixed Point Theorem describes an important class of situations in which fixed points exist, and we shall use it to specify the meaning of various recursive definitions. However, we emphasize that there are many generating functions that have least fixed points but do not satisfy the conditions of the Least Fixed Point Theorem. In these cases, some other means must be used to find the least fixed point.

5.2.6 Fixed Point Examples

Here we present several brief examples of the Least Fixed Point Theorem in action. We have discussed many of these examples informally already but will now show how the fixed point machinery formalizes the intuition underlying the iterative fixed point technique.

5.2.6.1 Sequence Examples

As a first application of the Least Fixed Point Theorem, we consider some examples. In order to model sequences of natural numbers, we will use the domain

$$s \in \mathit{Natseq} = \overline{\mathit{Nat}_\perp}^*.$$

We use the flat domain $\mathit{Nat}_\perp$ instead of Nat to model the elements of a sequence so that there is a distinguished bottom element to which *head* can map the empty sequence. We will assume that $(\mathit{tail}\ [\,]) = [\,]$, though we could alternatively introduce a new bottom element for sequences if we wanted to distinguish $(\mathit{tail}\ [\,])$ from $[\,]$. We use $\overline{\mathit{Nat}_\perp}^*$ rather than $\mathit{Nat}_\perp{}^*$ because the former is a pointed CPO that contains all the limiting values that are missing from the latter. In order to apply the iterative fixed point technique, we will need to assume that *Natseq* has the prefix ordering on sequences rather than the sum-of-products ordering.

The equation $s = (\mathit{cons}\ 3\ (\mathit{cons}\ (1 + (\mathit{head}\ s))\ [\,]))$ has as its associated generating function the following:

$$g_{\mathit{seq1}} : \mathit{Natseq} \to \mathit{Natseq} = \lambda s\,.\,(\mathit{cons}\ 3\ (\mathit{cons}\ (1 + (\mathit{head}\ s))\ [\,]))\,.$$

Natseq is a pointed CPO with bottom element $[\,]$, and it is not hard to show that g_{seq1} is continuous. Thus, the Least Fixed Point Theorem applies, and the

least fixed point can be found by iterating g starting with $[\,]$:

$$\begin{array}{l}(\mathbf{fix}_{Natseq}\ g_{seq1}) \\ \quad = \bigsqcup_{Natseq}\{(g^0_{seq1}\ [\,])\,,\ (g^1_{seq1}\ [\,])\,,\ (g^2_{seq1}\ [\,])\,,\ (g^3_{seq1}\ [\,])\,,\ \ldots\ \} \\ \quad = \bigsqcup_{Natseq}\{[\,],\ [3, \bot_{Nat_\bot}],\ [3,4]\} \\ \quad = [3,4].\end{array}$$

In this case, the unique fixed point [3,4] of g_{seq1} is reached after two iterations of g_{seq1}.

What happens when we apply this technique to an equation like

$$s\ =\ (cons\ \ (head\ \ s)\ \ (cons\ \ (1+\ (head\ \ s))\ \ [\,]))\,,$$

which has an infinite number of fixed points? The corresponding generating function is

$$g_{seq2} : Natseq \to Natseq\ =\ \lambda s\,.\ (cons\ \ (head\ \ s)\ \ (cons\ \ (1+\ (head\ \ s))\ \ [\,]))\,.$$

This function is continuous as long as $+$ returns $\bot_{Nat_\bot}$ when one of its arguments is $\bot_{Nat_\bot}$. The Least Fixed Point Theorem applies, and iterating g_{seq2} on $[\,]$ gives:

$$\begin{array}{l}(\mathbf{fix}_{Natseq}\ g_{seq2}) \\ \quad = \bigsqcup_{Natseq}\{(g^0_{seq2}\ [\,])\,,\ (g^1_{seq2}\ [\,])\,,\ (g^2_{seq2}\ [\,])\,,\ (g^3_{seq2}\ [\,])\,,\ \ldots\ \} \\ \quad = \bigsqcup_{Natseq}\{[\,],\ [\bot_{Nat_\bot}, \bot_{Nat_\bot}]\} \\ \quad = [\bot_{Nat_\bot}, \bot_{Nat_\bot}]\end{array}$$

After one iteration, the iterative fixed point technique finds the fixed point $[\bot_{Nat_\bot}, \bot_{Nat_\bot}]$, which is indeed less than all the other fixed points $[n, (n+1)]$. Intuitively, this result indicates that the solution is a sequence of two numbers, but that the value of those numbers cannot be determined without making an arbitrary decision. Note the crucial roles that the bottom elements $[\,]$ and $\bot_{Nat_\bot}$ play in this example. Each represents the value of a domain with the least information. Iterative application of the generating function may or may not refine these values by adding information.

A similar story holds for equations like

$$s\ =\ (cons\ \ (1+\ (head\ \ s))\ \ (cons\ \ (head\ \ s)\ \ [\,]))$$

that have no solutions in Nat^*. The reader can verify that this equation *does* have the unique solution $[\bot_{Nat_\bot}, \bot_{Nat_\bot}]$ in *Natseq* and that this solution can be found by an application of the Least Fixed Point Theorem.

As a final sequence example, we consider the equation $s = (cons\ \ 1\ \ s)$, whose associated generating function is

$$g_{seq3} : Natseq \to Natseq\ =\ \lambda s\,.\ (cons\ \ 1\ \ s)\,.$$

This function is continuous, and the Least Fixed Point Theorem can be invoked to find a solution to the original equation:

$$\begin{array}{l}(\mathbf{fix}_{Natseq}\ g_{seq3}) \\ \quad = \bigsqcup_{Natseq}\{(g^0_{seq3}\ [\,])\,,\ (g^1_{seq3}\ [\,])\,,\ (g^2_{seq3}\ [\,])\,,\ (g^3_{seq3}\ [\,])\,,\ \ldots\ \} \\ \quad = \bigsqcup_{Natseq}\{[\,],\ [1],\ [1,\ 1],\ [1,\ 1,\ 1],\ \ldots\ \} \\ \quad = [1,\ 1,\ 1,\ \ldots].\end{array}$$

In this case, the unique fixed point of g_{seq3} is an infinite sequence of 1s. This fixed point is not reached in a finite number of iterations, but is the limit of the sequence of approximations $\left(g^n_{seq3}\ [\,]\right)$. This example underscores why it is necessary to extend $Nat_\perp{}^*$ with $Nat_\perp{}^\infty$ to make *Natseq* a CPO. Without the infinite sequences in $Nat_\perp{}^\infty$, the iterative fixed point technique could not find a solution to some equations.

5.2.6.2 Function Examples

In the remainder of this book, we will typically apply the iterative fixed point technique to generating functions over function domains. Here we consider a few examples involving fixed points over the following domain of functions:

$$f \in Natfun = Nat \to Nat_\perp.$$

Since we assume that $\to$ designates *continuous* functions, *Natfun* is a domain of the continuous functions between *Nat* and $Nat_\perp$. *Natfun* is a CPO because the set of continuous functions between CPOs is itself a CPO under the usual ordering of functions. Furthermore, *Natfun* is pointed because $Nat_\perp$ is pointed. Recall that $Nat \to Nat_\perp$ is isomorphic to $Nat \rightharpoonup Nat$, so elements of *Natfun* can be represented by a function graph in which pairs whose target is $\perp_{Nat_\perp}$ are omitted.

Our first example is the definition of the doubling function studied earlier:

$$dbl = \lambda n\,.\,\mathbf{if}\ (n = 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (2 + (dbl\ (n-1)))\ \mathbf{fi}.$$

A solution to this definition is the fixed point of the generating function g_{dbl}:

$$\begin{array}{l}g_{dbl} : Natfun \to Natfun \\ = \lambda f.\ \lambda n.\ \mathbf{if}\,(n = 0)\ \mathbf{then}\ 0\ \mathbf{else}\,(2 + (f\ (n-1)))\ \mathbf{fi}.\end{array}$$

Natfun is a pointed CPO, and *Natfun*'s bottom element is the function whose graph is {}. In this CPO, $\bigsqcup$ on a chain of functions in $Nat \to Nat$ is equivalent to $\bigsqcup$ on a chain of graphs of functions in $Nat \rightharpoonup Nat$. It can be shown that g_{dbl}

is continuous, so the Least Fixed Point Theorem applies:

$$\begin{array}{l}(\mathbf{fix}_{Natfun}\ g_{dbl}) \\ \quad = \bigsqcup_{Natfun}\{(g^0_{dbl}\ \{\})\,,\ (g^1_{dbl}\ \{\})\,,\ (g^2_{dbl}\ \{\})\,,\ (g^3_{dbl}\ \{\})\,,\ \ldots\ \} \\ \quad = \bigsqcup_{Natfun}\{\{\},\ \{\langle 0,0\rangle\},\ \{\langle 0,0\rangle,\ \langle 1,2\rangle\},\ \{\langle 0,0\rangle,\ \langle 1,2\rangle,\ \langle 2,4\rangle\},\ \ldots\ \} \\ \quad = \{\langle n, 2n\rangle \mid n : Nat\}.\end{array}$$

Each $(g^n_{dbl}\ \{\})$ is a finite approximation of the doubling function that is only defined on the naturals $0 \le i \le n-1$. The least (and only) fixed point is the limit of these approximations: a doubling function defined on all naturals.

As an example of a function with an infinite number of fixed points, consider the following recursive definition of a function in *Natfun*:

$$even0 : Natfun\ =\ \lambda n\,.\ \mathbf{if}\ (n = 0)\ \mathbf{then}\ 0\ \mathbf{else}\ (even0\ \ (n\ mod\ 2))\ \mathbf{fi}.$$

Here $(a\ mod\ b)$ returns the remainder of dividing a by b. For each constant c in $Nat_\perp$, the function whose graph is

$$\bigcup_{n\,:\,Nat} \{\langle 2n, 0\rangle, \langle 2n+1, c\rangle\}$$

is a solution for *even0*. Each solution maps all even numbers to zero, but maps every odd number to the same constant c, where c is a parameter that distinguishes one solution from another. Each of these solutions is a fixed point of the generating function g_{even0}:

$$\begin{array}{l}g_{even0} : Natfun \to Natfun \\ = \lambda f\,.\ \ \lambda n\,.\ \ \mathbf{if}\,(n = 0)\ \ \mathbf{then}\ 0\ \ \mathbf{else}\,(f\ \ (n\ mod\ 2))\ \ \mathbf{fi}.\end{array}$$

It turns out that this function is continuous, so the Least Fixed Point Theorem gives:

$$\begin{array}{l}(\mathbf{fix}_{Natfun}\ g_{even0}) \\ \quad = \bigsqcup_{Natfun}\{(g^0_{even0}\ \{\})\,,\ (g^1_{even0}\ \{\})\,,\ (g^2_{even0}\ \{\})\,,\ (g^3_{even0}\ \{\})\,,\ \ldots\ \} \\ \quad = \bigsqcup_{Natseq}\{\{\},\ \{\langle 0,0\rangle\},\ \{\langle 0,0\rangle,\ \langle 2,0\rangle\},\ \{\langle 0,0\rangle,\ \langle 2,0\rangle,\ \langle 4,0\rangle\},\ \ldots\ \} \\ \quad = \{\langle 2n, 0\rangle \mid n : Nat\}.\end{array}$$

The least fixed point is a function that maps every even number to zero, but is undefined (i.e., yields $\perp_{Nat_\perp}$) on the odd numbers. Indeed, this is the least element of the class of fixed points described above; it uses the least arbitrary value for the constant c.

The solution for *even0* matches our intuitions about the operational behavior of programming language procedures for computing *even0*. For example, the definition for *even0* can be expressed in the SCHEME programming language via the following procedure:

```
(define (even0 n)
  (if (= n 0)
      0
      (even0 (mod n 2)))).
```

We expect this procedure to return zero in a finite number of steps for an even natural number, but to diverge for an odd natural number. The fact that the function *even0* maps odd numbers to $\perp_{Nat_\perp}$ can be interpreted as signifying that the procedure `even0` diverges on odd-numbered inputs.

▷ **Exercise 5.14** For each of the following equations:

- Characterize the set of all solutions to the equation in the specified solution domain;
- Use the iterative fixed point technique to determine the least solution to the equation.

Assume that $s : Natseq$, $p : \mathcal{P}(Nat)$, $f : Natfun$, and $h : Int \to Int_\perp$.

a. $s = (cons\ 2\ (cons\ (head\ (tail\ s))\ s))$

b. $s = (cons\ (1 + (head\ (tail\ s)))\ (cons\ 3\ s))$

c. $s = (cons\ 5\ (mapinc\ s))$, where *mapinc* is a function in $Natseq \to Natseq$ that maps every sequence $[n_1, n_2, n_3, \ldots]$ into the sequence $[(1 + n_1),\ (1 + n_2),\ (1 + n_3),\ \ldots]$

d. $p = \{1\} \cup \{x + 3 \mid x \in p\}$

e. $p = \{1\} \cup \{2x \mid x \in p\}$

f. $p = \{1\} \cup \{|2x - 4| \mid x \in p\}$

g. $f = \lambda n\,.\ (f\ n)$

h. $f = \lambda n\,.\ (f\ (1 + n))$

i. $f = \lambda n\,.\ (1 + (f\ n))$

j. $f = \lambda n\,.$ **if** $(n = 1)$
then 0
else if $(even?\ n)$
then $(1 + (f\ (n\,/\,2)))$
else $(f\ (n + 2))$
fi
fi

where *even?* is a predicate determining if a number is even.

k. $h = \lambda i\,.$ **if** $(i = 0)$ **then** 0 **else** $(h\ (i - 2))$ **fi** ◁

▷ **Exercise 5.15** Section 5.1.3 sketches an example involving the solution of an equation on line drawings involving the transformation T. Formalize this example by completing the following steps:

a. Represent line drawings as an appropriate pointed CPO *Lines*.

b. Express the transformation T as a continuous function g_T in $Lines \rightarrow Lines$.

c. Use the iterative fixed point technique to find the least fixed point of g_T. ◁

▷ **Exercise 5.16** A binary relation R on a set A is a subset of $A \times A$. The **reflexive transitive closure** of R is the smallest subset R' of $A \times A$ satisfying the following properties:

- If $a \in A$, then $\langle a, a \rangle \in R'$;
- If $\langle a, b \rangle$ is in R' and $\langle b, c \rangle$ is in R, then $\langle a, c \rangle$ is in R'.

a. Describe how the reflexive transitive closure of a binary relation can be expressed as an instance of the Least Fixed Point Theorem. What is the pointed CPO? What is the bottom element? What is the generating function?

b. Use the iterative fixed point technique to determine the reflexive transitive closure of the following relation on the set $\{\mathtt{a},\ \mathtt{b},\ \mathtt{c},\ \mathtt{d},\ \mathtt{e}\}$:

$$\{\langle \mathtt{a}, \mathtt{c} \rangle,\ \langle \mathtt{c}, \mathtt{e} \rangle,\ \langle \mathtt{d}, \mathtt{a} \rangle,\ \langle \mathtt{d}, \mathtt{b} \rangle,\ \langle \mathtt{e}, \mathtt{c} \rangle\}$$ ◁

▷ **Exercise 5.17** Show that each of the generating functions g_{seq1}, g_{seq2}, g_{seq3}, g_{dbl}, g_{even0} is continuous. ◁

5.2.7 Continuity and Strictness

We have seen how compound CPOs can be assembled out of component CPOs using the domain operators $_\perp$, $\times$, $+$, *, and $\rightarrow$. We have also seen how the pointedness of a compound CPO is in some cases dependent on the pointedness of its components.

But a pointed CPO D is not the only prerequisite of the Least Fixed Point Theorem. The other prerequisite is that the generating function $f : D \rightarrow D$ must be continuous. In the examples of the previous section, we waved our hands about the continuity of the generating functions, but did not actually prove continuity in any of the cases. The proofs are not difficult, but they are tedious. Below, we argue that all generating functions that can be expressed in the metalanguage summarized in Section A.4 are guaranteed to be continuous, so we generally do not need to worry about the continuity of generating functions.

We also introduce strictness, an important property for characterizing functions on pointed domains.

Recall that metalanguage expressions include:

- constants (both primitive values and primitive functions on such values);
- variables;
- assembly and disassembly operators for compound domains (e.g., $\langle \ldots \rangle$ and $Proj\,i$ notation for products, $Inj\,i$ and **matching** notation for sums, *cons*, *empty?*, *head*, and *tail* for sequences, λ abstraction and application for functions);
- syntactic sugar like **if** and the generalized pattern-matching version of **matching**.

It turns out that all of the assembly and disassembly operators for compound domains are continuous and that the composition of continuous functions is continuous (see [Sch86a] for the details). This implies that any function expressed as a composition of assembly and disassembly operators is continuous. As long as primitive functions are continuous and the **if** and **matching** notations preserve continuity, all functions expressible in this metalanguage subset must be continuous. Below, we refine our interpretation of primitive functions and the sugar notations so that continuity is guaranteed.

Assume for now that all primitive domains are flat CPOs. What does it mean for a function between primitive domains to be continuous? Since all chains on a flat domain D can contain at most two elements ($\perp_D$ and a non-bottom element d), the continuity of a function $f : D \to E$ between flat domains D and E is equivalent to the following monotonicity condition:

$$(f\ \perp_D) \sqsubseteq_E (f\ d)\,.$$

This condition is only satisfied in the following two cases:

- f maps $\perp_D$ to $\perp_E$, in which case d can map to any element of E;
- f maps *all* elements of D to the same non-bottom element of E.

In particular, f is *not* continuous if it maps $\perp_D$ and d to distinct non-bottom elements of E.

For example, a function *sqr* in $Nat_\perp \to Nat_\perp$ that maps $\perp_{Nat_\perp}$ to $\perp_{Nat_\perp}$ and every number to its square is continuous. So is the constant function *three*

that maps every element of $Nat_\perp$ (including $\perp_{Nat_\perp}$) to 3. But a function f that maps every non-bottom number n to its square and maps $\perp_{Nat_\perp}$ to 3 is *not* continuous, because $(f\ n)$ is not a refinement of the approximation $(f\ \perp_{Nat_\perp}) = 3$.

From a computational perspective, the continuity restriction makes sense because it only permits the modeling of computable functions. Uncomputable functions cannot be expressed without resorting to non-continuous functions. The celebrated halting function, which determines whether or not a program halts on a given input, is an example of an uncomputable function. Intuitively, the halting function requires a mechanism for detecting whether a computation is caught in an infinite loop; such a mechanism must map $\perp$ to one non-bottom element and other inputs to different non-bottom elements.

If D and E are pointed domains, a function $f : D \to E$ is **strict** if $(f\ \perp_D) = \perp_E$. Otherwise, f is **non-strict**. For example, the *sqr* function described above is strict, while the *three* function is non-strict. Although strictness and continuity are orthogonal properties in general, strictness does imply continuity for functions between flat domains (see Exercise 5.18).

Strictness is important because it captures the operational notion that a computation will diverge if it depends on an input that diverges. For example, strictness models the parameter-passing strategies of most modern languages, in which a procedure call will diverge if the evaluation of any of its arguments diverges. Non-strictness models the parameter-passing strategies of so-called lazy languages. See Chapters 7 and 11 for a discussion of these parameter-passing mechanisms.

When pointed CPOs are manipulated in our metalanguage, we shall assume the strictness of various operations:

- All the primitive functions on flat domains are strict. When such a function has multiple arguments, we will assume it is strict in each of its arguments. Thus, $+_{Nat_\perp}$ returns $\perp_{Nat_\perp}$ if either argument is $\perp_{Nat_\perp}$, and $=_{Nat_\perp}$ returns $\perp_{Bool_\perp}$ if either argument is $\perp_{Nat_\perp}$.
- An **if** expression is strict in its predicate whenever it is an element of $Bool_\perp$ rather than *Bool*. Thus the expression

  $$\textbf{if}\ x =_{Nat_\perp} y\ \textbf{then}\ 3\ \textbf{else}\ 3\ \textbf{fi}$$

  is guaranteed to return $\perp_{Nat_\perp}$ (*not* 3) if either x or y is $\perp_{Nat_\perp}$. Together with the strictness of $=_{Nat_\perp}$, the strictness of **if** predicates thwarts attempts to express non-computable functions. For example, the expression

  $$\textbf{if}\ x = \perp_{Nat_\perp}\ \textbf{then}\ \mathit{true}\ \textbf{else}\ \mathit{false}\ \textbf{fi}$$

will always return $\perp_{Bool_\perp}$.

- A **matching** expression is strict in its discriminant whenever it is an element of a pointed CPO. As with the strictness of **if** predicates, this restriction matches computational intuitions and prevents the expression of non-computable functions.

- If D is a pointed domain, we require the *head* operation on sequences to be strict on D^* under the prefix ordering. That is, (*head* [ ]) must equal $\perp_D$. If D is not pointed, or if D^* has the sum-of-products ordering, *head* is undefined for [ ]; i.e., it is only a partial function.

With the above provisions for strictness, it turns out that all functions expressible in the metalanguage are continuous.

Since we often want to specify new strict functions, it is helpful to have a convenient notation for expressing strictness. If f is any function between pointed domains D and E, then ($\mathbf{strict}_{D,E}\ f$) is a strict version of f. That is, ($\mathbf{strict}_{D,E}\ f$) maps $\perp_D$ to $\perp_E$ and maps every non-bottom element d of D to ($f\ \ d$). As usual, we will omit the subscripts on **strict** when they are clear from context. For example, a strict function in $Nat_\perp \to Nat_\perp$ that returns 3 for all non-bottom inputs can be defined as:

$$\mathit{strict\text{-}three} = (\mathbf{strict}\ \ (\lambda n\,.\,3))\,.$$

We adopt the abbreviation $\underline{\lambda}\,.\ \ldots$ for ($\mathbf{strict}\ \ (\lambda\,.\ \ \ldots)$), so $\underline{\lambda} n\,.\ 3$ is another way to write the above function.

▷ **Exercise 5.18**

a. Show that strictness and continuity are orthogonal by exhibiting functions in $D \to D$ that have the properties listed below. You may choose different Ds for different parts.

   i. Strict and continuous;

   ii. Non-strict and continuous;

   iii. Strict and non-continuous;

   iv. Non-strict and non-continuous.

b. Which combinations of properties from the previous part cannot be achieved if D is required to be a flat domain? Justify your answer. ◁

5.3 Reflexive Domains

Reflexive domains are domains that are defined by recursive domain equations. We have already seen reflexive domains in the context of PostFix:

$$StackTransform = Stack \rightarrow Stack$$
$$Stack = Value^* + Error$$
$$Value = Int + StackTransform.$$

These equations imply that a stack may contain as one of its values a function that maps stacks to stacks. A simpler example of reflexive domains is provided by the lambda calculus (see Chapter 6), which is based upon a single domain Fcn defined as follows:

$$Fcn = Fcn \rightarrow Fcn.$$

We know from set theory that descriptions of sets that contain themselves (even indirectly) as members are not necessarily well-defined. In fact, a simple counting argument shows that equations like the above are nonsensical if interpreted in the normal set-theoretic way. For example, if we (improperly) view $\rightarrow$ as the domain constructor for set theoretic functions from Fcn to Fcn, by counting the size of each set we find:

$$|Fcn| = |Fcn|^{|Fcn|}.$$

For any set Fcn with more than one element, $|Fcn|^{|Fcn|}$ is bigger than $|Fcn|$. Even if $|Fcn|$ is infinite, $|Fcn|^{|Fcn|}$ is a "bigger" infinity! In the usual theory of sets, the only solution to this equation is a trivial domain Fcn with one element. A computational world with a single value is certainly not a very interesting, and is a far cry from computationally complete world of the lambda calculus!

Dana Scott had the insight that the functions that can be implemented on a computer are limited to continuous functions. There are fewer continuous functions than set theoretic functions on a given CPO, since the set theoretic functions do not have to be monotonic (you can get more information out of them than you put in!). If we treat $\rightarrow$ as a constructor that describes computable (continuous) functions and we interpret "equality" in domain equations as isomorphisms, then we have a much more interesting world. In this world, we can show an isomorphism between Fcn and $Fcn \rightarrow Fcn$:

$$Fcn \approx Fcn \rightarrow Fcn.$$

The breakthrough came when Scott [Sco77] provided a constructive technique (the so-called **inverse limit construction**) that showed how to build such a domain and prove the isomorphism. Models exist as well for all of the

other domain constructors we have introduced (lifting, products, sums, sum-of-products, prefix ordering of sequences) and as long as we stick to well defined domain constructors, we can be assured that there is a non-trivial solution to our reflexive domain equations.

The beauty of this mathematical approach is that there is a formal way of giving meaning to programming language constructs without any use of computation. We shall not describe the details of the inverse limit construction here. For these, see Scott's 1976 Turing Award Lecture [Sco77], Chapter 11 of Schmidt [Sch86a], and Chapter 7 of Stoy [Sto85].

It is important to note that this construction requires that certain domains have bottom elements. For example, in order to solve the PostFix domain equations, we need to lift the *Stack* and *Answer* domains:

$$\begin{aligned} \mathit{StackTransform} &= \mathit{Stack} \to \mathit{Stack} \\ \mathit{Stack} &= (\mathit{Value}^* + \mathit{Error})_\perp \\ \mathit{Value} &= \mathit{Int} + \mathit{StackTransform} \\ \mathit{Answer} &= (\mathit{Int} + \mathit{Error})_\perp \end{aligned}$$

This lifting explains how non-termination can "creep in" when PostFix is extended with `dup`.

The inverse limit construction is only one way to understand reflexive domain equations. Many approaches to interpreting such equations have been proposed over the years. One popular modern approach is based on the notion of **information systems**. You can find out more about this approach in [GS90, Gun92, Win93].

5.4 Summary

Here are the "big ideas" of this chapter:

- The meaning of a recursive definition over a domain D can be understood as the fixed point of a function $D \to D$.
- Complete partial orders (CPOs) model domain elements as approximations that are ordered by information. In a CPO, every sequence of information-consistent approximations has a well-defined limit.
- A CPO D is pointed if it has a least element (bottom, written $\perp_D$). The bottom element, which stands for "no information," is used as a starting point for the fixed point process. Bottom can be used to represent a partial function as a total function. It is often used to model computations that diverge (go into an infinite loop). A function between CPOs is strict if it preserves bottom.

- Functions between CPOs are monotonic if they preserve the information ordering and continuous if they preserve the limits. Continuity implies monotonicity, but not vice versa.
- If D is a pointed CPO, every continuous function $f : D \to D$ has a least fixed point ($\mathbf{fix}_d\ f$) that is defined as the limit of iterating f starting at $\bot_D$.
- The domain constructors $_\bot$, $\times$, $+$, $\to$, and $\overline{*}$ can be viewed as operators on CPOs. In particular, $D_1 \to D_2$ is interpreted as the CPO of *continuous* functions from D_1 to D_2. Only some of these constructors preserve pointedness. The new domain constructor $_\bot$ extends a domain with a new bottom element, guaranteeing that it is pointed.
- Functions that can be expressed in the metalanguage of Section A.4 are guaranteed to be continuous. Intuitively, such functions correspond to the computable functions.
- Recursive domain equations that are not solvable when domains are viewed as sets can become solvable when domains are viewed as CPOs. The key ideas (due to Scott) are to interpret equality as isomorphism and to focus only on continuous functions rather than all set-theoretic functions. There are restricted kinds of CPOs for which any domain equations over a rich set of operators are guaranteed to have a solution.

Reading

This chapter is based largely on Schmidt's presentation in Chapter 6 of [Sch86a]. The excellent overview article by Gunter and Scott [GS90] presents alternative approaches involving more restricted domains and touches upon many technical details omitted above. See Mosses's article on denotational semantics [Mos90] to see how these more restricted domains are used in practice. Gunter's book [Gun92] discusses many domain issues in detail.

For an introduction to the techniques of solving recursive domain equations, see [Sto85, Sch86a, GS90, Gun92, Win93].

Chapter 6

FL: A Functional Language

Things used as language are inexhaustible attractive.

— *The Philosopher, Ralph Waldo Emerson*

6.1 Decomposing Language Descriptions

The study of a programming language can often be simplified if it is decomposed into three parts:

1. A **kernel** language that forms the essential core of the language.
2. **Syntactic sugar** that extends the kernel with convenient constructs. Such constructs can be automatically translated into kernel constructs via a process known as **desugaring**.
3. A **standard library** of primitive constants and operators supplied with the language.

We shall refer to the combination of a kernel, syntactic sugar, and a standard library as a **full language** to distinguish it from its components.

Decomposing a programming language definition into parts relieves a common tension in the design and analysis of programming languages. From the standpoint of reasoning about a language, it is desirable for a language to have only a few simple parts. However, from the perspective of programming in a language, it is desirable to concisely and conveniently express common programming idioms. A language that is too pared down may be easy to reason about but horrendous to program in — try writing factorial in PostFix+{`dup`}. On

the other hand, a language with many features may be convenient to program in but difficult to reason about — try proving some non-trivial properties about your next JAVA, C, ADA, or COMMON LISP program.

The technique of viewing a full language as mostly sugar coating around a kernel lets us have our cake and eat it too. When we want to reason about the language, we consider only the small kernel upon which everything else is built. But when we want to program in the language, we make heavy use of the syntactic sugar and standard library to express what we want in a readable fashion. Indeed, we can even add new syntactic sugar and new primitives without changing the properties of the kernel.

There are limitations to this approach. We'd like the kernel and full language to be close enough so that the desugaring is easy to understand. Otherwise we might have the situation where the kernel is a machine instruction set and the desugaring is a full-fledged compilation from high-level programs into object code. For this reason, we require that syntactic sugar be expressed via simple local transformations; no global program analysis is allowed.

We shall study this language decomposition technique in the context of a mini-language we call FL (for *F*unctional *L*anguage).[1] FL provides us with the opportunity to use the semantic tools developed in the previous chapters to analyze a programming language that is much closer to a "real" programming language than POSTFIX or EL. Along the way, we will introduce two approaches for modeling names in a programming language: substitution and environments.

FL is a language that examplifies what is traditionally known as the **functional programming paradigm**. As we shall see, functional programming languages are characterized by an emphasis on the manipulation of values that model mathematical functions. The name "functional language" is a little bit odd, since it suggest that languages not in this paradigm are somehow *dys*functional — a perception that many functional language aficionados actively promote! Perhaps **function-oriented languages** would be a more accurate term for this class of languages.

6.2 The Structure of FL

FL is a typical functional programming language for computing with numeric, boolean, symbolic, procedural, and compound data values. The computational model of FL is based on the functional programming paradigm exemplified by

[1] Our FL language is not to be confused with any other similarly-named language. In particular, our FL is *not* related to the FL functional programming language [BWW90, BWW+89] based on Backus's FP [Bac78].

such languages as ERLANG, FX, HASKELL, ML, and SCHEME. FL programs are free of side effects and make heavy use of first-class functional values (here called **procedures**[2]. Syntactically, FL bears a strong resemblance to SCHEME, but semantically we shall see that it is closer to so-called **purely functional lazy languages** like HASKELL and MIRANDA.

6.2.1 FLK: The Kernel of the FL Language

We begin by presenting the syntax and informal semantics of FLK, the FL kernel.

6.2.1.1 The Syntax of FLK

A well-formed FLK program is a member of the syntactic domain Program defined by the s-expression grammar in Figure 6.1. FLK programs have the form `(flk (`I_{formal}`*)` E_{body}`)`, where I_{formal}* are the **formal parameters** of the program and E_{body} is the **body expression** of the program. Intuitively, the formal parameters name program inputs and the body expression specifies the result value computed by the program for its inputs.

FLK expressions are s-expression syntax trees whose leaves are either literals or variable references. FLK literals include the unit literal, booleans, integers, and symbols. We adopt the SCHEME convention of writing the boolean literals as `#t` (true) and `#f` (false). The unit literal (`#u`) is used in situations where the value of an expression is irrelevant, such as contexts in C and JAVA modeled by the `void` type. For symbolic (i.e., non-numeric) processing, FLK supports the LISP-like notion of a **symbol**. Symbols are similar to strings in traditional languages, except that they a written with a different syntax (using the keyword `symbol` rather than double quotes) and they are atomic entities that cannot be decomposed into their component characters. For simplicity, FLK assumes a LISP-like convention in which symbols are sequences of characters (1) that do not include whitespace, bracket characters ({, }, `(`, `)`, `[`, `]`), or quote characters (`"`, `‘`, `’`); (2) that do not begin with `#`; and (3) in which case is ignored. So the symbols `xcoord`, `xCoord`, and `XCOORD` are considered equivalent.

[2]We shall consistently use the term **procedure** to refer to entities in programming languages that denote mathematical functions, and **function** to refer to the mathematical notion of function. In some languages, these two terms are used to distinguish different kinds of programming language entities. For example, in PASCAL, "function" refers to a subroutine that returns a result whereas "procedure" refers to a subroutine performs its work via side effect and returns no result. Much of the functional programming literature uses the term "function" to refer both to the programming language entity and the mathematical entity it denotes.

$P \in$ Program
$E \in$ Exp
$L \in$ Lit
K $\in$ Keyword $= \{$`call`, `if`, `pair`, `primop`, `proc`, `rec`, `symbol`, `error`$\}$
$Y \in$ Symlit $= \{$`x`, `lst`, `make-point`, `map_tree`, `4/3*pi*r^21`, $\ldots\}$
$I \in$ Identifier $=$ Symlit $-$ Keyword
$B \in$ Boollit $= \{$`#t`, `#f`$\}$
$N \in$ Intlit $= \{\ldots,$ `-2`, `-1`, `0`, `1`, `2`, $\ldots\}$
$O \in$ Primop $=$ *Defined by standard library (Section 6.2.3).*

P ::= `(flk (`I_{formal}`*)` E_{body}`)`	[Program]
E ::= L	[Literal]
| I	[Variable Reference]
| `(primop` O_{name} E_{arg}`*)`	[Primitive Application]
| `(proc` I_{formal} E_{body}`)`	[Abstraction]
| `(call` E_{rator} E_{rand}`)`	[Application]
| `(if` E_{test} E_{then} E_{else}`)`	[Branch]
| `(pair` E_{fst} E_{snd}`)`	[Pairing]
| `(rec` I_{var} E_{body}`)`	[Recursion]
| `(error` I_{msg}`)`	[Errors]
L ::= `#u`	[Unit Literal]
| B	[Boolean Literal]
| N	[Integer Literal]
| `(symbol` I`)`	[Symbolic Literal]

Figure 6.1: An s-expression grammar for FLK

A key difference between FLK and PostFix/EL is that that FLK provides constructs (`flk`, `proc`, and `rec`) that introduce names for values. Syntactically, names are expressed via **identifiers**. The rules for what constitutes a well-formed identifier differs from language to language. In FLK we shall assume that any symbol that is not one of the **reserved keywords** of the language (`call`, `if`, `pair`, `primop`, `proc`, `rec`, `symbol`, and `error`) can be used as an identifier. This means that expressions like `x-y` and `4/3*pi*r^2` are treated as atomic identifiers in FLK. In many other languages, these would be infix specifications of trees of binary operator applications.

For compound expressions, FLK supports procedural abstractions (`proc`) and applications (`call`), primitive applications (`primop`), conditional branches (`if`), pair creation (`pair`), simple recursion (`rec`), and error signaling (`error`).

Although many of the syntactic conventions of FLK are borrowed from Lisp-like languages, especially Scheme, it's worth emphasizing that FLK differs from these languages in some fundamental ways. For example, in Scheme, abstractions may take any number of formal parameters, are introduced via the keyword `lambda`, and are invoked via an application syntax with no keyword. In contrast, FLK abstractions have exactly one formal parameter, are introduced via the keyword `proc`, and are applied via the keyword `call`.

6.2.1.2 An Informal Semantics for FLK

Intuitively, every FLK expression denotes a value that is tagged with its type in addition to whatever information distinguishes it from other values of the same type. The primitive values supported by FLK include the unit value, boolean truth values, integers, and textual symbols. The unit value is the unique value of a distinguished type that has a single element. In addition, FLK supports pairs and procedures. A pair is a compound value that allows any two values (which may themselves be pairs) to be glued together to form a single value. A procedure is a value that represents a mathematical function by specifying how to map an input value to an output value.

We will informally describe the semantics of FLK by considering some sample evaluations of FLK expressions. We use the notation $E \xrightarrow[FLK]{} V$ to indicate that the expression E evaluates to the value V. Here are some example values that indicate our conventions for writing FLK values:

unit	The unit value
false, *true*	The boolean values
17, -3	Integer values
$'$*abstraction*, $'$*captain*	Symbolic values
procedure	Procedural values
$\langle 17, true\rangle$, $\langle procedure, \langle 'abstraction, unit\rangle\rangle$	Pair values
error:not-an-integer	Errors
$\infty-loop$	Non-termination (represents an infinite loop)

Additionally, the following abbreviation will be handy for representing lists of values that are encoded as a unit-terminated chain of pairs:

$$[V_1, V_2, \ldots, V_n] = \langle V_1, \langle V_2, \ldots \langle V_n, unit\rangle \ \ldots\rangle\rangle$$

For example, the notation $[17, true, \langle 'foo, procedure\rangle]$ is an abbreviation for a three-element list values $\langle 17, \langle true, \langle\langle 'foo, procedure\rangle, unit\rangle\rangle\rangle$.

Our value notation does not distinguish procedural values that denote different mathematical functions. For instance, a squaring procedure and a doubling procedure are both written *procedure*. This is because our operational semantics will not allow us to directly observe the function designated by a procedural value that is the outcome of a program. As explained in Section 3.4.4, intentionally blurring distinctions between certain values is sometimes necessary to enable program transformations. However, our notation for errors *does* distinguish errors with different messages.

The literal expressions designate constants in the language:

`#u` $\xrightarrow[FLK]{}$ *unit*
`#t` $\xrightarrow[FLK]{}$ *true*
`23` $\xrightarrow[FLK]{}$ *23*
`(symbol captain)` $\xrightarrow[FLK]{}$ $'$*captain*

The primitive application `(primop` O E_1 $\ldots$ E_n`)` denotes the result of applying the primitive operator named by O to the n values of the argument expressions E_i. The behavior of most of the primitive operators should be apparent from their names. E.g.,

`(primop not? #t)` $\xrightarrow[FLK]{}$ *false*
`(primop integer? 1)` $\xrightarrow[FLK]{}$ *true*
`(primop integer? #t)` $\xrightarrow[FLK]{}$ *false*
`(primop + 1 2)` $\xrightarrow[FLK]{}$ *3*
`(primop / 17 5)` $\xrightarrow[FLK]{}$ *3* {*integer division*}
`(primop rem 17 5)` $\xrightarrow[FLK]{}$ *2*
`(primop sym=? (symbol captain) (symbol abstraction))` $\xrightarrow[FLK]{}$ *false*
`(primop sym=? (symbol captain) (symbol Captain))` $\xrightarrow[FLK]{}$ *true*

The last example illustrates that FLK symbols are case-insensitive. The full list of primitive operations is specified by the FL standard library in Section 6.2.3.

The value of a primitive application is not defined when primitive functions are given the wrong number of arguments, when an argument has an unexpected type, or when integer division or remainder by 0 is performed. These situations are considered program errors:

```
(primop + 1) ⟶FLK error:too-few-args
(primop not? 1) ⟶FLK error:not-a-bool
(primop + #t 1) ⟶FLK error:not-an-integer
(primop / 1 0) ⟶FLK error:divide-by-zero
```

The abstraction `(proc` *I E*`)` specifies a procedural value that represents a mathematical function. The application `(call` E_1 E_2`)` stands for the result of applying the procedure denoted by E_1 to the operand value denoted by E_2. It is an error to use any value other than a procedure as an operator. Multiple-argument procedures can be simulated by currying (see Section A.2.5.1).

```
(proc x (primop * x x)) ⟶FLK procedure
(call (proc x (primop * x x)) 5) ⟶FLK 25
(call (call (proc a (proc b (primop - b a))) 2) 3) ⟶FLK 1
(call 3 5) ⟶FLK error:non-procedural-rator
(call not? #t) ⟶FLK error:unbound-variable
  {not? is a primop, not a variable name}
(call (proc x (call x x)) (proc x (call x x))) ⟶FLK ∞-loop
```

As in HASKELL, FLK's procedures are **non-strict**. This means that a call to a procedure may return a value even if one of its arguments denotes an error or a non-terminating computation. Intuitively, non-strictness indicates that an expression will never be evaluated if the rest of the computation does not require its value. For example:

```
(call (proc x 3) (primop / 1 0)) ⟶FLK 3
(call (proc x (primop + x 3))
      (primop / 1 0)) ⟶FLK error:divide-by-zero
(call (proc x 3)
      (call (proc x (call x x))
            (proc x (call x x)))) ⟶FLK 3
(call (proc x (primop + x 3) )
      (call (proc x (call x x))
            (proc x (call x x)))) ⟶FLK ∞-loop
```

Unlike FLK, most real-world languages (including C, JAVA, PASCAL, SCHEME, and ML) have **strict** procedures. In these langauges, operands of procedure applications are always evaluated, even if they are never referenced by the pro-

cedure body.

The branch expression (`if` E_{test} E_{then} E_{else}) requires the value of E_{test} to be a boolean, and evaluates one of E_{then} or E_{else} depending on whether the test is true or false:

```
(if (primop > 8 7) (primop + 2 3) (primop * 2 3)) ⟶FLK 5
(if (primop < 8 7) (primop + 2 3) (primop * 2 3)) ⟶FLK 6
(if (primop - 8 7) (primop + 2 3) (primop * 2 3))
  ⟶FLK error:non-bool-in-if-test
```

The pairing expression (`pair` E_{fst} E_{snd}) is the means of gluing two values together into a single value of the pair type. The values of the two components can be extracted via the primitive operators `fst` and `snd`. A chain pairs linked by their second components and terminated by the unit value is a standard way of encoding a list:

```
(pair 1 (pair 2 (pair 3 #u))) ⟶FLK [1, 2, 3]
```

Like procedure calls, pairing in FLK is non-strict. The result of `pair` is always a well-defined pair even if one (or both) of its argument expressions is not an FLK value. The unspecified nature of a contained value can only be detected when it is extracted from the pair.

```
(pair (primop not? #f) (primop / 1 0))
  ⟶FLK ⟨true, error:divide-by-zero⟩
(primop fst (pair (primop not? #f) (primop / 1 0))) ⟶FLK true
(primop snd (pair (primop not? #f) (primop / 1 0)))
  ⟶FLK error:divide-by-zero
```

As we shall see in Section 10.1.3, non-strict data structures are an important mechanism for supporting modularity in programs.

We choose to make `pair` a special form rather than a primitive like `not?` or `+` to emphasize the fact that pairing is non-strict. If we made `pair` a primitive operator, we would still have to treat it specially when we describe the semantics of the `primop` form because all the other primitives are strict. Treating `pair` as a special form provides a cleaner description of the semantics. This is a purely stylistic decision; it is also possible to treat `pair` as a binary primitive operator (see Exercise 6.20).

The recursion expression (`rec` I E) allows the expression of recursion equations over one variable. The value of the `rec` expression is the value of its body, where the value of I within E is the value of the entire `rec` expression. That is, the value returned by a recursion is the solution to the equation $I = E$. `rec` is used to specify recursive procedures and data structures. For example:

```
(rec fact (proc n
            (if (primop = n 0)
                1
                (primop * n (call fact (primop - n 1))))))
```
$\xrightarrow[FLK]{}$ *procedure* {*A factorial procedure.*}

```
(rec ones (pair 1 ones))
```
$\xrightarrow[FLK]{}$ [*1*, *1*, *1*, ...] {*An infinite sequence of 1s.*}

FLK programs are parameterized expressions. We use the notation $P \xrightarrow[FLK]{[V_1,\ldots,V_n]} V_{result}$ to indicate that running the FLK program P on argument values V_1, ..., V_n yields the result value V_{result}. For example:

`(flk (x) (* x x))` $\xrightarrow[FLK]{[5]}$ *25*

`(flk (a b) (/ (+ a b) 2))` $\xrightarrow[FLK]{[2,8]}$ *5*

`(flk (a b) (/ (+ a b) 2))` $\xrightarrow[FLK]{[2,8,11]}$ *error:wrong-number-of-args*

```
(flk (x nums)
  (call (rec scale
          (proc ys
            (if (primop unit? ys)                  {Is ys the empty list?}
                ys                                 {If so, return it;}
                (pair                              {otherwise, prepend the}
                  (primop * x (primop fst ys))     {scaled first number}
                  (call scale                      {to the result of scaling}
                        (primop snd xs))))))       {the rest of the numbers.}
        nums))
```
$\xrightarrow[FLK]{[4,[1,2,3]]}$ [*4*, *8*, *12*]

The penultimate example illustrates that it is an error if the number of arguments supplied to the program differs from the number of formal parameters declared. The final example illustrates that FLK program arguments may include values other than integers, such as lists of integers in this case.

In general, the values considered to be valid program arguments will be a proper subset of the values manipulated by a language. In languages such as C and JAVA, program arguments must be strings, and these can be parsed into other kinds of values (such as integers, floating-point numbers, arrays of numbers, etc.) where necessary. Program arguments are typically limited to literal data with simple textual representations, which excludes procedural values as program arguments. In the case of FLK, we shall assume that program arguments may be any of the literal values (unit, booleans, integers, symbols) and binary trees (i.e., trees with `pair` nodes) whose leaves are such literals. Since s-expressions can be represented as such trees, this will allow us to write FLK programs that manipulate representations of programming language ASTs.

This concludes our informal description of the semantics of FLK. While FLK has considerably more expressive punch than PostFix or FL, expressing even simple programs within FLK is rather cumbersome. In the next section, we will see how to extend FLK to another language, FL, that maintains simplicity in the semantics but yields a language in which it is practical to write (and read!) non-trivial functional programs.

6.2.2 FL Syntactic Sugar

6.2.2.1 Syntactic Sugar Forms

P	$\in$	Program		
D	$\in$	Def		
SX	$\in$	SExp		
E	$\in$	Exp		
I	$\in$	Identifier		
B	$\in$	Boollit	=	{`#t`, `#f`}
N	$\in$	Intlit	=	{..., `-2`, `-1`, `0`, `1`, `2`, ...}

P ::= `(fl (`I_{formal}`*)` E_{body} $D_{definitions}$`*)`	[Program]
D ::= `(define` I_{name} E_{value}`)`	[Definition]
E ::= ...	[FLK constructs]
| `(lambda (`I_{formal}`*)` E_{body}`)`	[Multi-Abstraction]
| `(`E_{rator} E_{rand}`*)`	[Multi-Application]
| `(list` $E_{element}$`*)`	[List]
| `(quote` SX`)`	[S-Expression]
| `'`SX	[S-Expression Shorthand]
| `(cond (`E_{test} E_{action}`)* (else` $E_{default}$`))`	[N-Way Branch]
| `(scand` $E_{conjunct}$`*)`	[Short-Circuit And]
| `(scor` $E_{disjunct}$`*)`	[Short-Circuit Or]
| `(let ((`I_{var} E_{defn}`)*)` E_{body}`)`	[Local Binding]
| `(letrec ((`I_{var} E_{defn}`)*)` E_{body}`)`	[Recursive Binding]
SX ::= I	[Symbol]
| `#u`	[Unit]
| B	[Boolean]
| N	[Integer]
| `(`SX_{elt}`*)`	[List]

Figure 6.2: Grammar for FL syntactic sugar.

The syntax of FL's syntactic abbreviations are specified in the grammar presented in Figure 6.2. In the definition of E, the ellipses ... stand for all the expression productions in the FLK grammar. The new expression forms in Figure 6.2 can be used anywhere the nonterminal E appears in the kernel FLK constructs as well as in the new syntactic abstractions. We first explain informally the meaning of each abbreviation before showing how to desugar them into FLK. Many of these syntactic abbreviations are inspired by constructs in LISP dialects, but we shall see that some of them have somewhat different meanings in FL than in LISP.

FL's `lambda` construct can bind any number (possibly zero) of identifiers within a procedure body. In the tagless multi-application form, a procedure can be applied to any number (possibly zero) of arguments. Because multi-applications are the only tagless form, the lack of an explicit tag is not ambiguous. Because applications tend to be the most common kind of compound expression, eliminating the explicit tag for this case makes expressions more concise. The multi-abstraction and multi-application forms are inspired by SCHEME syntax. Unlike SCHEME, FL supports implicit currying with these constructs. For example, suppose that E_{abs3} is the three-parameter multi-abstraction

```
(lambda (a b c) (primop * a (primop + b c))).
```

Then (`$E_{abs3}$ 2 3 4`) denotes *14*, (`$E_{abs3}$ 2 3`) denotes the same procedure as `(lambda (c) (primop * 2 (primop + 3 c)))`, and (`$E_{abs3}$ 2`) denotes the same procedure as `(lambda (b c) (primop * 2 (primop + b c)))`.

The `list` construct is a shorthand for creating lists by a sequence of nested pairings. (`list` E_1 ... E_n) constructs a unit-terminated, chain of n pairs linked by their second components where the value of E_i is the value of the first element of the ith pair in the chain. For example,

```
(list (primop + 1 2) (primop = 3 4) (pair 4 5))
```

is equivalent to

```
(pair (primop + 1 2)
      (pair (primop = 3 4)
            (pair (pair 4 5)
                  #u))).
```

The `quote` construct facilitates the construction of **s-expressions** , which are recursively defined to be literals (unit, numeric, boolean, and symbolic) and lists of s-expressions. Quoted s-expressions are a very concise way to specify tree-structured data. The `quote` form can be viewed as a means of a constructing a tree from a printed representation of the tree. For example, the s-expression

```
(quote (1 (#t three) (four 5 six)))
```

is a shorthand for

```
(list 1
      (list #t (symbol three))
      (list (symbol four) 5 (symbol six))).
```

To make the abbreviation even more concise, we adopt the LISP convention that `'`*SX* is a shorthand for `(quote` *SX*`)`, so the above example can also be written `'(1 (#t three) (four 5 six))`. The ability to express s-expressions so concisely with the quotation forms makes them very handy for specifying programs that manipulate program phrases from languages with s-expression syntax. For example, the POSTFIX program `(postfix 1 (2 mul) exec)` can be represented as the FL s-expression form `'(postfix 1 (2 mul) exec)`.

The `cond` construct is an n-way conditional branch that stands for a nested sequence of `if` expressions. For example,

```
(cond ((primop > temp 80) (symbol hot))
      ((primop < temp 50) (symbol cold))
      (else (symbol mild)))
```

is equivalent to

```
(if (primop > temp 80)
    (symbol hot)
    (if (primop < temp 50)
        (symbol cold)
        (symbol mild))).
```

The `scand` and `scor` expressions provide for so-called **short-circuit** evaluation of boolean conjunctions and disjunctions, respectively. If a false value is encountered in the left-to-right evaluation of the conjuncts of a `scand` form, then the result of the form is the false value, regardless of whether subsequent conjuncts contain errors or infinite loops. So

```
(scand (primop = 1 2) (primop / 3 0))
```

evaluates to *false* but

```
(scand (primop / 3 0) (primop = 1 2))
```

signals a divide-by-zero error. Similarly, if a true value is encountered in the left-to-right evaluation of the disjuncts of a `scor` form, then the result of the form is the true value, regardless of whether the subsequent disjuncts contain errors or infinite loops.

The `(let ((`I_1 E_1`) ... (`I_n E_n`))` E_0`)` expression evaluates E_0 in a con-

text where the names $I_1 \ldots I_n$ are bound to the values of the expressions $E_1 \ldots E_n$. For example,

```
(let ((a (primop * 4 5))
      (b (primop + 3 4)))
  (/ (primop + a b) (primop - a b)))
```

evaluates to *2*.

The `(letrec ((`I_1 E_1`) ... (`I_n E_n`))` E_{body}`)` expression is similar to the `let` expression except that the names $I_1 \ldots I_n$ are visible inside of the expressions $E_1 \ldots E_n$. The `letrec` expression is similar to the `rec` expression, except that it can be thought of as solving a group of mutually recursive equations. For example,

```
(letrec ((even? (lambda (x)
                  (if (primop = x 0)
                      #t
                      (odd? (primop - x 1)))))
         (odd? (lambda (y)
                 (if (primop = y 0)
                     #f
                     (even? (primop - y 1))))))
  (list (even? 0) (odd? 1) (odd? 2) (even? 3)))
```

evaluates to $[true, true, false, false]$.

The top-level program construct `(program (`$I_{formals}$`*)` E_{body} $D_{definitions}$`*)` evaluates the body expression E_{body} in a context where

- the formal program parameters $I_{formals}$* are bound to the program arguments;
- the definition names $D_{definitions}$* in are bound to the values of the corresponding definition expressions;
- and each member of a set of **standard identifiers** (names in the standard library) is bound to the value specified by the library.

The advantage of a standard library is that many primitive constants and procedures can be factored out of the syntax of the language. Of course, it is still necessary to specify the components of the library somewhere in a language description. Typically the library is specified by listing all elements in the library along with a description of the semantics of each one. We will do this for the FL library in Section 6.2.3.

Definitions make it convenient to name top-level values (typically procedures) that are used within E_{body}. The value expressions of the definitions are evaluated

in a mutually recursive context: the expression in a definition may refer to any name introduced by the definitions.

Consider the following sample FL program:

```
(fl (ns) (pair (map even? ns) (map odd? ns))
  (define even? (lambda (x)
                  (if (= x 0)
                      #t
                      (odd? (- x 1)))))
  (define odd? (lambda (y)
                 (if (= y 0)
                     #f
                     (even? (- y 1)))))
  (define map (lambda (f xs)
                (if (unit? xs)
                    xs
                    (pair (f (fst xs))
                          (map f (snd xs)))))))
```

The body expression `(pair (map even? ns) (map odd? ns))` refers to the procedures `even?`, `odd?`, and `map` defined via definitions within P. As above, `even?` and `odd?` are mutually recursive. The fact that standard identifiers are bound to appropriate procedures in the program body and definitions means that `=`, `-`, `unit?`, `fst`, and `snd` can all be used without the `primop` tag.

6.2.2.2 Desugaring

The transformation that desugars FL into FLK is presented in Figures 6.3 and 6.4. The transformation is specified by two desugaring functions:

1. $\mathcal{D}_{\text{exp}}$ maps an FL expression to a FLK expression.

2. $\mathcal{D}_{\text{prog}}$ maps FL programs to FLK programs.

As these desugaring functions walk down FL program and expression ASTs, they perform local transformations that replace the syntactic sugar constructs of FL by FLK constructs. Some clauses of the functions require the introduction of an identifier. In these cases, we want to ensure that the name does not conflict with any identifiers used by the programmer (or other identifiers introduced by the rules themselves). An implementation of the desugaring rules will include a way to generate such new names. We refer to these variables as **fresh.** (See page 237 for further discussion of fresh variables.)

In Figure 6.3, the top clauses descend the syntactic constructs of FL that are inherited from FLK, recursively applying $\mathcal{D}_{\text{exp}}$ to all subexpressions. This will

$\mathcal{D}_{\text{exp}} : \text{Exp}_{FL} \rightarrow \text{Exp}_{FLK}$

$\mathcal{D}_{\text{exp}}[\![L]\!] = L$
$\mathcal{D}_{\text{exp}}[\![I]\!] = I$
$\mathcal{D}_{\text{exp}}[\![\texttt{(primop } O\ E_1\ \ldots\ E_n\texttt{)}]\!] = \texttt{(primop } O\ \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \ldots\ \mathcal{D}_{\text{exp}}[\![E_n]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \texttt{(call } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(if } E_{test}\ E_{then}\ E_{else}\texttt{)}]\!] = \texttt{(if } \mathcal{D}_{\text{exp}}[\![E_{test}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{then}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{else}]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(pair } E_{fst}\ E_{snd}\texttt{)}]\!] = \texttt{(pair } \mathcal{D}_{\text{exp}}[\![E_{fst}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{snd}]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(rec } I_{var}\ E_{body}\texttt{)}]\!] = \texttt{(rec } I_{var}\ \mathcal{D}_{\text{exp}}[\![E_{body}]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(error } I_{msg}\texttt{)}]\!] = \texttt{(error } I_{msg}\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda () } E\texttt{)}]\!] = \texttt{(proc } I_{fresh}\ \mathcal{D}_{\text{exp}}[\![E]\!]\texttt{)}$, where I_{fresh} is fresh
$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I\texttt{) } E\texttt{)}]\!] = \texttt{(proc } I\ \mathcal{D}_{\text{exp}}[\![E]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_1\ I_{rest}{}^{+}\texttt{) } E\texttt{)}]\!] = \texttt{(proc } I_1\ \mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_{rest}{}^{+}\texttt{) } E\texttt{)}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(}E\texttt{)}]\!] = \texttt{(call } \mathcal{D}_{\text{exp}}[\![E]\!]\ \texttt{\#u)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_2\texttt{)}]\!] = \texttt{(call } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_2\ E_{rest}{}^{+}\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{((call } E_1\ E_2\texttt{) } E_{rest}{}^{+}\texttt{)}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(list)}]\!] = \texttt{\#u}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(list } E_1\ E_{rest}{}^{*}\texttt{)}]\!] = \texttt{(pair } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![\texttt{(list } E_{rest}{}^{*}\texttt{)}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(quote \#u)}]\!] = \texttt{\#u}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(quote } B\texttt{)}]\!] = B$
$\mathcal{D}_{\text{exp}}[\![\texttt{(quote } N\texttt{)}]\!] = N$
$\mathcal{D}_{\text{exp}}[\![\texttt{(quote } I\texttt{)}]\!] = \texttt{(symbol } I\texttt{)}$
$\mathcal{D}_{\text{exp}}[\![\texttt{(quote (}SX_1\ \ldots\ SX_n\texttt{))}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(list (quote } SX_1\texttt{) } \ldots\ \texttt{(quote } SX_n\texttt{))}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(cond (else } E_{default}\texttt{))}]\!] = \mathcal{D}_{\text{exp}}[\![E_{default}]\!]$
$\mathcal{D}_{\text{exp}}[\![\texttt{(cond (}E_{test1}\ E_{action1}\texttt{) (}E_{testi}\ E_{actioni}\texttt{)}^{*}\texttt{ (else } E_{default}\texttt{))}]\!]$
$= \texttt{(if } \mathcal{D}_{\text{exp}}[\![E_{test1}]\!]$
$\mathcal{D}_{\text{exp}}[\![E_{action1}]\!]$
$\mathcal{D}_{\text{exp}}[\![\texttt{(cond (}E_{testi}\ E_{actioni}\texttt{)}^{*}\texttt{ (else } E_{default}\texttt{))}]\!]\texttt{)}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(scand } E_{conjunct}{}^{*}\texttt{)}]\!]$ and $\mathcal{D}_{\text{exp}}[\![\texttt{(scor } E_{disjunct}{}^{*}\texttt{)}]\!]$ *Left as exercises.*

$\mathcal{D}_{\text{exp}}[\![\texttt{(let ((}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{)) } E_0\texttt{)}]\!]$
$= \mathcal{D}_{\text{exp}}[\![\texttt{((lambda (}I_1\ \ldots\ I_n\texttt{) } E_0\texttt{) } E_1\ \ldots\ E_n\texttt{)}]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(letrec ((}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{)) } E_0\texttt{)}]\!]$
$= \mathcal{D}_{\text{exp}}[\![\texttt{(call (rec } I_{churchList}$
$\texttt{(proc } I_{selector}$
$\texttt{(}I_{selector}\ \texttt{(}I_{churchList}\ \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) } E_1\texttt{))}$
$\vdots$
$\texttt{(}I_{churchList}\ \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) } E_n\texttt{)))))}$
$\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) } E_0\texttt{))}]\!]$
where $I_{churchList} \neq I_{selector}$ are fresh and $\notin \bigcup_{i\ =\ 0}^{n}$ *FreeIds*$[\![E_i]\!]$

Figure 6.3: Desugaring FL expressions into FLK expressions.

$\mathcal{D}_{\text{prog}}$: $\text{Program}_{FL} \rightarrow \text{Program}_{FLK}$

$\mathcal{D}_{\text{prog}}$ ⟦(fl (I_{formal}*) E_{body} (define I_1 E_1) ... (define I_n E_n))⟧

```
  = (flk (I_formal*)
      D_exp⟦(let ((unit? (lambda (x) (primop unit? x)))
                  (boolean? (lambda (x) (primop boolean? x)))
                    ⋮
                  (+ (lambda (x y) (primop + x y)))
                    ⋮
                  (unit #u)
                  (true #t)
                  (false #f)
                  (cons (lambda (x y) (pair x y)))
                  (car (lambda (p) (primop fst p)))
                  (cdr (lambda (p) (primop snd p)))
                  (null? (lambda (x) (primop unit? x)))
                  (null (lambda () #u))
                  (nil #u)
                  (equal? ...) {Definition of this predicate left as an exercise.})
                  )
             (letrec ((I_1 E_1)
                        ⋮
                      (I_n E_n))
               E_body))⟧
```

Figure 6.4: Desugaring FL programs into FLK programs.

expand any syntactic sugar constructs appearing in the subexpressions. Note that $\mathcal{D}_{\text{exp}}$ acts as the identity function when applied to an FLK expression.

The rules for desugaring multi-abstraction into `proc` and multi-applications into `call` are based on the same currying trick that we use extensively in the metalanguage. (See Exercise 6.15 for an alternative approach to desugaring these constructs.) The recursive `list` desugaring creates a unit-terminated chain of pairs. The recursive `quote` desugaring descends an s-expression tree and builds up a corresponding tree of pairs with constants as leaves. The `cond` construct desugars into a nested sequence of `if`s. The `scand` and `scor` desugarings are left as exercises.

A `let` desugars into an application of an abstraction. This underscores the fact that abstractions are a fundamental means of naming in FL. Note that $E_1 \ \ldots \ E_n$ are outside the scope of $I_1 \ \ldots \ I_n$ and therefore cannot refer to the variables named by these identifiers.

However, in a `letrec`, the $E_1 \ \ldots \ E_n$ are *inside* the scope of $I_1 \ \ldots \ I_n$ and should refer to the variables named by these identifiers. Achieving this effect is challenging. We will present the desugaring in two stages. Suppose that `nth` is a standard identifier bound to a procedure that takes a list and an integer n and returns the nth element of the list (where elements are numbered from 1 up). Then an almost-correct desugaring for `letrec` is:

$\mathcal{D}_{\text{exp}}[\![$`(letrec ((`$I_1$ E_1`) ... (`I_n E_n`)) `E_0`)`$]\!]$
=
$\mathcal{D}_{\text{exp}}[\![$`(let ((`$I_{outer}$ `(rec `I_{inner}
`(let ((`I_1 `(nth `I_{inner} `1))`
$\vdots$
`(`I_n `(nth `I_{inner} n`)))`
`(list `E_1 ... E_n`)))))`
`(let ((`I_1 `(nth `I_{outer} `1))`
$\vdots$
`(`I_n `(nth `I_{outer} n`)))`
E_0`))`$]\!]$
where $I_{outer} \neq I_{inner}$ are fresh identifiers
and I_{outer}, $I_{inner} \notin \bigcup_{i=0}^{n}$ *FreeIds*$[\![E_i]\!]$

FreeIds is defined in Section 6.3.1 and in Figure 6.10. Assume for the moment that I_{outer} and I_{inner} are brand new names that don't conflict with any other names.

The basic idea of the desugaring is this: since `rec` can only find a single fixed point, design that fixed point to be a list of the n fixed points we really want. Inside the `rec`, the value of formal I_{inner} (which is a list of length n)

is destructured into its n elements, and the `list` expression is evaluated in a context where I_i is bound to the ith element of the list. Since `let` and `list` are both non-strict in FL, the solution to the `rec` is nontrivial. The name I_{outer} is bound to the solution of the `rec` and this list of length n is similarly destructured so that the body expression E_0 can be evaluated in a context where each I_i is bound to its individual solution.

The above result is an adequate desugaring, but it is complicated, and its use of the standard identifier `nth` is not only unaesthetic but also can lead to bugs due to name capture. For this reason, we will present an alternative desugaring that is more elegant. This desugaring is based on the same idea but represents lists as procedures. In this representation, which we shall call a **Church list**, an n-element list is a unary procedure whose single argument is an n-argument selector procedure that is applied to the n elements of the list. If $I_{churchList}$ is bound to an n-element Church list, then the application

(*I*churchList `(lambda (`*I*1 ... *I*n`)` *I*i`)`)

extracts the ith element of the list. More generally, the application

(*I*churchList `(lambda (`*I*1 ... *I*n`)` *E*`)`)

returns the value of E in a context where each I_i is bound to the ith element of the list. Church lists give rise to the desugaring for recursive bindings shown in Figure 6.3.

$\mathcal{D}_{\text{prog}}$ is defined by a single clause, which transforms the definitions and body of a program using the `let` and `letrec` constructs. The desugaring makes standard identifiers available to the definitions and the body of the program by using an outer `let` to binding them to functions that perform the corresponding primitive applications via `primop`. Since multi-argument `lambda`s are used in the bindings, functions associated with the binary function names are appropriately curried. For example, in an FL program, `(+ 1)` stands for the incrementing function. The standard identifier `cons` makes FLK's `pair` construct available as a curried FL procedure, and the traditional LISP names `car`, `cdr`, `null?`, and `nil` are provided as synonyms for `fst`, `snd`, `unit?`, and `#u`. There are a few other handy synonyms as well. The mutually recursivene nature of the definitions is implemented by desugaring them into a `letrec`.

Note that $\mathcal{D}_{\text{exp}}$ is applied once again to the result of $\mathcal{D}_{\text{exp}}$ on `letrec` and $\mathcal{D}_{\text{prog}}$ on `program`.

▷ **Exercise 6.1** Provide the missing desugarings for FL's `scand` and `scor` constructs (see Figure 6.3). ◁

▷ **Exercise 6.2** The desugaring for `letrec` in Figure 6.4 requires a pair of fresh identifiers. There is another desugaring for `letrec` that requires no fresh identifiers whatsoever. This desugaring has a recursive structure not exhibited by the other versions. Below is a skeleton of the desugaring.

$\mathcal{D}_{\text{exp}}[\![$`(letrec ((`$I_1$ E_1`) ... (`I_n E_n`)) `E_0`)`$]\!]$
=
$\mathcal{D}_{\text{exp}}[\![$`(let ((`$I_1$ `(rec` I_1 $\square_1$`)) ... (`I_n `(rec` I_n $\square_n$`))) `E_0`)`$]\!]$

where the boxes $\square_i$ are to be filled in appropriately.

a. Give the general form for expressions that fill the boxes $\square_i$ in such a way that the above skeleton defines a correct desugaring for `letrec`.

b. Using your approach, how many `rec`s will appear in a desugaring of a `letrec` with 5 bindings?

c. Give a closed form solution for the number of `rec`s that will appear in a desugaring of a `letrec` with n bindings.

d. Comment on the practicality of this `letrec` desugaring. ◁

▷ **Exercise 6.3** Two constructs are said to be **idempotent** (roughly, "of equal power") if each can be expressed as a desugaring into the other. For example, multi-argument procedures and single-argument procedures are idempotent: multi-argument abstractions and calls can be desugared into single-argument ones via currying; and single-argument abstractions and calls are a special subcase of the multi-argument ones. On the other hand, pairs and procedures are *not* idempotent; although Church's techniques give a desugaring of pairs into procedures, procedure abstractions and calls cannot be desugared into pairs.

We have considered a version of FLK where `rec` is the kernel recursion construct and FL's `letrec` is desugared into `rec`. Show that `rec` and `letrec` are idempotent by providing a desugaring of `rec` into `letrec`. ◁

▷ **Exercise 6.4** Many LISP dialects support an alternative version of `define` for constructing new functions. The syntax is of the form

`(define (` *function-name arg-1 ... arg-n*`)` *function-body*`)`

For example, the squaring function can be defined as:

```
(define (square x) (* x x))
```

Extend the desugaring for FL to handle this syntax. Hint: It is easier to add another processing step for definitions rather than modifying the desugaring of `program` expressions. ◁

▷ **Exercise 6.5** It is often useful for the value of a `let`-bound variable to depend on the value of a previous `let`-bound variable. In the current version of FL, achieving this

behavior requires nested `let` expressions. For example:

```
(let ((r (+ 1 2)))
  (let ((square-r (* r r)))
    (let ((circum (* 2 (* pi square-r))))
      ... code using r, square-r, and circum ...
      )))
```

Many Lisp dialects support a `let*` construct that looks just like `let` except that its variables are guaranteed to be bound to their associated values in the order that they appear in the list of bindings. A val expression in `let*` can refer to the result of a previous binding within the same `let*`. Using `let*`, the above example could be rendered:

```
(let* ((r (+ 1 2))
       (square-r (* r r))
       (circum (* 2 (* pi square-r))))
  ... code using r, square-r, and circum ...
  )
```

Write an appropriate desugaring for `let*`. ◁

▷ **Exercise 6.6** It is common to create locally recursive procedures and then call them immediately to start a process. For example, iterative factorial can be expressed in FL as:

```
(define fact
  (lambda (n)
    (letrec ((iter (lambda (num ans)
                     (if (= num 0)
                         ans
                         (iter (- num 1) (* num ans))))))
      (iter n 1)))))
```

Some versions of Lisp have a "named `let`" or "`let` loop" construct that makes this pattern easier to express. The construct is of the form

(let I_{name} ((I_{var} E_{val})*) E_{body})

It looks like a `let` expression except that it has an additional identifier I_{name}. The n variables I_{var} are first bound to the values E_{val} and then the E_{body} is evaluated in a context where these bindings are in effect *and* the name I_{name} refers to a procedure of the n variables I_{var} that computes E_{body}. Using named let, the iterative factorial construct can be expressed more succinctly as:

```
(define fact
  (lambda (n)
    (let iter ((num n) (ans 1))
      (if (= num 0)
          ans
          (iter (- num 1) (* num ans))))))
```

Extend the desugaring for `let` to handle named `let`. ◁

▷ **Exercise 6.7** In FL, definitions are only allowed within the `program` construct at "top-level"; yet a local form of definition within `lambda` and `let` expressions would often be useful. Generalize the idea of definitions by modifying FL to support local definitions. Design a syntax for your change, and show how to express it in terms of a desugaring. ◁

▷ **Exercise 6.8** Ben Bitdiddle is upset by the desugaring for nullary (i.e., zero-argument) abstractions and applications. He argues (correctly) that, according to the desugarings, the FL expression `((lambda (x) x))` will return `#u`. He believes that evaluating this expression should give an error.

One way to fix this problem is to package up multiple arguments into some sort of data structure. See Exercise 6.15 for an example of this approach. Here we will consider other approaches for handling nullary abstractions and applications.

a. Bud Lojack suggests desugaring `(lambda () `E`)` into E and `(`E`)` into E. Give examples of FL expressions that have a questionable behavior under this desugaring.

b. Paula Morwicz suggests a desugaring in which

$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E\texttt{)}]\!] = \texttt{(call (call } \mathcal{D}_{\text{exp}}[\![E]\!] \texttt{ \#t) \#u)}$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_2\texttt{)}]\!] = \texttt{(call (call } \mathcal{D}_{\text{exp}}[\![E_1]\!] \texttt{ \#f) } \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)}$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_2\ E_{rest}{}^{+}\texttt{)}]\!] = \texttt{((call (call } \mathcal{D}_{\text{exp}}[\![E_1]\!] \texttt{ \#f) } \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{)} \ \mathcal{D}_{\text{exp}}[\![E_{rest}{}^{+}]\!]\texttt{)}$$

   i. Give the corresponding desugarings for multi-abstractions.

   ii. What value does `((lambda (x) x))` have under this desugaring?

c. Ben reasons that the fundamental problem exhibited by the nullary desugarings is that there is no way to call a procedure without passing it an argument. He decides to extend FLK with the following kernel forms for parameterless procedures:

   `(freeze` E`)`: Return a "frozen" value that suspends the evaluation of E.

   `(thaw` E`)`: Unsuspends the expression frozen within a frozen value. Gives an error if called on any value other than one created by `freeze`.

   Show how `freeze` and `thaw` can be used to fix Ben's problem.

d. Sam Antix doesn't like the fact that multi-abstractions and multi-applications both have three desugaring clauses. Figuring that only two clauses should suffice in each case, he develops the following desugaring rules based on Ben's `freeze` and `thaw` commands:

$$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda () } E\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(freeze } E\texttt{)}]\!]$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_1\ I_{rest}\texttt{*) } E\texttt{)}]\!] = \texttt{(proc } I_1\ \mathcal{D}_{\text{exp}}[\![\texttt{(lambda (}I_{rest}\texttt{*) } E\texttt{)}]\!]\texttt{)}$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(thaw } E\texttt{)}]\!]$$
$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_1\ E_{rest}\texttt{*)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{((call } E_1\ E_2\texttt{) } \mathcal{D}_{\text{exp}}[\![E_{rest}\texttt{*}]\!]\texttt{)}]\!]$$

Discuss the strengths and weaknesses of Sam's desugaring. ◁

▷ **Exercise 6.9†** Show that a desugaring process based on the rules in Figures 6.3 and 6.4 is guaranteed to terminate. ◁

6.2.3 The FL Standard Library

The FL standard library is shown in Figure 6.5. All of FLK's primitives (those names that can be used in `primop`) are included as curried procedures. Note that FL only supports integers and not floating point numbers, so arithmetic operations like `+`, `*`, `<=`, etc. only work on integers. It would be straightforward to extend FL to support floating point numbers, and in some code examples it will be convenient to assume that FL does support floating point numbers. In such examples, we will use arithmetic operation names prefixed with `f` to indicate floating point operations: e.g., `f+`, `f*`, and `f<=`.

The standard library also includes a number of other standard identifiers that are convenient, such as constants (`unit`, `true`, `false`, and `nil`), SCHEME-style operations on lists (`cons`, `car`, `cdr`, `null?`), a generic binary equality tester (`equal?`) that tests for equality between any two FL values that are not procedures.

6.2.4 Examples

Although FL is a toy language, it packs a fair bit of expressive punch. In this section, we illustrate the expressive power of FL in the context of a few examples.

6.2.4.1 List Utilities

As a simple example of FL procedures, consider the list procedures in Figure 6.6. The `list?` procedure takes a value and determines if it is a list – i.e., a sequence of pairs terminated with the unit value. The `length` procedure returns the length

Primitives (can be used in `primop`):

`unit?`	Unary type predicate for the unit value.
`boolean?`	Unary type predicate for booleans.
`integer?`	Unary type predicate for integers.
`symbol?`	Unary type predicate for symbols.
`procedure?`	Unary type predicate for procedures (i.e., a functional value).
`pair?`	Unary type predicate for pairs.
`not?`	Unary boolean negation.
`and?`	Binary boolean conjunction (not short-circuit).
`or?`	Binary boolean disjunction (not short-circuit).
`bool=?`	Binary boolean equality predicate.
`+`	Binary integer addition.
`-`	Binary integer subtraction.
`*`	Binary integer multiplication.
`/`	Binary integer division.
`%`	Binary integer remainder.
`=`	Binary integer equality predicate.
`!=`	Binary integer inequality predicate.
`<`	Binary integer less-than predicate.
`<=`	Binary integer less-than-or-equal-to predicate.
`>`	Binary integer greater-than predicate.
`>=`	Binary integer greater-than-or-equal-to predicate.
`sym=?`	Binary symbol equality.
`fst`	Unary selector of the first element of a given pair.
`snd`	Unary selector of the second element of a given pair.

Other Standard Identifiers:

`unit`	The unit value.
`true`	Boolean truth.
`false`	Boolean falsity.
`cons`	Binary list constructor.
`car`	Unary list selector — head of list.
`cdr`	Unary list selector — tail of list.
`nil`	The empty list (synonym for the unit value).
`null`	Unary empty list constructor.
`null?`	Unary empty list predicate.
`equal?`	Generic binary equality test

Figure 6.5: FL Standard Library.

of a list. The `member?` procedure determines if a value is an element of a list. The `merge` procedure takes a less-than-or-equal-to predicate `leq` and two lists `xs` and `ys` that are assumed to be sorted according to this predicate and returns the sorted list containing all the elements of both lists (including duplicates, if any). The `alts` procedure returns a pair of (1) all the odd-indexed[3] elements and (2) all the even-indexed elements of a given list, preserving the relative order of elements in each sublist. The `merge-sort` procedure takes an ordering predicate and a list of elements and returns a list of the same elements ordered according to the ordering predicate.

Here are some sample uses of these procedures:

```
(list? 17)  ⟶FL  false
(list? (list 7 2 5))  ⟶FL  true
(list? (pair 3 (pair 4 5)))  ⟶FL  false
```

```
(length (list))  ⟶FL  0
(length (list 7 2 5))  ⟶FL  3
```

```
(member? 2 (list 7 2 5))  ⟶FL  true
(member? 17 (list 7 2 5))  ⟶FL  false
(member? '* '(+ - * /))  ⟶FL  true
```

```
(merge < (null) (list 3 4 6))  ⟶FL  [3, 4, 6]
(merge < (list 1 6 8) (list 3 4 6))  ⟶FL  [1, 3, 4, 6, 6, 8]
```

```
(alts (null))  ⟶FL  ⟨[], []⟩
(alts (list 7))  ⟶FL  ⟨[7], []⟩
(alts (list 7 2))  ⟶FL  ⟨[7], [2]⟩
(alts (list 7 2 4 5 1 4 3))  ⟶FL  ⟨[7, 4, 1, 3], [2, 5, 4]⟩
```

```
(merge-sort <= (list 7 2 4 1 5 4 3))  ⟶FL  [1, 2, 3, 4, 4, 5, 7]
(merge-sort >= (list 7 2 4 1 5 4 3))  ⟶FL  [7, 5, 4, 4, 3, 2, 1]
(merge-sort (lambda (a b) (<= (% a 4) (% b 4)))
            (list 7 2 4 1 5 4 3))  ⟶FL  [4, 4, 1, 5, 7, 2, 3]
```

[3] Assume that list elements are indexed starting with 1.

6.2.4.2 An ELM Interpreter

As a more interesting example of an FL program, in Figure 6.7 we use FL to write an interpreter for the ELM subset of the EL language (Exercise 3.10). Recall that ELM is EL without conditional and boolean expressions. The `elm-eval` procedure evaluates an ELM expression relative to a list of numbers, `args`, which are the the program inputs. ELM expressions are represented as FL s-expressions. `elm-eval` is written as a dispatch on the type of expression, which is determined by the syntax predicates `lit?`, `arg?`, and `arithop?`. The selectors `lit-num`, `arg-index`, `arithop-op`, `arithop-rand1`, `arithop-rand2` extract components of syntax nodes. The `arg-index` procedure returns the `index`th element of the given list `nums` (where indices are assumed to start at 1). The `primop->proc` procedure converts a symbol (such as `'+`) to a binary FL procedure (such as the addition procedure `+`).

Here are some examples of the `elm-eval` procedure in action:

```
(elm-eval '(* (arg 1) (arg 1)) '(5)) ⟶FL 25
(elm-eval '(/ (+ (arg 1) (arg 2)) 2) '(6 8)) ⟶FL 7
(elm-eval '(+ (arg 1) (arg 2)) '(3)) ⟶FL error:arg-index-out-of-bounds
```

6.2.4.3 A Pattern Matcher

Programs that match a pattern against a tree structure are so useful that they should be part of every programmer's bag of tricks. Figures 6.8 and 6.9 present a simple pattern matching program written in FL.

The pattern matcher manipulates trees represented as s-expressions. Patterns are trees whose leaves are either constants (unit, booleans, integers, or symbols) or pattern variables. We represent the pattern variable named *I* by the s-expression `(? I)`. Because of this convention, the symbol `?` is considered special and should never be used as one of the symbol constants in the pattern or the structure being matched. Examples of legal patterns include:

```
(? pat)
(The (? adjective) programmer (? adverb) hacked (? noun))
((? a) is equal to (? a))
(((? a) (? b)) is the reflection of ((? b) (? a)))
```

A pattern *p* matches an s-expression *s* if there is some set of bindings between

```
(define list?
  (lambda (val)
    (scor (null? val)
          (scand (pair? val) (list? (snd val))))))

(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst))))))

(define member?
  (lambda (elt lst)
    (scand (not (null? lst))
           (scor (equal? elt (car lst))
                 (member? elt (cdr lst))))))

(define merge
  (lambda (leq xs ys)
    (cond ((null? xs) ys)
          ((null? ys) xs)
          ((leq (car xs) (car ys))
           (cons (car xs) (merge leq (cdr xs) ys)))
          (else
           (cons (car ys) (merge leq xs (cdr ys)))))))

(define alts
  (lambda (ws)
    (if (null ws)
        (pair (null) (null))
        (let ((alts-rest (alts (cdr ws))))
          (pair (cons (car ws) (snd alts-rest))
                (fst alts-rest))))))

(define merge-sort
  (lambda (leq zs)
    (if (scor (null? zs) (null? (cdr zs)))
        zs
        (let ((split (alts zs)))
          (merge (fst split) (snd split))))))
```

Figure 6.6: Some list procedures written in FL.

```
(fl (pgm args)
  (cond ((not? (elm-program pgm)) (error ill-formed-program))
        ((not? (list? args)) (error ill-formed-argument-list))
        ((not? (= (elm-nargs pgm) (length args)))
         (error wrong-number-of-args))
        (else  (elm-eval (elm-body pgm) args)))

  (define elm-eval
    (lambda (exp args)
      (cond ((lit? exp) (lit-num exp))
            ((arg? exp) (get-arg (arg-index exp) args))
            ((arithop? exp)
            ((primop->proc (arithop-op exp))
             (elm-eval (arithop-rand1 exp) args)
             (elm-eval (arithop-rand2 exp) args)))
            (else (error illegal-expression)))))

  (define get-arg
    (lambda (index nums)
      (cond ((scor (<= index 0) (null? nums))
             (error arg-index-out-of-bounds))
            ((= index 1) (car nums))
            (else (get-arg (- index 1) (cdr nums))))))

  (define primop->proc
    (lambda (sym)
      (cond ((sym=? sym '+) +) ((sym=? sym '-) -)
            ((sym=? sym '*) *) ((sym=? sym '/) /)
            (else (error illegal-op)))))

  ;; Abstract syntax
  (define elm-program?
    (lambda (sexp)
      (scand (list? sexp) (= (length sexp) 3) (sym=? (car exp) 'elm))))
  (define elm-program-nargs (lambda (sexp) (car (cdr sexp))))
  (define elm-program-body (lambda (sexp) (car (cdr (cdr sexp)))))
  (define lit? integer?)
  (define lit-num (lambda (lit) lit))
  (define arg?
    (lambda (exp)
      (scand (list? exp) (= (length exp) 2) (sym=? (car exp) 'arg))))
  (define arg-index (lambda (exp) (car (cdr exp))))
  (define arithop?
    (lambda (exp)
      (scand (list? exp) (= (length exp) 3) (member? (car exp) '(+ - * /)))))
  (define arithop-op (lambda (exp) (car exp)))
  (define arithop-rand1 (lambda (exp) (car (cdr exp))))
  (define arithop-rand2 (lambda (exp) (car (cdr (cdr exp)))))

  ;; List utilities
  (define list? (lambda (sexp) ...))
  (define length (lambda (xs) ...))
  (define member? (lambda (x xs) ...))
```

Figure 6.7: An interpreter for ELM, a subset of EL.

pattern variables and s-expressions such that instantiating the variables with their bindings in p yields s. Constraints on the form of the pattern can be specified by using the same pattern variable in more than one place.

For example, consider the pattern `((? a) is equal to (? a))`. It matches the following two s-expressions:

```
(1 is equal to 1)   and
((Ben Bitdiddle) is equal to (Ben Bitdiddle))
```

but not

```
(1 is equal to 2)   or
(Ben Bitdiddle is equal to Ben Bitdiddle)
```

In the final example, `Ben Bitdiddle` is two s-expressions and cannot be matched by a single pattern.

The entry point of the pattern matcher is the `match-sexp` procedure, which takes a pattern and an s-expression as arguments. If the pattern does not match the s-expression, `match-sexp` returns the symbol `*failed*`. Otherwise, `match-sexp` returns a dictionary structure that contains pattern variable bindings that make the match successful. `match-sexp` just passes responsibility to `match-with-dict`, which does the real work.

In addition to a pattern and an s-expression, `match-with-dict` takes a dictionary. It matches the pattern to the s-expression in the context of the dictionary. That is, any match of a variable in the pattern must be consistent with the binding that is already in the dictionary. In high-level terms, `match-with-dict` performs a left-to-right depth-first walk in lock-step over both the pattern tree and s-expression tree. A dictionary representing the bindings of variables seen so far flows along this depth-first path. Along the path, the matching process checks whether:

- an internal node of the pattern tree has the same number of subtrees as the corresponding internal node of the s-expression.
- a constant leaf in the pattern is matched by exactly the same constant leaf in the corresponding position of the pattern.
- a variable leaf in the pattern is matched by an s-expression that is consistent with the bindings represented by the current dictionary.

A successful check allows the dictionary to flow to the next part of the path, possibly extended with a new binding. After an unsuccessful check, the dictionary is replaced by a failure symbol that propagates through the rest of the path.

```
(define match-sexp (lambda (pat sexp)
                     (match-with-dict pat sexp (dict-empty))))

(define match-with-dict
  (lambda (pat sexp dict)
    (cond ((failed? dict) dict)    ; Propagate failures.
          ((null? pat)
           (if (null? sexp)
               dict                ; PAT and SEXP both ended.
               (fail)))            ; PAT ended but SEXP didn't.
          ((null? sexp) (fail))    ; SEXP ended but PAT didn't.
          ((pattern-constant? pat)
           (if (sexp=? pat sexp) dict (fail)))
          ((pattern-variable? pat)
           (dict-bind (pattern-variable-name pat) sexp dict))
          (else (match-with-dict (cdr pat)
                                 (cdr sexp)
                                 (match-with-dict (car pat)
                                                  (car sexp)
                                                  dict))))))

(define pattern-variable?
  (lambda (pat) (if (pair? pat)
                    (sexp=? (car pat) '?)
                    #f)))

(define pattern-variable-name (lambda (sexp) (car (cdr sexp))))

(define pattern-constant?
  (lambda (p) (or (symbol? p) (integer? p) (boolean? p) (unit? p))))

(define fail (lambda () '*failed*))
(define failed? (lambda (dict) (sexp=? dict '*failed*)))
```

Figure 6.8: A pattern matcher in FL, part 1.

There are many possible representations for dictionaries. We represent a dictionary as a list of bindings, where each binding is a pair of a pattern variable identifier and the associated s-expression.

Examples of using `match-sexp`:

```
(match-sexp '(a short sentence) '(a short sentence))
```
$\xrightarrow[FL]{}$ [] {*Match succeeds with the empty dictionary.*}

```
(match-sexp '(a short sentence) '(a longer sentence))
```
$\xrightarrow[FL]{}$ *'*failed** {*Match failed.*}

```
(match-sexp '((? article) (? adjective) sentence)
            '(a longer sentence))
```
$\xrightarrow[FL]{}$ $[\langle 'article, 'a\rangle, \langle 'adjective, 'longer\rangle]$

```
;; Can make use of FL's currying
(define m1 (match-sexp '((a (b (? c))) (((? c) b) a))))

(m1 '((a (b (c (d)))) (((c (d)) b) a)))
```
$\xrightarrow[FL]{}$ $[\langle 'c, ['c, ['d]]\rangle]$

```
(m1 '((a (b (c (d)))) ((((d) c) b) a)))
```
$\xrightarrow[FL]{}$ *'*failed**

6.3 Variables and Substitution

Intuitively, the meaning of an FLK abstraction `(proc` I E`)` shouldn't depend on the particular name chosen for I, which is known as its **formal parameter**. Just as we expect the meaning of an integral to be independent of the choice of the variable of integration (so that $\int_a^b f(x)dx = \int_a^b f(y)dy$), we expect the meaning of an FLK abstraction to be invariant under a change to the name of its variable. Thus, the identity abstraction `(proc a a)` should also be expressible as `(proc x x)` or `(proc square square)`. Furthermore, the variable references named by `a`, `x`, and `square` are logically distinct from any variable references coincidentally sharing the same name in other expressions.

This section formalizes this intuition about variables in FLK expressions.

6.3.1 Terminology

First, it's important to tease apart several related but distinct concepts in our terminology concerning names. We reserve the word **variable** for the logical entity that is introduced by an abstraction and is referenced by a variable reference. The word **identifier** designates the name that stands for a given variable within an expression. The identity abstraction discussed above has a

```
;;; Dictionaries
(define dict-bind
  (lambda (sym1 sexp dict)
    (let ((value (dict-lookup sym1 dict)))
      (cond ((unbound? value)
             (dict-adjoin-binding (binding-make sym1 sexp) dict))
            ((sexp=? value sexp) dict)
            (else (fail))))))

(define dict-lookup
  (lambda (name dict)
    (cond ((dict-empty? dict) (unbound))
          ((sym=? name (binding-name (dict-first-binding dict)))
           (binding-value (dict-first-binding dict)))
          (else (dict-lookup name (dict-rest-bindings dict))))))

(define dict-empty (lambda () (list)))
(define dict-empty? null?)
(define dict-adjoin-binding cons)
(define dict-first-binding car)
(define dict-rest-bindings cdr)
(define unbound (lambda () '*unbound*))
(define unbound? (lambda (sym) (sexp=? sym '*unbound*)))

;;; Bindings
(define binding-make cons)
(define binding-name car)
(define binding-value cdr)

;; Utilities
(define sexp=?
  (lambda (obj1 obj2)
    (cond ((unit? obj1) (unit? obj2))
          ((and (boolean? obj1) (boolean? obj2)) (boolean=? obj1 obj2))
          ((and (integer? obj1) (integer? obj2)) (= obj1 obj2))
          ((and (symbol? obj1) (symbol? obj2)) (sym=? obj1 obj2))
          ((and (pair? obj1) (pair? obj2))
           (and (sexp=? (car obj1) (car obj2))
                (sexp=? (cdr obj1) (cdr obj2))))
          (else #f))))
```

Figure 6.9: A pattern matcher in FL, part 2.

single variable, and the identifier that names it is arbitrary. In the expression `(proc x (call x (proc x x)))` there are two logically distinct variables, but they happen to be named by the same identifier.

Sometimes it is useful to distinguish different **occurrences** of an identifier or subexpression within an expression. In the expression

```
(call (proc x x) (proc x x))
```

there are four occurrences of the identifier `x` and two occurrences of the subexpression `(proc x x)`. In order to refer to a particular occurrence, we can imagine that each distinct identifier or expression has been numbered from left to right starting with 1. Thus, we could view the above application as

$$\texttt{(call (proc x}^1\ \texttt{x}^2\texttt{)}^1\ \texttt{(proc x}^3\ \texttt{x}^4\texttt{)}^2\texttt{)}\ ,$$

where the superscripts distinguish the occurrences of an identifier or subexpression. When we say "the ith occurrence of `x`" we mean $\texttt{x}^i$.

We shall say that the formal parameter I appearing in an FLK abstraction `(proc` I E`)` is a **binding occurrence** of I and that the abstraction **binds** I. An occurrence of an identifier in an FLK expression I is **bound** if it is a binding occurrence or it occurs in the body of some abstraction that binds I; otherwise, that occurrence of the identifier is said to be **free**. For example, in `(proc a (proc b (call a c)))`, the single occurrence of `b` and both occurrences of `a` are bound, while the single occurrence of `c` is free. The freeness or boundness of an identifier occurrence depends on the context in which the identifier is viewed. Thus, in the previous example, the second occurrence of `a` is free in `(call a c)` and in `(proc b (call a c))` but not in `(proc a (proc b (call a c)))`. It is possible in one expression to have some occurrences of an identifier that are bound and other occurrences of the same identifier that are free. In `(call (proc a a) a)` the first and second occurrences of `a` are bound, while the third occurrence is free.

An identifier (as opposed to an *occurrence* of an identifier) is said to be a **free identifier** (likewise, **bound**) in an expression if at least one of its occurrences is free (likewise, bound) in the expression. For instance, in the expression

```
(call b (proc a (proc b (call a c))))
```

`a` and `b` are bound identifiers and `b` and `c` are free identifiers. Similarly, a variable is said to be **free** (likewise, **bound**) in an expression if the identifier that names it is free (likewise, bound). Note that an identifier may be both bound and free in an expression, but a variable can only be one or the other. An expression is **closed** if it contains no free identifiers (or, equivalently, no free variables). Expressions with free variables often arise when considering subexpressions of a

given expression. For instance, in the subexpression `(proc b (call b a))` of the closed expression `(proc a (proc b (call b a)))`, the identifier `a` names a free variable.

Using definition by structural induction, it is straightforward to define functions *FreeIds* and *BoundIds* that map FLK expressions to sets of their free and bound identifiers, respectively. These functions are presented in Figure 6.10. Both functions have signature

$$\text{Exp} \to \mathcal{P}(\text{Identifier})$$

where $\mathcal{P}$(Identifier) is the power set (set of all subsets) of Identifier. For example,

$$\begin{aligned} \mathit{FreeIds}[\![\texttt{(call b (proc a (proc b (call a c))))}]\!] &= \{\texttt{b}, \texttt{c}\} \\ \mathit{BoundIds}[\![\texttt{(call b (proc a (proc b (call a c))))}]\!] &= \{\texttt{a}, \texttt{b}\} \end{aligned}$$

One subtle note deserves mention. An I that appears within double brackets on the left hand side of the definitions stands for a variable reference that is an element of the syntactic domain Exp. On the other hand, an unbracketed I on the right hand side of the definitions stands for an element of the syntactic domain Identifier.

▷ **Exercise 6.10** For each of the following FLK expressions:

- Indicate for every occurrence of an identifier whether it is bound or free.
- Determine the free identifiers and bound identifiers of the expression.

a. `(proc x (call x y))`

b. `(call (proc z (proc x (call (call x y) z))) z)`

c. `(call z (proc y (call (proc z (call x y)) z)))`

d. `(proc x (call (call (proc y (call (proc z (call x r)) y)) y) z))` ◁

6.3.2 General Properties of Variables

Throughout mathematical and computational notation, variables serve as syntactic placeholders that range over some set of semantic entities. Variables are manipulated in two different kinds of expressions:

1. A **variable declaration** introduces a new placeholder into an expression.
2. A **variable reference** uses a placeholder within an expression.

$$
\begin{aligned}
\mathit{FreeIds} &: \text{Exp} \to \mathcal{P}(\text{Identifier}) \\
\mathit{FreeIds}[\![L]\!] &= \{\} \\
\mathit{FreeIds}[\![I]\!] &= \{I\} \\
\mathit{FreeIds}[\![\texttt{(primop } O\ E_1\ \ldots\ E_n\texttt{)}]\!] &= \bigcup_{i=1}^{n} \mathit{FreeIds}[\![E_i]\!] \\
\mathit{FreeIds}[\![\texttt{(proc } I\ E\texttt{)}]\!] &= \mathit{FreeIds}[\![E]\!] - \{I\} \\
\mathit{FreeIds}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] &= \mathit{FreeIds}[\![E_1]\!] \cup \mathit{FreeIds}[\![E_2]\!] \\
\mathit{FreeIds}[\![\texttt{(if } E_1\ E_2\ E_3\texttt{)}]\!] &= \mathit{FreeIds}[\![E_1]\!] \cup \mathit{FreeIds}[\![E_2]\!] \\
&\qquad \cup \mathit{FreeIds}[\![E_3]\!] \\
\mathit{FreeIds}[\![\texttt{(pair } E_1\ E_2\texttt{)}]\!] &= \mathit{FreeIds}[\![E_1]\!] \cup \mathit{FreeIds}[\![E_2]\!] \\
\mathit{FreeIds}[\![\texttt{(rec } I\ E\texttt{)}]\!] &= \mathit{FreeIds}[\![E]\!] - \{I\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{BoundIds} &: \text{Exp} \to \mathcal{P}(\text{Identifier}) \\
\mathit{BoundIds}[\![L]\!] &= \{\} \\
\mathit{BoundIds}[\![I]\!] &= \{\} \\
\mathit{BoundIds}[\![\texttt{(primop } O\ E_1\ \ldots\ E_n\texttt{)}]\!] &= \bigcup_{i=1}^{n} \mathit{BoundIds}[\![E_i]\!] \\
\mathit{BoundIds}[\![\texttt{(proc } I\ E\texttt{)}]\!] &= \{I\} \cup \mathit{BoundIds}[\![E]\!] \\
\mathit{BoundIds}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] &= \mathit{BoundIds}[\![E_1]\!] \cup \mathit{BoundIds}[\![E_2]\!] \\
\mathit{BoundIds}[\![\texttt{(if } E_1\ E_2\ E_3\texttt{)}]\!] &= \mathit{BoundIds}[\![E_1]\!] \cup \mathit{BoundIds}[\![E_2]\!] \\
&\qquad \cup \mathit{BoundIds}[\![E_3]\!] \\
\mathit{BoundIds}[\![\texttt{(pair } E_1\ E_2\texttt{)}]\!] &= \mathit{BoundIds}[\![E_1]\!] \cup \mathit{BoundIds}[\![E_2]\!] \\
\mathit{BoundIds}[\![\texttt{(rec } I\ E\texttt{)}]\!] &= \{I\} \cup \mathit{BoundIds}[\![E]\!]
\end{aligned}
$$

Figure 6.10: Definition of the free and bound identifiers of a FLK expression.

The region of an expression in which a particular variable may be referenced is called the **scope** of that variable.

In standard notations, variables are typically represented by identifiers, and declarations and references are distinguished in the format of expressions. For example, compare how variables are declared and referenced in notations for FLK, integration, summation, union, and logical quantification (in each case, the declaring occurrence of the variable x has been boxed):

`(proc` $\boxed{\texttt{x}}$ `x)` $\int_a^b x\, d\boxed{x}$ $\sum_{\boxed{x}=1}^{n} x^2$ $\bigcup_{\boxed{x}\in A} x$ $\forall \boxed{x}.f(x) = g(x)$

Notations in which variables are represented by identifiers share the following properties:

1. Modulo certain restrictions to be discussed shortly, it is possible to consistently rename a variable within its scope without changing the meaning of the entire expression. Thus, in each of the above notations, the x can be changed to y without changing the meaning:

   `(proc y y)` $\int_a^b y\, dy$ $\sum_{y=1}^{n} y^2$ $\bigcup_{y\in A} y$ $\forall y.f(y) = g(y)$

2. Within the scope S of a variable I, the declaration of a new variable with the same name I creates a new scope S' in which the outer variable cannot be referenced. The region S' is called a **hole in the scope** of S. For example, any reference to x within the empty box ($\Box$) in the following examples would refer to the variable declared by the inner x, not the outer x.

   `(proc x (call x (proc x □)))` $\int_a^b x \cdot \left(\int_c^x \Box\, dx\right) dx$ $\prod_{x=1}^{n}\left(\sum_{x=1}^{x}\Box\right)$

   $\bigcup_{x\in A}\langle x, \bigcap_{x\in B}\Box\rangle$ $\forall x.((f(x) = g(x)) \wedge \exists x.\Box)$

6.3.3 Abstract Syntax DAGs and Stoy Diagrams

The chief structural feature of variables is that they permit sharing in an expression: the same variable introduced by a declaration can be used by many variable reference occurrences. We have said before that syntactic expressions can be viewed as abstract syntax trees, but since trees allow no sharing of substructure, they are inadequate for illustrating the sharing nature of variables.

We need the more general directed acyclic graph (DAG) to faithfully show the structure of an expression with variables.

As an example, consider the following FLK expression:[4]

```
(call (proc a (call a a)) (proc a (proc b a)))
```

In this expression, there are two distinct variables named `a`, and the variable named by `b` is declared without being referenced. Figure 6.11 shows an abstract syntax DAG corresponding to this expression. In the DAG, the three distinct variables in the expression are represented by distinct nodes labeled `variable`.

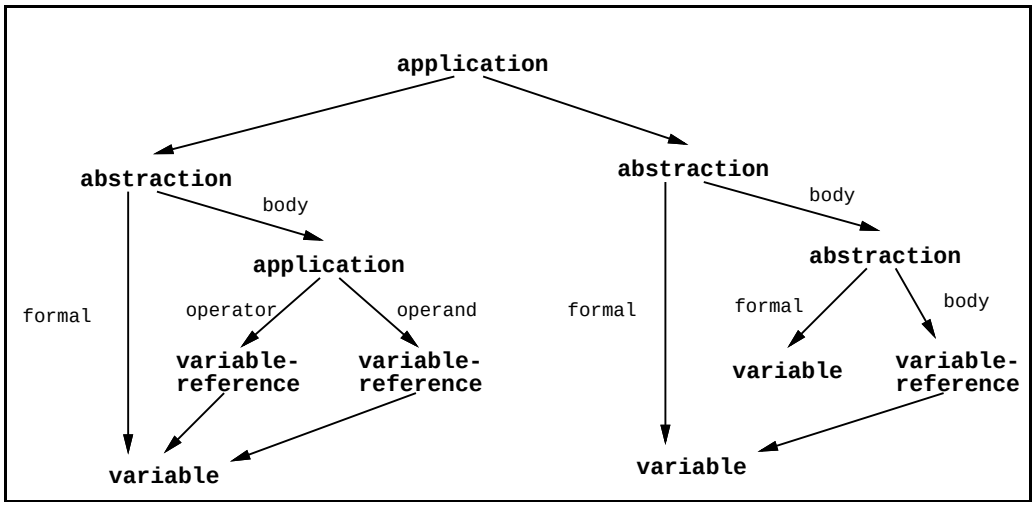


Figure 6.11: Abstract syntax DAG for `(call (proc a (call a a)) (proc a (proc b a)))`

Since sharing is explicit in the structure of the DAG, no identifiers are necessary in the DAG representation of the expression. The key reason variables are traditionally represented with identifiers is that they allow DAGs to be encoded within linear and tree-based notational frameworks. Unfortunately, encodings of DAGs based on identifiers complicate reasoning about expressions because of incidental properties of the identifiers. For example, the notion of a "hole in the scope" introduced earlier is not inherent in the nature of variables, but is a side effect of the fact that when variables are represented by identifiers, a nested pair of variables can accidentally share the same name. We'll see below that identifiers are the major sore spot when defining notions of renaming and substitution on FLK expressions.

[4]In the following discussion, we shall focus only on FLK expressions, but the same techniques could be applied to any notation using variables.

Every closed expression can always be represented by a DAG with no identifiers. However, expressions containing free variables pose a problem because they contain references to a variable without also containing its declaration. Since expressions with free variables are common, we'd like to handle them within the DAG framework. The DAG representation must include the names of any free identifiers because the names of free identifiers actually matter (for example, the expression `(proc b (call b a))` does not have the same meaning as `(proc b (call b c))` in every context). Figure 6.12 shows the DAG representation of `(proc b (call b a))`. The free variable is declared by a special free variable node annotated with the name of the variable.

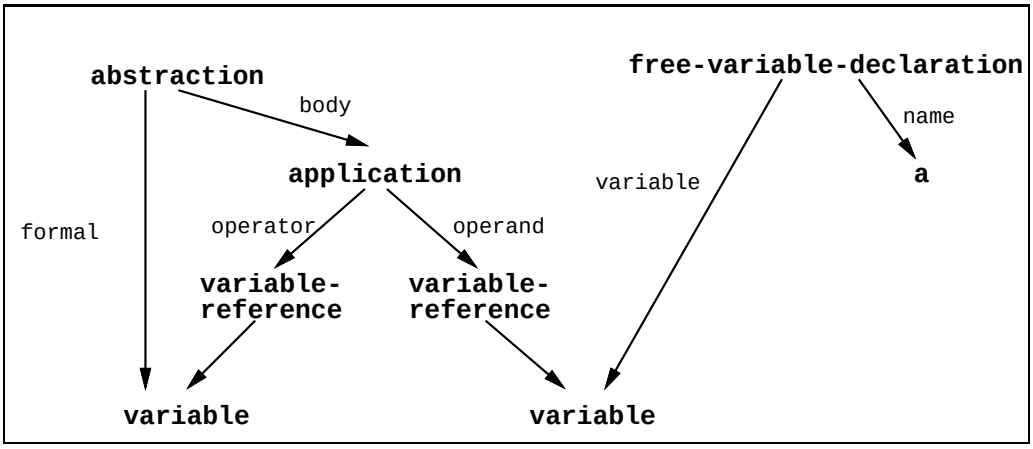
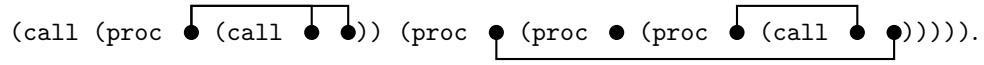


Figure 6.12: Abstract syntax DAG for `(proc b (call b a))`

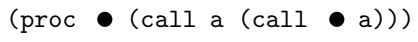
Abstract syntax DAGs take up a lot of real estate on the printed page, so we shall use a more compact notation due to Joseph Stoy [Sto85]. Stoy's notation is a kind of wiring diagram for expressions in which the position corresponding to a variable reference is connected by a wire to the position corresponding to the variable declaration. For example, a **Stoy diagram** for the expression

```
(call (proc a (call a a)) (proc a (proc b (proc c (call c a)))))
```

is



We extend Stoy's notation to handle free variables by simply leaving every free variable reference where it occurs in the expression. Thus, the Stoy diagram for `(proc b (call a (call b a)))` is:



Observe that all identifiers sharing the same name in a Stoy diagram must name the same free variable.

6.3.4 Alpha-Equivalence

Since we really care about the implied DAG structure of an expression and not the vagaries of particular choices of identifiers for variable names, it is natural to equate FLK expressions that share the same DAG representation. We shall use the notation

$$E_1 =_\alpha E_2$$

(pronounced "E_1 is alpha-equivalent to E_2") to mean that E_1 and E_2 designate the same abstract syntax DAG. Thus,

```
(proc a (proc b (call b a)))  =α  (proc b (proc a (call a b)))
                              =α  (proc one
                                    (proc two (call two one)))
```

and

```
(proc b (call b a)) =α (proc c (call c a))
```

but

```
(proc a (proc b (call b a))) ≠α (proc a (proc a (call a a)))
```

and

```
(proc b (call b a)) ≠α (proc b (call b c))
```

Since alpha-equivalence is an equivalence relation, it partitions FLK expressions into equivalence classes that share the same DAG. We shall generally assume throughout the rest of our discussion on FLK that each FLK expression serves as a representative of its equivalence class and that syntactic manipulations on expressions are functions on these equivalence classes rather than on individual expressions. For example, *FreeIds* is a well-defined function not only on FLK expressions but also on alpha-equivalence classes of FLK expressions because

$$E_1 =_\alpha E_2$$

implies

$$\mathit{FreeIds}[\![E_1]\!] = \mathit{FreeIds}[\![E_2]\!].$$

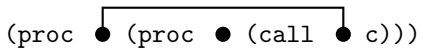
On the other hand, *BoundIds* is not a meaningful function on alpha-equivalence classes because it depends on syntactic details of an expression that are not represented in its DAG structure. Thus `(proc a a)` $=_\alpha$`(proc b b)`, but

$$\mathit{BoundIds}[\![\texttt{(proc a a)}]\!] = \{\texttt{a}\} \neq_\alpha \{\texttt{b}\} = \mathit{BoundIds}[\![\texttt{(proc b b)}]\!]\ .$$

6.3.5 Renaming and Variable Capture

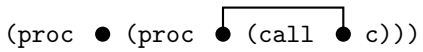
Equipped with a deeper understanding of the structure of variables, we're ready to consider the subtleties of renaming a variable introduced by an abstraction. A correct variable renaming is one that preserves the alpha-equivalence class of the expression — i.e., does not alter its abstract syntax DAG or Stoy diagram. The naïve approach of consistently renaming the declaration occurrence of the variable and all its references is not always appropriate because of a situation known as **variable capture**. There are two kinds of variable capture, both of which will be illustrated in the following example.

Consider the expression `(proc a (proc b (call a c)))`, whose Stoy diagram is shown below:



Suppose we want to rename the variable named `a` in this expression. For almost all possible identifiers, a simple consistent renaming will do. For example, renaming `a` to `x` produces the expression `(proc x (proc b (call x c)))` which has the same Stoy diagram as the original.

Suppose, however, that we choose the identifier `b` as the new name for `a`. Then the naïve renaming method yields `(proc b (proc b (call b c)))`, whose Stoy diagram,



is *not* the same as that for the original expression. The inner binding occurrence of `b` has created a hole in the scope of the outer binding occurrence of `b` in which the outer `b` cannot be seen. Because an inner abstraction just happens to bind the new name, all references to the new name within the body of the inner abstraction are accidentally captured by that abstraction. We shall refer to this situation as **internal variable capture**.

A slightly different problem is encountered if we choose `c` as the new name for `a`. In that case, naïve renaming yields `(proc c (proc b (call c c)))`, whose Stoy diagram is

The free identifier `c` has accidentally been captured by the declaration occurrence of the new name. Here the declaration of the new name has captured a free identifier in the body of the renamed abstraction; above, the internal abstraction captured a reference to the renamed variable. Since the captured

variable is declared external to the renamed abstraction, we shall refer to this second situation as **external variable capture**.

Internal and external variable capture are not unique to FLK. They can occur in any naming system in which logically distinct variables can accidentally be identified. As we shall see later, variable capture commonly rears its ugly head in languages supporting dynamic scoping or macro expansion.

We would like it to be the case that such coincidental choices of identifiers in renamings do not destroy the structural integrity of an FLK expression. One way of doing this is to guarantee that each new variable name introduced by a renaming appears nowhere else in the FLK expression. However, this approach is overly restrictive and gives little insight into the true nature of the problem. Below, we shall precisely define a general syntactic renaming operator that avoids both forms of variable capture.

6.3.6 Substitution

Variable renaming is a special case of a more general syntactic operation on FLK expressions called **substitution**. It is often desirable to substitute a given expression for all free variable references of the variable named by a given identifier within another expression. For example, we might want to replace each free `a` within

```
(call a (proc b (call (proc a (call a b)) a)))
```

by the application `(call c d)` to yield

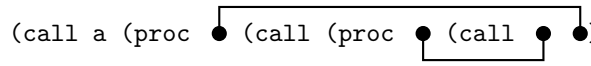
```
(call (call c d) (proc b (call (proc a (call a b)) (call c d))))
```

We use the notation $[E/I]$ to denote a function that maps a given expression into another expression in which E has been substituted for all free variable references named by I. Thus, $[E_1/I]E_2$ denotes the result of substituting E_1 for the free occurrences of I in E_2. Using this notation, the above example can be expressed as:

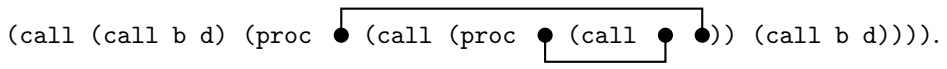
```
[(call c d)/a](call a (proc b (call (proc a (call a b)) a)))
= (call (call c d) (proc b (call (proc a (call a b)) (call c d))))
```

A correct substitution is one which preserves the logical structure both of the expression being substituted (E_1) and the expression substituted into (E_2) — except, of course, for the free variable being substituted for. Although substitution might seem like a straightforward idea, it is plagued with variable capture subtleties similar to those that lurk in renaming. In fact, several well-known logicians gave incorrect definitions for substitution before a correct one was found.

As an example of a problematic situation, suppose that `(call b d)` rather than `(call c d)` were being substituted for `a` in the above example. Since the expression being substituted into has the Stoy diagram



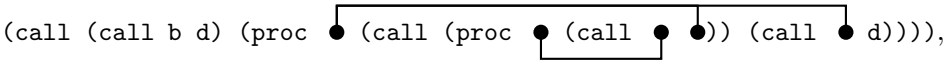
`[(call b d)/a](call a (proc b (call (proc a (call a b)) a)))` should have the Stoy diagram



However, a naïve syntactic approach to substitution would yield the expression

```
(call (call b d) (proc b (call (proc a (call a b)) (call b d)))),
```

whose Stoy diagram,



shows that variable capture violates the integrity of the free variable `b` within the second occurrence of `(call b d)`.

Figure 6.13 presents a method of substitution that avoids variable capture. Substitution is defined by structural induction on the expression substituted into. However, there is sometimes more than one clause per expression type because some expression types have subcases that depend on interactions between the variable I_{subst} being replaced and variables within the expression substituted into. For example, $[E/I_{subst}]I_{exp}$ is E if I_{subst} and I_{exp} are syntactically identical, but is the original expression I_{exp} if I_{subst} and I_{exp} are not the same. These different subcases are expressed in Figure 6.13 by implicit pattern matching or explicit restrictions.

As seen in Figure 6.13, most of the rules straightforwardly distribute the substitution over the subexpressions of an expression. The tricky case is substituting into a variable declaration construct (`proc` or `rec`). For example, consider the case for `proc`:

$$[E_{new}/I_{subst}]\texttt{(proc } I_{bound} \ E_{body}\texttt{)},$$

In the case where I_{subst} and I_{bound} are the same, no substitutions can be permitted inside the abstraction because I_{bound} declares a variable that is distinct from the one named by I_{subst}. Without this restriction, we could derive results like

$$\texttt{[b/a](proc a (call a b))} = \texttt{(proc b (call b b))}$$

in which external variable capture invalidates the purported substitution.

When I_{subst} and I_{bound} are distinct, the crucial situation to handle is where I_{subst} appears free in E_{body} (so a substitution will definitely take place) *and*

$$
\begin{array}{rcl}
[E_{new}/I_{sub}]L & = & L \\
{[E_{new}/I_{sub}]I_{sub}} & = & E_{new} \\
{[E_{new}/I_{sub}]I_{expr}} & = & I_{expr}, \text{ where } I_{sub} \neq I_{expr} \\
{[E_{new}/I_{sub}](\texttt{primop } O} & = & (\texttt{primop } O\ [E_{new}/I_{sub}]E_1\ \ldots\ [E_{new}/I_{sub}]E_n) \\
\quad E_1\ \ldots\ E_n) & & \\
{[E_{new}/I_{sub}](\texttt{proc } I_{sub}\ E_{body})} & = & (\texttt{proc } I_{sub}\ E_{body}) \\
{[E_{new}/I_{sub}](\texttt{proc } I\ E_{body})} & = & (\texttt{proc } I_{fresh}\ [E_{new}/I_{sub}]([I_{fresh}/I]E_{body}))\ , \\
 & & \quad \text{where } I_{sub} \neq I \text{ and} \\
 & & \quad I_{fresh} \notin \{I_{sub}\} \cup FreeIds[\![E_{new}]\!] \\
 & & \qquad\qquad \cup FreeIds[\![E_{body}]\!] \\
{[E_{new}/I_{sub}](\texttt{call } E_{rator}\ E_{rand})} & = & (\texttt{call } [E_{new}/I_{sub}]E_{rator}\ [E_{new}/I_{sub}]E_{rand}) \\
{[E_{new}/I_{sub}](\texttt{if } E_1\ E_2\ E_3)} & = & (\texttt{if } [E_{new}/I_{sub}]E_1 \\
 & & \quad [E_{new}/I_{sub}]E_2 \\
 & & \quad [E_{new}/I_{sub}]E_3) \\
{[E_{new}/I_{sub}](\texttt{pair } E_1\ E_2)} & = & (\texttt{pair } [E_{new}/I_{sub}]E_1\ [E_{new}/I_{sub}]E_2) \\
{[E_{new}/I_{sub}](\texttt{rec } I_{sub}\ E_{body})} & = & (\texttt{rec } I_{sub}\ E_{body}) \\
{[E_{new}/I_{sub}](\texttt{rec } I\ E_{body})} & = & (\texttt{rec } I_{fresh}\ [E_{new}/I_{sub}]([I_{fresh}/I]E_{body}))\ , \\
 & & \quad \text{where } I_{sub} \neq I \text{ and} \\
 & & \quad I_{fresh} \notin \{I_{sub}\} \cup FreeIds[\![E_{new}]\!] \\
 & & \qquad\qquad \cup FreeIds[\![E_{body}]\!]
\end{array}
$$

Figure 6.13: The definition of substitution.

E_{new} contains a free reference to I_{bound}. This reference will be captured by the bound variable of the abstraction unless we're careful. A simple example of this situation is:

`[b/a](proc b (call b a))`.

Here, the substituted expression `b` contains (in fact, is) a free reference to a variable whose name happens to be the same as the name of the variable bound by the abstraction. A naïve substitution would yield `(proc b (call b b))`, in which the outer variable named `b` has been accidentally captured by the inner variable of the same name. To prevent this internal variable capture, it is necessary to first consistently rename the bound variable of the abstraction with an identifier that is not the same as I_{subst} and is free neither in E_{new} nor in E_{body}. After this renaming, substitution can be performed on E_{body} without threat of variable capture. In our example, the bound variable `b` can be renamed to `c`, say, yielding the alpha-equivalent abstraction `(proc c (call c a))`. Then substitution can be performed on the body to yield the correct expression

`(proc c [b/a](call c a))` = `(proc c (call c b))`.

In the case where $I_{subst} \neq I_{bound}$, it is always correct to perform the described renaming of the bound variable of the abstraction, but it is not always necessary. If I_{subst} is not free in E_{body}, renaming is not required because no substitution will be performed inside the abstraction anyway. And if I_{bound} doesn't appear in E_{new}, no internal variable capture can arise, and it is safe to directly substitute into the body of the abstraction without a renaming step.

In the rule for substituting into an abstraction, it is necessary to choose an identifier that is not the same as I_{subst} and is free neither in E_{new} nor in E_{body}. The notion of choosing an identifier that satisfies certain properties often arises when manipulating syntactic expressions in which variables are represented by identifiers. Such an identifier is said to be **fresh**. When describing a syntactic manipulation, it is always necessary to specify any constraints involved in choosing the fresh identifiers.

Keep in mind that all the complexity for renaming and substitution arises from dealing with linear (in this case, textual) representations for declaration/reference relationships that are not linear or even tree-like. If FLK expressions were represented instead as DAGs or Stoy diagrams, renaming would be unnecessary and substitution would be straightforward.

▷ **Exercise 6.11** Use the definition of substitution in Figure 6.13 to determine the results of the following substitutions. Assume that fresh identifiers are taken from the list $\mathtt{v}_1$, $\mathtt{v}_2$, $\mathtt{v}_3$, ..., and that the first identifier from the list that satisfies the given constraint is chosen as the fresh identifier.

a. `[(call (call b c) d)/a](proc a (proc b (call (call c b) a)))`

b. `[(call (call b c) d)/b](proc a (proc b (call (call c b) a)))`

c. [(call (call b c) d)/c](proc a (proc b (call (call c b) a)))

d. [(call (call b c) d)/d](proc a (proc b (call (call c b) a)))

e. [(call (call b c) d)/b](proc a (proc b (call c a))) ◁

▷ **Exercise 6.12** Consider the case for substituting into `proc` abstractions,

$$[E_{new}/I_{subst}](\texttt{proc}\ \ I_{bound}\ \ E_{body}),$$

where $I_{subst} \neq I_{bound}$. Here I_{bound} is consistently renamed to be a variable I_{fresh} that is not free in either E_{new} or E_{body} and is not equal to I_{subst}.

a. Provide an example of an incorrect substitution that would be permitted if the restriction $I_{fresh} \notin \mathit{FreeIds}[\![E_{new}]\!]$ were lifted.

b. Provide an example of an incorrect substitution that would be permitted if the restriction $I_{fresh} \notin \mathit{FreeIds}[\![E_{body}]\!]$ were lifted.

c. Provide an example of an incorrect substitution that would be permitted if the restriction $I_{fresh} \neq I_{subst}$ were lifted.

d. Would it be possible to consistently rename the free variables of E_{new} (within both E_{new} and E_{body}) instead of renaming I_{bound}? Explain your answer, using examples where appropriate. ◁

▷ **Exercise 6.13** Assuming that I_1 and I_2 are distinct, and that $I_2 \notin \mathit{FreeIds}[\![E_1]\!]$, prove the following useful equivalence:

$$[E_1/I_1]([E_2/I_2]E_3) = [([E_1/I_1]E_2)/I_2]([E_1/I_1]E_3)$$

(Hint: Do the proof by induction on the height of E_3.) ◁

▷ **Exercise 6.14** The notion of **simultaneous substitution** is an extension to the substitution function we have seen. A simultaneous substitution, $[E_1 \ldots E_n/I_1 \ldots I_n]$, is a function of a single expression that performs the substitutions $[E_1/I_1] \ldots [E_n/I_n]$ in parallel on that expression. It differs from a sequence of substitutions in that an I_i appearing in one of the E_j is never substituted for. For example, simultaneous substitution of I_2 for I_1 and I_1 for I_2 in the expression `(call` I_1 I_2`)` swaps the two identifiers:

$$[I_2, I_1/I_1, I_2](\texttt{call}\ \ I_1\ \ I_2) = (\texttt{call}\ \ I_2\ \ I_1)$$

whereas neither ordering of two single substitutions has this behavior:

$$[I_2/I_1]([I_1/I_2](\texttt{call}\ \ I_1\ \ I_2)) = (\texttt{call}\ \ I_2\ \ I_2)$$

$$[I_1/I_2]([I_2/I_1](\texttt{call}\ \ I_1\ \ I_2)) = (\texttt{call}\ \ I_1\ \ I_1)$$

Write a formal definition of simultaneous substitution for FLK. ◁

▷ **Exercise 6.15** Suppose that FL is extended with the following constructs for manipulating tuples of elements:

`(tuple` E^*`)`:	Non-strict constructor of a tuple with any number of elements.
`(tuple-ref` E i`)`:	Suppose i is a positive integer i and E is a tuple t. Return the ith element of t (assume 1-based indexing).
`(tuple?` E`)`:	Predicate determining if E is a tuple.
`(tuple-length` E`)`:	Returns the number of elements in the tuple.

Tuples provide an alternate way to desugar multi-abstractions and multi-applications. Multi-applications can package arguments into a tuple that is unpackaged by a multi-abstraction.

a. Provide tuple-based desugarings for multi-abstractions and multi-applications. You may find substitution helpful. Explain any design choices that you make.

b. Discuss the advantages and disadvantages of the tuple-based desugaring versus the desugaring based on currying. ◁

6.4 An Operational Semantics for FLK

6.4.1 An SOS for FLK

Figure 6.14 presents an SOS for FLK. In addition to the semantic domains of FLK, the SOS uses the following domains:

- The ValueExp domain is a subset of Exp_{FLK} consisting of expressions that model the values manipulated by FLK programs. The notations {`(symbol` I`)`}, {`(proc` I E`)`}, {`(pair` E_1 E_2`)`}, and {`(error` I_{msg}`)`} indicate the set of all expressions that match the given pattern. The value expressions include all the literals, as well as abstractions (representing procedural values), pairings (representing pair values), and error expressions.

- Each input to an FLK program is an s-expression value from the SExpVal domain. This is a subset of ValueExp that excludes all `proc` and `error` forms.

- The Answer domain models final answers in the execution of FLK programs. It is similar to ValueExp except that it replaces all `proc` expressions by the procedure value token `procval` and replaces all `pair` expressions

Domains

$$
\begin{array}{rcll}
V & \in & \text{ValueExp} & = \{\texttt{\#u}\} \cup \text{Boollit} \cup \text{Intlit} \cup \{\texttt{(symbol } I\texttt{)}\} \\
 & & & \cup\{\texttt{(proc } I\ E\texttt{)}\} \cup\{\texttt{(pair } E_1\ E_2\texttt{)}\} \cup\{\texttt{(error } I_{msg}\texttt{)}\} \\
SV & \in & \text{SExpVal} & = \{\texttt{\#u}\} \cup \text{Boollit} \cup \text{Intlit} \cup \{\texttt{(symbol } I\texttt{)}\} \\
 & & & \{\texttt{(pair } SV_1\ SV_2\texttt{)}\} \\
I & \in & \text{Inputs} & = \text{SExpVal}^* \\
A & \in & \text{Answer} & = \{\texttt{\#u}\} \cup \text{Boollit} \cup \text{Intlit} \cup \{\texttt{(symbol } I\texttt{)}\} \\
 & & & \cup\{\texttt{procval}\} \cup\{\texttt{pairval}\} \cup\{\texttt{(error } I_{msg}\texttt{)}\}
\end{array}
$$

SOS

The FLK SOS has the form $FLKSOS = \langle \text{Exp}_{FLK}, \Rightarrow, \text{ValueExp}, IF, OF \rangle$, where:

$\Rightarrow$ is a deterministic transition relation defined in Figures 6.15 and 6.16.

$IF : \text{Program}_{FLK} \times \text{Inputs} \to \text{Exp}_{FLK}$
$= \lambda\langle \texttt{(flk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}, [SV_1, \ldots, SV_k]\rangle\ .$
if n = k **then** $([SV_i/A_i]_{i=1}^{n})E_{body}$
else `(error wrong-number-of-args)` **fi**

$OF : \text{ValueExp} \to \text{Answer}$
$= \lambda V\ .$ **matching** V
$\triangleright$ `(proc` I E`)` $\Vert$ `procval`
$\triangleright$ `(pair` E_1 E_2`)` $\Vert$ `pairval`
$\triangleright$ **else** V **endmatching**

Figure 6.14: An SOS for FLK.

by the pair value token `pairval`. These tokens distinguish the types of procedure and pair values, but the structure of these values is not observable.

The configuration space for the FLK SOS consists of FLK expressions. The input function *IF* maps an FLK program and a sequence of s-expression argument values to an initial configuration by substituting the arguments for the formal parameter names in the body of the program. The final configurations of the SOS are modeled by the ValueExp domain. The output function *OF* erases the details of all procedure and pair values.

The SOS rewrite relation $\Rightarrow$ is defined by the rewrite rules in Figures 6.15 and 6.16. Applications are handled by the [*call-apply*] and [*call-operator*] rules. The [*call-apply*] rule makes use of the FLK substitution operator to evaluate the application of an abstraction. The [*call-operator*] progress rule permits rewrites on the operator. No rewrites are performed on the operand so these rules are non-strict, like `if` and unlike `primop`.

$$\texttt{(call (proc } I\ E_1\texttt{)}\ E_2\texttt{)} \Rightarrow [E_2/I]E_1 \qquad [\textit{call-apply}]$$

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(call } E_1\ E_2\texttt{)} \Rightarrow \texttt{(call } E_1{}'\ E_2\texttt{)}} \qquad [\textit{call-operator}]$$

$$\texttt{(if \#t } E_1\ E_2\texttt{)} \Rightarrow E_1 \qquad [\textit{if-true}]$$

$$\texttt{(if \#f } E_1\ E_2\texttt{)} \Rightarrow E_2 \qquad [\textit{if-false}]$$

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(if } E_1\ E_2\ E_3\texttt{)} \Rightarrow \texttt{(if } E_1{}'\ E_2\ E_3\texttt{)}} \qquad [\textit{if-test}]$$

$$\texttt{(rec } I\ E\texttt{)} \Rightarrow [\texttt{(rec } I\ E\texttt{)}/I]E \qquad [\textit{rec}]$$

$$\frac{E \Rightarrow E'}{\texttt{(primop } O\ E\texttt{)} \Rightarrow \texttt{(primop } O\ E'\texttt{)}} \qquad [\textit{unary-arg}]$$

$$\frac{E_1 \Rightarrow E_1{}'}{\texttt{(primop } O\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(primop } O\ E_1{}'\ E_2\texttt{)}} \qquad [\textit{binary-arg-1}]$$

$$\frac{E_2 \Rightarrow E_2{}'}{\texttt{(primop } O\ V\ E_2\texttt{)} \Rightarrow \texttt{(primop } O\ V\ E_2{}'\texttt{)}} \qquad [\textit{binary-arg-2}]$$

Figure 6.15: FLK rewrite rules, part 1.

Strict languages would include progress rules for operands of procedure calls (as FLK does for primitives), and these rules would reflect constraints on evaluation order. However, even strict languages have non-strict conditionals to avoid errors (e.g., division by zero) and infinite loops (the base case of a recursion, such as the factorial of 0).

The semantics of recursion is especially simple in the SOS framework. It is obtained by simply "unwinding" the recursion equation one level. Programmers often follow the same approach when trying to hand-simulate the behavior of recursive procedures.

The three progress rules [*unary-arg*], [*binary-arg-1*], and [*binary-arg-2*] suffice for forcing the evaluation of arguments in a primitive application. The metavariable V in rule [*binary-arg-2*] is used to express a constraint that the first operand must be a value; thus the first argument must be fully evaluated before the second argument is evaluated. These three rules are actually instantiations of a single general rule to evaluate any number of arguments in left-to-right order:

$$\frac{E_i \Rightarrow E_i\,'}{\begin{array}{l}\texttt{(primop}\ O\ V_1\ \ldots V_{i-1}\ E_i\ \ldots E_n\texttt{)} \\ \Rightarrow\texttt{(primop}\ O\ V_1\ \ldots V_{i-1}\ E_i\,'\ \ldots E_n\texttt{)}\end{array}} \qquad [\textit{prim-arg}]$$

where n can be any nonnegative integer (including 0) and i ranges between 0 and n. The notation is intended to indicate that the first $i-1$ arguments have all been fully evaluated, and the ith expression is in the process of evaluation.

A sampling of the remaining primitive operator rules are given in Figure 6.16. These rules define the behavior of each primitive operator. The *calculate* function used in the [+] rule serves the same purpose as it did in the PostFix SOS.

Like the PostFix SOS, the FLK SOS models most errors with stuck states. If the final configuration happens to be an `error` form, then this will be returned as the outcome of the program. But if a configuration is stuck because it contains a problematic subexpression such as `(primop + 1 #t)` or an `error` form, the outcome of the program will be `stuck`. See Exercise 6.21 for an alternative approach to handle errors in FLK.

6.4.2 Example

Figure 6.17 illustrates a sample proof-structured evaluation of the expression

```
(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
```

- `not`:

  `(primop not? #f)` $\Rightarrow$ `#t` [*not-1*]

  `(primop not? #t)` $\Rightarrow$ `#f` [*not-2*]

- `left` and `right`:

  `(primop left (pair` E_1 E_2`))` $\Rightarrow E_1$ [*left*]

  `(primop right (pair` E_1 E_2`))` $\Rightarrow E_2$ [*right*]

- `integer?` (other predicates are defined similarly):

  `(primop integer?` N`)` $\Rightarrow$ `#t` [*integer?-integer*]

  `(primop integer? #u)` $\Rightarrow$ `#f` [*integer?-unit*]

  `(primop integer?` B`)` $\Rightarrow$ `#f` [*integer?-boolean*]

  `(primop integer? (symbol` I`))` $\Rightarrow$ `#f` [*integer?-symbol*]

  `(primop integer? (proc` I E`))` $\Rightarrow$ `#f` [*integer?-abstraction*]

  `(primop integer? (pair` E_1 E_2`))` $\Rightarrow$ `#f` [*integer?-pair*]

- `and?` (`or?` is defined similarly):

  `(primop and? #t #t)` $\Rightarrow$ `#t` [*and-true-true*]

  `(primop and? #t #f)` $\Rightarrow$ `#f` [*and-true-false*]

  `(primop and? #f #t)` $\Rightarrow$ `#f` [*and-false-true*]

  `(primop and? #f #f)` $\Rightarrow$ `#f` [*and-false-false*]

- `+` (other binary operators are similar, except for `/` and `rem`):

  `(primop +` N_1 N_2`)` $\Rightarrow$ (*calculate* `+` N_2 N_1) [*+*]

- `/` (`rem` is similar):

  `(primop /` N_1 N_2`)` $\Rightarrow$ (*calculate* `/` N_2 N_1), where $N_2 \neq$ `0` [*/*]

Figure 6.16: FLK rewrite rules, part 2.

based on the above rewriting rules. Each rewriting step is annotated with a justification that explains how the step follows from previous steps and a rewrite rule.

A more condensed form of the evaluation in Figure 6.17 treats as a single rewrite any axiom rewrite in conjunction with any number of rewrites implied by progress rules. This gives rise to a linear sequence of rewrites, where the rewrite arrow can be subscripted with the name of the axiom applied. The example from the figure then becomes:

```
(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
⇒[call-apply] (call (call (proc a (proc b (primop - b a)))
                          (primop + 4 1))
                    3)
⇒[call-apply] (call (proc b (primop - b (primop + 4 1)))
                    3)
⇒[call-apply] (primop - 3 (primop + 4 1))
⇒[+] (primop - 3 5)
⇒[−] -2
```

▷ **Exercise 6.16** Use the rewrite rules to show the evaluation of the following expressions:

a. `(primop left (pair 1 (primop not? 3)))`

b. `(primop left (primop right (primop right`
`(rec p (pair 1 (pair 2 p))))))`

The first expression illustrates the non-strictness of `pair` while the second illustrates the unwinding nature of `rec`. ◁

▷ **Exercise 6.17** Since FLK is non-strict, it is not necessary for `if` to be a distinguished construct. Instead, `if` could be a unary primitive operator that returns a (curried) binary function. That is, instead of being written `(if` E_1 E_2 E_3`)`, conditionals could be expressed as

`(call (call (primop if` E_1`)` E_2`)` E_3`)`

Give the rewrite rules for `if` as a unary primitive operator. ◁

▷ **Exercise 6.18** Functional computation in a dynamically typed language can be viewed as a bureaucracy where envelopes (values containing a type and other information) are shuffled around by the interpreting agent that performs the computation.[5] In many steps of the computation, envelopes are simply moved around without being

[5] Phil Agre introduced us to this point of view.

```
(call (proc f (call f (primop + 4 1)))
      (proc a (proc b (primop - b a))))
⇒ (call (proc a (proc b (primop - b a)))                    1: call-apply
        (primop + 4 1))

(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
⇒ (call (call (proc a (proc b (primop - b a)))      2: 1 & call-operator
              (primop + 4 1))
        3)

(call (proc a (proc b (primop - b a)))
      (primop + 4 1))
⇒ (proc b (primop - b (primop + 4 1)))                      3: call-apply

(call (call (proc a (proc b (primop - b a)))
            (primop + 4 1))
      3)
⇒ (call (proc b (primop - b (primop + 4 1)))        4: 3 & call-operator
        3)

(call (proc b (primop - b (primop + 4 1)))
      3)
⇒ (primop - 3 (primop + 4 1))                               5: call-apply

(primop + 4 1) ⇒ 5                                                  6: +

(primop - 3 (primop + 4 1)) ⇒ (primop - 3 5)          7: binary-arg-2

(primop - 3 5) ⇒ -2                                                 8: +

(call (call (proc f (call f (primop + 4 1)))
            (proc a (proc b (primop - b a))))
      3)
⇒* -2                                                 9: 2 & 4 & 5 & 7 & 8
```

Figure 6.17: Example evaluation of an FLK expression.

opened. In the formation of a non-strict pair, for instance, two envelopes are simply stuffed into a larger envelope without ever having their contents examined. During other stages — a primitive addition, for instance — the contents (type and content information) of envelopes must definitely be examined.

With this perspective in mind, for each FLK expression describe when the contents of envelopes must be examined. In other words, which contexts demand the value of an expression? ◁

▷ **Exercise 6.19** Suppose we want to extend FL with a `least` construct. Given a numeric predicate, `least` returns the least non-negative integer that satisfies the predicate. For example,

`(least (proc (x) (= x (* x x))))` $\xrightarrow[FL]{}$ *0*
`(least (proc (a) (> (* a a) 10)))` $\xrightarrow[FL]{}$ *4*
`(least (proc (x) (< x 0)))` $\xrightarrow[FL]{}$ ∞*-loop* {*Looks, but no solution*}
`(least (proc (x) x))` $\xrightarrow[FL]{}$ *error:Non-bool-in-if-test*

a. Must the argument to `least` always be an abstraction? If so, explain why; if not, give a counterexample.

b. One way to add `least` is to extend the syntax of FLK to include `(least` *E*`)` as a new expression type. Extend the operational semantics of FLK to handle the `least` expression. Keep in mind that a SOS has five parts; make the appropriate modifications to each of the parts.

Hint: In addition to adding `(least` *E*`)` to the configuration space, it is also desirable to add a configuration of the form `(*least*` *E N*`)`. Configurations like `*least*` that are not valid as expressions in the language are often useful for representing intermediate states of computations.

c. Alternately, `least` could be written as a user-defined procedure that is standardly available in the body of a `program`. Show how to implement `least` with this approach. ◁

▷ **Exercise 6.20** In FLK, `pair` is a primitive construct built into the syntax. In a non-strict language, though, there is no need for `pair` to be primitive.

a. One option is to include `pair` as a primitive primop operator. Implement this change by modifying the operational semantics of FLK.

b. Is it possible to define `pair` as a user-defined procedure? How would you implement `left`, `right`, and `pair?` ? ◁

▷ **Exercise 6.21** Like the PostFix SOS, the FL SOS uses stuck states to model errors. For example, all of the following stuck states correspond to error situations:

```
a                              ; Unbound variable
(primop / 1 0)                 ; Division by 0
(primop + 1 #t)                ; Inappropriate argument type
(primop + 1 2 3)               ; Inappropriate number of arguments
(call 1 2)                     ; Attempt to apply a non-procedure
(if (symbol nonbool) 2 3)      ; Non-boolean test in an IF.
```

Rather than using stuck states to model errors, we can use the fact that ValueExp includes the form `(error` I_{msg}`)` to explicit represent and propagate errors. For this approach, the rewrite rules need to (1) convert stuck expressions to an appropriate `error` form and (2) propagate `error` forms so that they eventually become final configurations. For example, we could have the rule

$$\texttt{(call } N \ E\texttt{)} \Rightarrow \texttt{(error non-procedural-rator)} \qquad [\textit{integer-operator-error}]$$

to express the fact that it is an error to use an integer in the operator position of an application.

Make all necessary modifications and additions to the FLK rewrite rules in order to handle the explicit introduction and propagation of `error` forms. Make sure that errors propagate appropriately; e.g.,

```
(primop + 1 (primop / 1 0))
```

should rewrite to an error because it has a subexpression that rewrites to an error. ◁

▷ **Exercise 6.22** After carefully studying the SOS for FLK, Paula Morwicz proclaims that it is safe to use a naive substitution strategy (i.e., one that does not rename bound variables) in the [*call-apply*] and [*rec*] rules *as long as* the original expression being evaluated does not contain any unbound variables (i.e., free identifiers).

a. Show that Paula is right. That is, show that the name capture problems addressed by the definition of substitution in Figure 6.13 cannot occur during the evaluation of an FLK expression that has no unbound variables.

b. Give an example of an FLK expression containing an unbound variable that evaluates to the wrong answer if the the naive substitution strategy is used.

c. Suppose that every FLK expression were alpha-renamed so that all variables had distinct identifiers and no bound variable used the same identifier as any unbound variable. Under these conditions, is it always safe to use the naive substitution strategy? If so, explain; if not, give a counter-example. ◁

▷ **Exercise 6.23** After reading up on the the lambda calculus, Sam Antix decides to experiment with some new rewrite rules for the FL SOS.

a. The first rule he tries is the so-called **eta rule**:

$$\begin{array}{l}\texttt{(proc } I \texttt{ (}E\ I\texttt{))} \Rightarrow E, \\ \text{where } \ I \notin \mathit{FreeIds}[\![E]\!]\end{array} \qquad [\textit{eta}]$$

Although this rule is reasonable in the lambda calculus, it greatly changes the semantics of FLK. Demonstrate this fact by showing a FLK expression that can evaluate to two different values along two different transition paths.

b. The eta rule can be made safe by restricting the form of E. Describe such a restriction, and explain why the rule is safe.

c. After getting rid of the [*eta*] rule, Sam experiments with a rule that allows rewrites within the body of an abstraction:

$$\frac{E \Rightarrow E'}{(\texttt{proc}\ I\ E) \Rightarrow (\texttt{proc}\ I\ E')} \qquad [\textit{proc-body}]$$

How does the addition of this rule change the semantics of FLK? For example, does it make it possible for an expression to rewrite to two different values via two different transition paths? Does it enable new kinds of transition paths? ◁

6.5 A Denotational Definition for FLK

In this section, we develop a denotational semantics for FLK. A complete denotational semantics for FLK appears in Figures 6.18–6.22. The semantic algebras for this semantics appear in Figure 6.18, and Figures 6.19 and 6.20 define auxiliary functions and values. These definitions provide the landscape that serves as the backdrop for our future denotational definitions, as well as for the valuation functions in Figures 6.21 and 6.22.

It is always best to begin a study of a denotational semantics with a careful look at the semantic algebras. Here is what we can see by looking at the FLK semantic algebras in Figure 6.18.

The values that can be expressed by an FLK expression are modeled by the *Expressible* domain, which is a lifted sum of *Value* and *Error*. Errors, like symbols, are modeled as identifiers.[6] *Value* contains unit, boolean, integer, and symbol values, as well as pair and procedure values, which are defined recursively in terms of *Expressible*. The bottom element of the *Expressible* domain represents a non-terminating computation in FLK.

Whereas the SOS for FLK used substitution to model naming, the denotational semantics uses **environments** as a kind of virtual substitution. When a value is bound to an identifier, that binding is stored in the environment used to evaluate expressions within the scope of that binding. Identifiers that represent

[6] We are being a little loose here. Program identifiers often exclude language key words, like `let`. Such restrictions should not be applied to program data or errors.

c	$\in$	*Computation*	$=$	*Expressible*
δ	$\in$	*Denotable*	$=$	*Computation*
p	$\in$	*Procedure*	$=$	$Denotable \rightarrow Computation$
β	$\in$	*Binding*	$=$	$(Denotable + Unbound)_{\perp}$
e	$\in$	*Environment*	$=$	Identifier $\rightarrow$ *Binding*
		Unbound	$=$	$\{unbound\}$
x	$\in$	*Expressible*	$=$	$(Value + Error)_{\perp}$
v	$\in$	*Value*	$=$	$Unit + Bool + Int + Sym + Pair + Procedure$
		Unit	$=$	$\{unit\}$
i	$\in$	*Int*	$=$	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
b	$\in$	*Bool*	$=$	$\{true, false\}$
y	$\in$	*Sym*	$=$	Identifier
a	$\in$	*Pair*	$=$	$Computation \times Computation$
		Error	$=$	Identifier

Figure 6.18: The semantic algebras for FLK.

variable references are looked up in the current environment. Environments map identifiers to bindings, where *Binding* is a lifted sum of denotable values and the trivial domain *Unbound*. The trivial element acts as an "unbound marker" that indicates that an identifier is not bound in an environment.

The environment functions (Figure 6.19) have been updated to be consistent with the *Binding* domain. In particular, there is now a distinction between *extend*, which associates a name with a denotable in an environment, and *bind*, which associates a name with a binding in an environment. The figure introduces shorthand notation for these functions that will be used in future valuation clauses.

There is no a priori reason why the class of entities that can be named in an environment has to be the same as that denoted by arbitrary expressions. For this reason, there is a separate semantic domain, *Denotable*, for the set of values that can be associated with names in environments. There are many possible relationships between *Denotable* and *Expressible*:

- *Denotable* may be the same as *Expressible*. This is the case in FL.
- *Denotable* may be a superset of *Expressible* — some entities may be named but not computed. For example, languages in which procedures are not first-class typically have ways to name procedures (usually via a declaration) even though procedures cannot be values of expressions.
- *Denotable* may be a subset of *Expressible* — some entities may be computed, but not named. For example, in certain languages variables cannot

Environment operations:

empty-env : *Environment*
$= \lambda I\,.\,(\mathit{Unbound} \mapsto \mathit{Binding}\ \mathit{unbound})$

lookup : *Environment* $\rightarrow$ Identifier $\rightarrow$ *Binding*
$= \lambda e I\,.\,(e\ I)$

bind : *Environment* $\rightarrow$ Identifier $\rightarrow$ *Binding* $\rightarrow$ *Environment*
$= \lambda e I_1 \beta\,.\,\lambda I_2\,.$ **if** (*same-identifier?* $I_1\ I_2$) **then** β **else** (*lookup* $e\ I_2$) **fi**
(*bind* $e\ I\ \beta$) will be abbreviated $[I :: \beta]e$; this notation associates to the right:
$[I_2 :: \beta_2][I_1 :: \beta_1]e = [I_2 :: \beta_2]([I_1 :: \beta_1]e)$

extend : *Environment* $\rightarrow$ Identifier $\rightarrow$ *Denotable* $\rightarrow$ *Environment*
$= \lambda e I \delta\,.\,(\mathit{bind}\ e\ I\ (\mathit{Denotable} \mapsto \mathit{Binding}\ \delta))$
(*extend* $e\ I\ \delta$) will be abbreviated $[I : \delta]e$; this notation associates to the right:
$[I_2 : \delta_2][I_1 : \delta_1]e = [I_2 : \delta_2]([I_1 : \delta_1]e)$

Figure 6.19: Auxiliary functions and values for FLK, Part II.

name values that represent errors and nontermination. We shall study this example in detail when we discuss call-by-value semantics in Chapter 7.

- The relationship between *Denotable* and *Expressible* may be more complex. Consider a language in which procedures are denotable but not expressible, and errors are expressible but not denotable. (FORTRAN is in this category.)

Thus, the definitions of *Denotable* and *Expressible* in the denotational semantics of a given language contain some important information about high-level features of the language. The availability of this kind of information is the reason why, when reading a denotational semantics, it is advisable to first carefully study domain equations and function signatures before delving into the details of the valuation functions.

The meaning of an expression with respect to an environment depends on the formulation of the meaning function used. To provide a level of abstraction, we will define a new domain called *Computation*. The *Computation* domain names the domain of meanings that we can get from evaluating an expression in an environment:

$\mathcal{E}$: Exp $\rightarrow$ *Environment* $\rightarrow$ *Computation*

The *Computation* domain, with the helper functions in Figure 6.20 (described more below), allows us to factor out some complex details and have compact clauses in our valuation functions. In FLK, the benefit is largely that we can factor out much of the error checking. When we extend FL, e.g., in order to add state in Chapter 8, *Computation* will become more complex, but it will allow the valuation functions to remain relatively simple.

The *Computation*, *Denotable*, and *Value* domains all serve as knobs that can be tweaked to specify different languages. The *Procedure* domain's argument value must be denotable (otherwise the argument could not be named by a formal parameter).

We assume that the *Computation* domain comes equipped with a set of helper functions shown in Figure 6.20. *val-to-comp* treats a value as computation, while *err-to-comp* treats an error as one. *with-value* is a generalized version of the various functions we have already seen with this name. It unpackages a computation into a value (if possible) and applies to this value a function that returns another computation. In the case where the computation cannot be coerced to a value, it is passed along unchanged. The other *with-* functions (which can be written in terms of *with-value*), are similar, except that they may also generate new error computations rather than just passing along old ones.

The valuation functions of Figures 6.21 and 6.22 are relatively compact, thanks in large part to the *Computation* abstraction and the associated helper functions. However, semantics written in this style can take some time to get used to. It is helpful to keep in mind the signatures of all functions, as well as the purposes of the various auxiliary functions. To see how much more complicated the valuation clauses would be, compare the one-line `if` clause of Figure 6.21 with:

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(if } E_1 \ E_2 \ E_3\texttt{)}]\!] = \\
\quad \lambda e\,.\ \textbf{matching}\ (\mathcal{E}[\![E_1]\!]\ e) \\
\qquad \triangleright\,(\mathit{Value} \mapsto \mathit{Computation}\ v)\ [\!]\ \textbf{matching}\ v \\
\qquad\qquad\qquad\qquad\qquad\qquad \triangleright\,(\mathit{Bool} \mapsto \mathit{Value}\ b)\ [\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ b\ \ \textbf{then}\ (\mathcal{E}[\![E_2]\!]\ e)\ \ \textbf{else}\ (\mathcal{E}[\![E_3]\!]\ e)\ \ \textbf{fi} \\
\qquad\qquad\qquad\qquad\qquad\qquad \triangleright\,\textbf{else}\ (\mathit{err\text{-}to\text{-}comp}\ \texttt{non-bool-in-if-test}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endmatching} \\
\qquad \triangleright\,\textbf{else}\ (\mathcal{E}[\![E_1]\!]\ e) \\
\qquad \textbf{endmatching}
\end{array}
$$

This sort of error checking would be repeated throughout the valuation clauses.

▷ **Exercise 6.24** Recall that the integer division and remainder operators (`/` and `rem`) are different than other binary operators because they are ill-defined when the second argument is *0*. Write the valuation clause for $\mathcal{P}[\![/]\!]$. ◁

Usual operations on *Bool*: *not*, *and*, *or*
Usual operations on *Int*: $+$, -, $*$, $/$, ...
Equality operation on Identifier: *same-identifier?*

val-to-comp : *Value* $\rightarrow$ *Computation* $=$ Value $\mapsto$ Computation
err-to-comp : *Error* $\rightarrow$ *Computation* $=$ Error $\mapsto$ Computation
den-to-comp : *Denotable* $\rightarrow$ *Computation* $= \lambda\delta\,.\,\delta$
error-comp : *Computation* $=$ (*err-to-comp* `error`)

with-value : *Computation* $\rightarrow$ (*Value* $\rightarrow$ *Computation*) $\rightarrow$ *Computation*
$= \lambda cf\,.$ **matching** c
$\triangleright$ (*Value* $\mapsto$ *Computation* v) $[\![]$ $(f\ v)$
$\triangleright$ **else** c
endmatching

with-values : *Computation*$*$ $\rightarrow$ (*Value*$*$ $\rightarrow$ *Computation*) $\rightarrow$ *Computation*
$= \lambda c{*}f\,.$ **matching** $c*$
$\triangleright\, []_{Computation}$ $[\![]$ $(f\ []_{Value})$
$\triangleright\, c_{fst}\ .\ c_{rest}{*}$ $[\![]$ (*with-value* c_{fst}
$(\lambda v_{fst}\,.$ (*with-values* $c_{rest}{*}$ $(\lambda v_{rest}{*}\,.$ $(f\ (v_{fst}\ .\ v_{rest}{*})))))$
endmatching

with-boolean-val : *Value* $\rightarrow$ (*Bool* $\rightarrow$ *Computation*) $\rightarrow$ *Computation*
$= \lambda v\,.$ **matching** v
$\triangleright$ (*Bool* $\mapsto$ *Value* b) $[\![]$ $(f\ b)$
$\triangleright$ **else** (*err-to-comp* `not-a-boolean`)
endmatching
Similar for *with-unit-val*, *with-integer-val*, *with-symbol-val*, *with-pair-val*.

with-boolean-comp : *Computation* $\rightarrow$ (*Bool* $\rightarrow$ *Computation*) $\rightarrow$ *Computation*
$= \lambda cf\,.$ (*with-value* c $(\lambda v\,.$ (*with-boolean-val* v f)))
Similar for *with-procedure-comp*.

with-denotable : *Binding* $\rightarrow$ (*Denotable* $\rightarrow$ *Computation*) $\rightarrow$ *Computation*
$= \lambda\beta f\ .$ **matching** β
$\triangleright$ (*Denotable* $\mapsto$ *Binding* δ) $[\![]$ $(f\ \delta)$
$\triangleright$ (*Unbound* $\mapsto$ *Binding* *Unbound*) $[\![]$ (*err-to-comp* `unbound-var`)
endmatching

Figure 6.20: Auxiliary functions and values for FLK, Part I.

$\mathcal{E}$: Exp $\to$ *Environment* $\to$ *Computation*
$\mathcal{E}^*$: Exp* $\to$ *Environment* $\to$ *Computation**
$\mathcal{L}$: Lit $\to$ *Value*
$\mathcal{P}$: Primop $\to$ *Value** $\to$ *Computation*
$\mathcal{B}$: Boollit $\to$ *Bool*
$\mathcal{N}$: Intlit $\to$ *Int*

$\mathcal{E}[\![L]\!] = \lambda e\,.\,(\textit{val-to-comp}\ \mathcal{L}[\![L]\!])$

$\mathcal{E}[\![I]\!] = \lambda e\,.\ (\textit{with-denotable}\ (\textit{lookup}\ e\ I)\ \lambda\delta\,.\,(\textit{den-to-comp}\ \delta))$

$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] =$
$\quad \lambda e\,.\,(\textit{val-to-comp}\ (\textit{Procedure} \mapsto \textit{Value}\ \ (\lambda\delta\,.\,(\mathcal{E}[\![E]\!]\ [I:\delta]e))))$

$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,(\textit{with-procedure-comp}\ \ (\mathcal{E}[\![E_1]\!]\ e)\ \ (\lambda p\,.\,(p\ \ (\mathcal{E}[\![E_2]\!]\ e))))$

$\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] =$
$\quad \lambda e\,.\,(\textit{with-boolean-comp}\ \ (\mathcal{E}[\![E_1]\!]\ e)\ \ (\lambda b\,.\,\textbf{if}\ b\ \textbf{then}\ (\mathcal{E}[\![E_2]\!]\ e)\ \textbf{else}\ (\mathcal{E}[\![E_3]\!]\ e)\ \textbf{fi}))$

$\mathcal{E}[\![\texttt{(rec}\ I\ E\texttt{)}]\!] = \lambda e\,.\,(\textbf{fix}_{\textit{Computation}}\ \ (\lambda c\,.\,(\mathcal{E}[\![E]\!]\ [I:c]e)))$

$\mathcal{E}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\ (\textit{val-to-comp}\ (\textit{Pair} \mapsto \textit{Value}\ \langle(\mathcal{E}[\![E_1]\!]\ e),\ (\mathcal{E}[\![E_2]\!]\ e)\rangle))$

$\mathcal{E}[\![\texttt{(primop}\ O\ E^*\texttt{)}]\!] = \lambda e\,.\ (\textit{with-values}\ \ (\mathcal{E}^*[\![E^*]\!]\ e)\ \ (\lambda v^*\,.\,(\mathcal{P}[\![O]\!]\ v^*)))$

$\mathcal{E}[\![\texttt{(error}\ I\texttt{)}]\!] = \lambda e\,.\ (\textit{err-to-comp}\ I)$

$\mathcal{E}^*[\![]\!] = \lambda e\,.\ [\,]_{\textit{Computation}}$

$\mathcal{E}^*[\![E_{fst}\ .\ E_{rest}{}^*]\!] = \lambda e\,.\ (\mathcal{E}[\![E_{fst}]\!]\ e)\ .\ (\mathcal{E}^*[\![E_{rest}{}^*]\!]\ e)$

$\mathcal{L}[\![\texttt{\#u}]\!] = (\textit{Unit} \mapsto \textit{Value}\ \textit{unit})$
$\mathcal{L}[\![B]\!] = (\textit{Bool} \mapsto \textit{Value}\ \mathcal{B}[\![B]\!])$
$\mathcal{L}[\![N]\!] = (\textit{Int} \mapsto \textit{Value}\ \mathcal{N}[\![N]\!])$
$\mathcal{L}[\![\texttt{(symbol}\ I\texttt{)}]\!] = (\textit{Sym} \mapsto \textit{Value}\ I)$

$\mathcal{B}$ and $\mathcal{N}$ defined as usual.

Figure 6.21: Valuation functions for FLK, Part I

$\mathcal{P}[\![\texttt{not?}]\!] = \lambda v^* .$ **matching** v^*
  $\triangleright [v]_{Value}$ $[\!]$ $(\textit{with-boolean-val}$
      v
      $\lambda b\ .\ (\textit{val-to-comp}\ (\textit{Bool} \mapsto \textit{Value}\ (\textit{not}\ b))))$
  $\triangleright$ **else** $(\textit{err-to-comp}$ `not?-wrong-number-of-args`$)$
  endmatching

$\mathcal{P}[\![\texttt{left}]\!] = \lambda v^* .$ **matching** v^*
  $\triangleright [v]_{Value}$ $[\!]$ $(\textit{with-pair-val}\ v\ \lambda c_l c_r\ .\ c_l)$
  $\triangleright$ **else** $(\textit{err-to-comp}$ `left-wrong-number-of-args`$)$
  endmatching

Similarly for `right`

$\mathcal{P}[\![\texttt{integer?}]\!] =$
$\lambda v^* .$ **matching** v^*
  $\triangleright [v]_{Value}$ $[\!]$ **matching** v
    $\triangleright (\textit{Int} \mapsto \textit{Value}\ i)$ $[\!]$ $(\textit{val-to-comp}\ (\textit{Bool} \mapsto \textit{Value}\ \textit{true}))$
    $\triangleright$ **else** $(\textit{val-to-comp}\ (\textit{Bool} \mapsto \textit{Value}\ \textit{false}))$
    endmatching
  $\triangleright$ **else** $(\textit{err-to-comp}$ `integer?-wrong-number-of-args`$)$
  endmatching

Similarly for other predicates

$\mathcal{P}[\![\texttt{+}]\!] = \lambda v^* .$ **matching** v^*
  $\triangleright [v_1\ v_2]_{Value}$ $[\!]$ $(\textit{with-integer}\ v_1$
      $(\lambda i_1\ .\ (\textit{with-integer}\ v_2$
        $(\lambda i_2\ .\ (\textit{val-to-comp}$
          $(\textit{Int} \mapsto \textit{Value}\ (+\ i_1\ i_2)))))))$
  $\triangleright$ **else** $(\textit{err-to-comp}$ `+-wrong-number-of-args`$)$
  endmatching

Similarly for other binary operators, except `/` and `rem`, which give an error on a second argument of 0.

Figure 6.22: Valuation functions for FLK, Part II

▷ **Exercise 6.25** In FLK, `error` expressions take a manifest constant as the name of the error. There are other possible error strategies. One is to have only a single error value, which might simplify the semantics while making errors less helpful in practice. Another approach is to allow the argument of `error` to be a computed value. If we alter the syntax of FLK to support the form `(error` *E*`)`, then

a. Write the evaluation clause for `(error` *E*`)`.

b. What is the meaning of an `error` expression whose argument results in an error?

◁

▷ **Exercise 6.26** Construct an operational semantics for FLK that uses explicit environments rather than substitutions. [Hint: it is a good idea to introduce a closure object that pairs a lambda expression with the environment it is evaluated in.] ◁

▷ **Exercise 6.27** Write a denotational semantics for FL that does not depend on its desugaring into FLK. That is, the valuation clauses should directly handle features such as `define`, `let`, `letrec`, and procedures with multiple arguments. ◁

Chapter 7

Naming

A good name is rather to be chosen than great riches

— Proverbs 22:1

Naming is a central issue in programming language design. The fact that programming languages use names to refer to various objects and processes is at the heart of what makes them languages.

At the very least, a programming language must have a primitive set of names (literals and standard identifiers) and a means of combining the names into compound names (expressions). In a purely functional programming language, every expression is a name for the value it computes. In FL, for instance, `9`, `(+ 4 5)`, and `((lambda (a) (* a a)) (+ 1 2))` are just three different names for the number nine. In non-functional languages, there are more complex relationships between names and values that we shall explore later.

Expressions built merely out of primitives and a means of combination quickly become complex and cumbersome. Any practical language must also provide a means of abstraction for abbreviating a long name with a shorter one. Programming languages typically use symbolic identifiers as abbreviations and have binding constructs that specify the association between the abbreviation and the entity for which it stands. FL has the binding constructs `lambda`, `let`, `letrec`, and `define`; these are built on top of FLK's binding constructs: `proc` and `rec`. Using such constructs, it is possible to remove duplications to obtain more concise, readable, and efficient expressions. For example, naming allows us to transform the procedure:

```
(lambda (a b c)
  (list (+ (- 0 b)
           (sqrt (- (* b b)
                    (* 4 (* a c)))))
        (- (- 0 b)
           (sqrt (- (* b b)
                    (* 4 (* a c)))))
```

into the equivalent procedure:

```
(lambda (a b c)
  (let ((discriminant (sqrt (- (* b b)
                               (* 4 (* a c)))))
        (-b (- 0 b)))
    (list (+ -b discriminant)
          (- -b discriminant))))
```

Naming seems like such a simple idea that it's hard to imagine the subtleties hidden therein. A sampling of naming facilities in modern programming languages reveals a surprising number of ways to think about names. Some of the dimensions along which these facilities vary are:

- *Denotable Values*: What entities in a language can be named by global variables? By local variables? By formal parameters of procedures? By field names of a record?

- *Parameter Passing Mechanisms*: What is the relationship between the actual arguments provided to a procedure call and the values named by the formal parameters of the procedure?

- *Scoping*: How are new variables declared? Over what part of the program text and its associated computation does a declaration extend? How are references to a variable matched up with the associated declaration?

- *Name Control*: What mechanisms exist for structuring names to minimize name clashes in large programs?

- *Multiple Namespaces*: Can an identifier refer to more than one variable within a single expression?

- *Name Capture*: Does the language exhibit any name capture problems like those that cropped up with naïve substitution in FLK?

- *Side Effects*: Can the value associated with a name change over time?

The goal of this chapter is to explore many of the above dimensions. Our discussion of FL already introduced some of the basic concepts and terminology of naming, e.g., scope, free and bound variables, name capture, substitution, and environments. Here we give a fuller account of the issues involved in naming. Along the way, we shall pay particular attention to the effects that choices in naming design have on the expressive power of a language.

Certain naming issues (e.g., side effects, many parameter passing mechanisms) are intertwined with other aspects of dynamic semantics that we will cover later: state, control, data, nondeterminism, and concurrency. We defer these topics until the necessary concepts have been introduced.

7.1 Parameter Passing

Procedure application is the inverse operation to procedural abstraction. An abstraction packages formal parameters together with a body expression that refers to them, while application unpackages the body and evaluates it in a context where the formal parameters are associated with the arguments to the call. There are numerous methods for associating the formal parameter names with the arguments. These methods are called **parameter passing mechanisms**. Here we shall focus on two such mechanisms:

- In the **call-by-name (CBN)** mechanism, a formal parameter names the computation designated by an unevaluated argument expression. This corresponds to the non-strict argument evaluation strategy exhibited by FL in the previous chapter. CBN evolved out of the lambda calculus, and variants of CBN have found their way into ALGOL 60 and various functional programming languages (such as HASKELL and MIRANDA).

- In the **call-by-value (CBV)** mechanism, a formal parameter names the value of an evaluated argument expression. This corresponds to the strict argument evaluation strategy used by most modern languages (e.g., C, PASCAL, SCHEME, ML, SMALLTALK, POSTSCRIPT, etc.).

Later we shall explore additional parameter passing mechanisms (e.g., call-by-denotation on page 275 and call-by-reference in Chapter 8).

7.1.1 Call-by-Name and Call-by-Value: The Operational View

Figure 7.1 summarizes the difference between CBN and CBV in an operational framework. Both mechanisms share the following progress rule for the operator of a `call` expression, which is not shown in the figure:

$$\texttt{(call (proc } I\ E_1\texttt{)}\ E_2\texttt{)} \Rightarrow [E_2/I]E_1 \qquad [\textit{cbn-call}]$$

Call-By-Name

$$\frac{E_2 \Rightarrow E_2\,'}{\texttt{(call}\ V_1\ E_2\texttt{)} \Rightarrow \texttt{(call}\ V_1\ E_2\,'\texttt{)}} \qquad [\textit{cbv-rand-progress}]$$

$$\texttt{(call (proc } I\ E_1\texttt{)}\ V\texttt{)} \Rightarrow [V/I]E_1 \qquad [\textit{cbv-call}]$$

Call-By-Value

Figure 7.1: Essential operational semantics of CBN and CBV parameter passing.

$$\frac{E_1 \Rightarrow E_1\,'}{\texttt{(call}\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(call}\ E_1\,'\ E_2\texttt{)}} \qquad [\textit{rator-progress}]$$

Under CBN, the entire operand expression (not just its value) is substituted for the formal parameter of the abstraction. Figure 7.2 illustrates how this substitution works in some particular examples. Notice that the number of times the operand expression is evaluated under CBN depends on how many times the formal parameter is used within the body. If the formal is never used, the operand is never evaluated.

```
(call (proc x (primop * x x)) (primop + 2 3))
⇒ (primop * (primop + 2 3) (primop + 2 3))
⇒ (primop * 5 (primop + 2 3))
⇒ (primop * 5 5)
⇒ 25

(call (proc x 3) (primop / 1 0)) ⇒ 3

(call (proc x 3) (call (proc a (call a a))
                       (proc a (call a a)))) ⇒ 3
```

Figure 7.2: Under CBN, the entire operand expression is substituted for a formal parameter.

In the CBV strategy, the operand of an application is first completely evaluated, and then the resulting value is substituted for the formal parameter within

the body of the abstraction. The [*cbv-call*] rule is only applicable when the operand of the application is a value in the syntactic domain ValueExp of value expressions. The [*cbv-rand-progress*] rule permits evaluation of the operand. Together, these two rules force complete evaluation of the operand position before substitution.

Figure 7.3 shows some examples of CBV evaluation. The first example shows that in CBV, the operand expression is evaluated exactly once, regardless of how many times the formal is needed within the evaluation of the abstraction body. The other examples illustrate that CBV can yield errors or nontermination in cases where CBN would return a value.

```
(call (proc x (primop * x x)) (primop + 2 3))
    ⇒ (call (proc x (primop * x x)) 5)
    ⇒ (primop * 5 5)
    ⇒ 25

    (call (proc x 3) (primop / 1 0)) {This stuck expression models
                                      an error.}
    (call (proc x 3) (call (proc a (call a a))
                           (proc a (call a a))))
    ⇒ (call (proc x 3) (call (proc a (call a a))
                             (proc a (call a a))))
    ⇒ ...                  {An infinite loop.}
```

Figure 7.3: Under CBV, argument expressions are evaluated before being substituted for formal parameters.

Each of the two mechanisms has benefits and drawbacks. The above examples showed that CBN can evaluate an operand expression multiple times when the formal parameter is referenced more than once. This is less efficient than CBV, which is guaranteed to evaluate the operand exactly once. On the other hand, the CBV strategy of evaluating the operand exactly once may cause the computation to hang even when the operand value is not required by the procedure body. This is the situation where the CBN strategy of evaluating the operand at every parameter reference pays off; since there are no references, the operand is *never* evaluated. As Joseph Stoy has noted, CBN means evaluating the operand as many times as necessary, but sometimes this means no times at all! In this case CBN is "infinitely" more efficient than CBV, because it produces an answer when CBV does not.

The CBN mechanism has its roots in the lambda calculus, which is essentially a pared down version of FL that supports only applications, abstractions, and

variable references. CBN corresponds to a leftmost, outermost reduction strategy for the lambda calculus called **normal order reduction**. An important feature of normal order reduction in the lambda calculus is that it is guaranteed to find a **normal form** (i.e., value) for an expression if one exists. Furthermore, if any other reduction strategy finds a normal form, it must find the same one as the normal order strategy (modulo alpha equivalence).

The reduction strategy that corresponds to CBV, i.e., arguments must be reduced to normal form before substitution for formals, is called **applicative order reduction**.

We believe that a similar statement holds for FL:

> FLK **CBN/CBV Conjecture:** If an FLK expression $E \stackrel{*}{\Rightarrow}$CBV V, then $E \stackrel{*}{\Rightarrow}$CBN V', where V and V' are equivalent in some appropriate sense.

The fuzziness of "some appropriate sense" is due to the fact that ValueExp is different for the two mechanisms. In CBN, ValueExp includes `pair`s with arbitrary expression as parts, while in CBV, it includes only `pair`s with component values. As of this writing, we are still fleshing out a formal proof of this conjecture. But the intuition is clear: CBN terminates more often than CBV, and if they both terminate, they must terminate with "equivalent" values.

From the theoretical perspective, CBN clearly seems superior to CBV. Then why do so many languages use CBV and hardly any use CBN? As hinted above, a pragmatic reason is that CBN implies certain implementation overheads. Perhaps an even more important reason is that CBN and side-effects do not mix. As we shall see in the next chapter, imperative programs using CBN are notoriously hard to reason about. But here we shall focus only on the issue of overheads.

As a non-trivial example, let's compare the CBN and CBV mechanisms on the following call to an iterative factorial procedure written in FL:[1]

```
((rec fact-iter (lambda (n ans)
                  (if (= n 0)
                      ans
                      (fact-iter (- n 1) (* n ans)))))
 3 1)
```

Transition sequences for the two parameter passing mechanisms are shown in Figures 7.4 and 7.5. In the transition sequences, the abbreviation *FACT-ITER* stands for the expression

[1] We use FL rather than FLK for this example because it would be too cumbersome to express in FLK. We assume in the example an SOS in which FL expressions are appropriate configurations. For instance, in this SOS, multi-argument applications are performed in a single rewrite step.

```
(rec fact-iter (lambda (n ans)
                 (if (= n 0)
                     ans
                     (fact-iter (- n 1) (* n ans)))))
```

while the abbreviation *UNWOUND-FACT-ITER* stands for the expression

```
(lambda (n ans)
  (if (= n 0)
      ans
      (FACT-ITER (- n 1) (* n ans))))
```

```
(FACT-ITER 3 1)
⇒* (UNWOUND-FACT-ITER 3 1)
⇒* (if (= 3 0) 1 (FACT-ITER (- 3 1) (* 3 1)))
⇒* (if #f 1 (FACT-ITER (- 3 1) (* 3 1)))
⇒* (FACT-ITER (- 3 1) (* 3 1)))
⇒* (UNWOUND-FACT-ITER (- 3 1) (* 3 1)))
⇒* (UNWOUND-FACT-ITER 2 (* 3 1)))
⇒* (UNWOUND-FACT-ITER 2 3)
⇒* (if (= 2 0) 3 (FACT-ITER (- 2 1) (* 2 3)))
⇒* (if #f 3 (FACT-ITER (- 2 1) (* 2 3)))
⇒* (FACT-ITER (- 2 1) (* 2 3))
⇒* (UNWOUND-FACT-ITER (- 2 1) (* 2 3))
⇒* (UNWOUND-FACT-ITER 1 (* 2 3))
⇒* (UNWOUND-FACT-ITER 1 6)
⇒* (if (= 1 0) 6 (FACT-ITER (- 1 1) (* 1 6)))
⇒* (if #f 6 (FACT-ITER (- 1 1) (* 1 6)))
⇒* (FACT-ITER (- 1 1) (* 1 6)))
⇒* (UNWOUND-FACT-ITER (- 1 1) (* 1 6)))
⇒* (UNWOUND-FACT-ITER 0 (* 1 6))
⇒* (UNWOUND-FACT-ITER 0 6)
⇒* (if (= 0 0) 6 (FACT-ITER (- 0 1) (* 0 6)))
⇒* 6
```

Figure 7.4: CBV transition path computing the iterative factorial of 3.

As indicated by the figures, CBN can be much less efficient than CBV. There are two important sources of overhead:

1. CBN often requires more *time*[2] than CBV in the case where an argument

[2]Here we assume that the time taken by an evaluation is related to the number of evaluation steps in the operational semantics. This is often a reasonable assumption.

```
(FACT-ITER 3 1)
⇒* (UNWOUND-FACT-ITER 3 1)
⇒* (if (= 3 0) 1 (FACT-ITER (- 3 1) (* 3 1)))
⇒* (if #f 1 (FACT-ITER (- 3 1) (* 3 1)))
⇒* (FACT-ITER (- 3 1) (* 3 1)))
⇒* (UNWOUND-FACT-ITER (- 3 1) (* 3 1)))
⇒* (if (= (- 3 1) 0) (* 3 1) (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (if (= 2 0) (* 3 1) (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (if #f (* 3 1) (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
⇒* (UNWOUND-FACT-ITER (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
⇒* (if (= (- (- 3 1) 1) 0)
        (* (- 3 1) (* 3 1))
        (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (if (= (- 2 1) 0)
        (* (- 3 1) (* 3 1))
        (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (if (= 1 0)
        (* (- 3 1) (* 3 1))
        (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (if #f
        (* (- 3 1) (* 3 1))
        (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))))
⇒* (FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (UNWOUND-FACT-ITER (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))
⇒* (if (= (- (- (- 3 1) 1) 1) 0)
        (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
        (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if (= (- (- 2 1) 1) 0)
        (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
        (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if (= (- 1 1) 0)
        (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
        (FACT-ITER (- (- (- (- 3 1) 1) 1) 1)
                   (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if (= 0 0)
        (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
        (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (if #t
        (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
        (FACT-ITER (- (- (- (- 3 1) 1) 1) 1) (* (- (- (- 3 1) 1) 1) (* (- (- 3 1) 1) (* (- 3 1) (* 3 1))))))
⇒* (* (- (- 3 1) 1) (* (- 3 1) (* 3 1)))
⇒* (* (- 2 1) (* (- 3 1) (* 3 1)))
⇒* (* 1 (* (- 3 1) (* 3 1)))
⇒* (* 1 (* 2 (* 3 1)))
⇒* (* 1 (* 2 3))
⇒* (* 1 6)
⇒* 6
```

Figure 7.5: CBN transition path computing the factorial of 3.

is used more than once in the body of an abstraction, because then the same argument expression must be evaluated multiple times. For example, in Figure 7.5, the value of `(- 3 1)` is calculated five times, compared to only once in Figure 7.4.

2. CBN often requires more *space* than CBV because expressions whose values are not currently needed may grow as their evaluations are deferred until later. For example, the expression in the `ans` operand position of a call to *FACT-ITER* grows by one multiplication with every recursive call.

In practice, there are techniques for ameliorating both of these sources of overhead. The time inefficiency is typically finessed by **memoization**, a technique that evaluates an operand and caches its value the first time it is referenced. Further references simply return the cached value rather than evaluating the operand again. We shall bump into it again when we study **lazy evaluation** in Chapter 8.

The space overhead is perhaps more insidious. It can be improved by graph-based expression representations that share substructure, but this trick does not stop the space consumed by operands for parameters like `ans` from growing in size with every call. This problem, known as a **dragging tail**, is disturbing because it destroys desirable space management properties for FL. The CBV version of factorial requires only a constant amount of space to keep track of control and state variables.[3] In contrast, the CBN version requires space for the unevaluated state variables that grows linearly with the input to factorial. When executing these kinds of programs on a machine with finite storage resources, a CBN strategy is more likely to run out of space than a CBV strategy. A technique called **strictness analysis** can improve CBN by modifying it to use CBV for operand evaluation when it is possible to prove that the operand will be required at least once.

The various tricks for improving CBN make it much more palatable, but the techniques still require overheads that some implementors find unacceptable. For example, memoization implies that a flag must be tested at every variable reference. Since variable references are rather common, the extra check is often considered prohibitive without special hardware support.

Parameter passing mechanisms are used to describe not only procedure calls, but also other constructs that bind names to values. For example, FL's binding constructs (`lambda`, `let`, `letrec`, and `define`) have an implicit method for associating names with values because they desugar into FLK's abstractions and

[3]For simplicity, we ignore the fact that the larger numbers required for larger inputs to factorial actually require more space.

applications. We shall assume (unless stated otherwise) that all binding constructs inherit their parameter passing mechanism from procedure calls. Thus, `(let ((a (/ 1 0))) 3)` evaluates to `3` in a CBN language, but generates an error in a CBV language.

▷ **Exercise 7.1** In a CBV language, it is often useful to delay the evaluation of an argument until a later time. This behavior can be specified with the pair of constructs `(lazy` *E*`)` and `(touch` *E*`)`. Informally, `(lazy` *E*`)` wraps *E* up without evaluating it, while `(touch` *E*`)` unwraps *E* until it is no longer embedded in a `lazy`. On a non-`lazy` value, `touch` acts as an identity. For example, in CBV FL:

```
(touch (+ 1 2)) ⟶FL 3

(touch (lazy (+ 1 2))) ⟶FL 3

(touch (lazy (lazy (+ 1 2)))) ⟶FL 3

(let ((a (lazy (+ 1 2)))
      (b (lazy (/ 1 0))))
  (touch a)) ⟶FL 3

(let ((a (lazy (+ 1 2)))
      (b (lazy (/ 1 0))))
  (touch b)) ⟶FL error:divide-by-zero
```

a. Extend the operational semantics of FLK to handle `lazy` and `touch`. Recall that an SOS has five parts; make whatever changes are necessary to each of the parts.

b. Can `lazy` and `touch` be implemented by syntactic sugar? If so, give the desugarings; if not, explain why.

c. Show how to translate CBN FL into a CBV version of FL that is equipped with `lazy` and `touch`. ◁

▷ **Exercise 7.2** The desugaring rule for `letrec` specified in Section 6.2 was designed for a CBN language. Does it still work in a CBV language? Explain, using concrete example(s) to justify your answer.

◁

▷ **Exercise 7.3** Show by example that an FL expression that diverges under CBN need not diverge under CBV. How does this fact relate to the CBN/CBV conjecture made above? ◁

7.1.2 Call-by-Name and Call-by-Value: The Denotational View

The denotational descriptions of CBN and CBV FL semantics have an interesting and important difference. This difference is found in the kinds of things that a formal parameter can name. In CBN, a formal parameter can name a value, an error, or a non-terminating computation. In short, a formal parameter in CBN can name the unrestricted meaning of an FL expression. This makes sense, as in CBN we conceptually pass an unevaluated expression as an actual parameter, and we only evaluate an actual parameter when absolutely necessary. In CBN, a formal parameter can name the arbitrary result of an FL expression. A CBV FL semantics is substantially more restrictive than a CBN semantics. In CBV, we can only pass values (e.g., integers, booleans, pairs, etc.) as actual parameters, and thus formal parameters can only name values. As we shall describe in detail below, the difference in the kinds of entities that can be named by formal parameters is reflected in the definition of the *Denotable* domain.

Rather than giving the clauses of the valuation function $\mathcal{E}$ in full for each of the parameter passing mechanisms, we shall only describe clauses for those constructs that are pertinent to parameter passing: variable references, `proc`, and `call`. (The `rec` and `pair` clauses are also relevant, but we defer discussion of them until later.) The valuation clause for `proc` is the same for both mechanisms:

$$\mathcal{E}[\![\texttt{(proc } I\ E\texttt{)}]\!] = \lambda e\,.\ (\textit{val-to-comp}\ (\textit{Procedure} \mapsto \textit{Value}\ (\lambda \delta\,.\ (\mathcal{E}[\![E]\!]\ [I : \delta]e))))$$

Figure 7.6 gives the denotational semantics for CBN versus CBV. The essential difference is that CBN environments name elements of *Computation* while CBV environments can only name elements of *Value*. In FL,

$$\textit{Computation} = \textit{Expressible} = (\textit{Value} + \textit{Error})_{\perp}$$

so that CBN environments can name all expressible values (including error and divergence), while CBV environments can only name "regular" values.

The CBN domain equation

$$\delta \in \textit{Denotable} = \textit{Computation}$$

indicates that a formal parameter can denote any computation, which in the case of FL includes error and divergence. A CBN procedure can return an element of *Value* even when it is passed one of these irregular values. Thus, call-by-name procedures are not strict. Here's an example of CBN (where e is an arbitrary environment):

$$\delta \in \textit{Denotable} = \textit{Computation}$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\ (\textit{with-denotable}\ (\textit{lookup}\ e\ I)\ (\lambda\delta\,.\,\delta))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\ (\textit{with-procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\ (p\ \ (\mathcal{E}[\![E_2]\!]\ e))))$$

Call-By-Name

$$\delta \in \textit{Denotable} = \textit{Value}$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\ (\textit{with-denotable}\ (\textit{lookup}\ e\ I)\ (\lambda\delta\,.\ (\textit{val-to-comp}\ \delta)))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\ (\textit{with-procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\ (\textit{with-value}\ (\mathcal{E}[\![E_2]\!]\ e)\ p)))$$

Call-By-Value

Figure 7.6: Essential denotational semantics of CBN and CBV parameter passing. For FLK, *Computation* = *Expressible*, but the CBN semantics will still be valid later when *Computation* is updated to reflect extensions to FLK. Likewise, the CBV semantics will still be valid when the *Value* domain is extended.

$$
\begin{array}{l}
(\mathcal{E}[\![\texttt{(call (proc x 3) (primop / 1 0))}]\!]\ e) \\
= (\textit{with-procedure}\ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e)\ (\lambda p\ .\ (p\ \ (\mathcal{E}[\![\texttt{(primop / 1 0)}]\!]\ e)))) \\
= (\textit{with-procedure}\ (\textit{val-to-comp} \\
\qquad\qquad\qquad\qquad (\textit{Procedure} \mapsto \textit{Value} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\delta\ .\ (\mathcal{E}[\![\texttt{3}]\!]\ [\texttt{x} : \delta]e)))) \\
\qquad (\lambda p\ .\ (p\ \ \textit{error}))) \\
= ((\lambda\delta\ .\ (\mathcal{E}[\![\texttt{3}]\!]\ [\texttt{x} : \delta]e))\ \ \textit{error}) \\
= (\mathcal{E}[\![\texttt{3}]\!]\ [\texttt{x} : \textit{error}]e) \\
= (\textit{val-to-comp}\ (\textit{Int} \mapsto \textit{Value}\ 3))
\end{array}
$$

The CBV domain equation

$$\delta \quad \in \quad \textit{Denotable} \quad = \quad \textit{Value}$$

indicates that identifiers may be bound to values such as integers, booleans, symbols, procedures, and pairs, but may *not* be bound to objects denoting an error or nontermination. The treatment of error and bottom is the only semantic difference between CBV and CBN.

The CBV clause for `call` uses *with-value* to guarantee that only elements of the domain *Value* are passed to p. This accounts for the strict nature of CBV evaluation. As an illustration of CBV, let's once again consider the meaning of the FLK expression `(call (proc x 3) (primop / 1 0))`:

$$
\begin{array}{l}
(\mathcal{E}[\![\texttt{(call (proc x 3) (primop / 1 0))}]\!]\ e) \\
= (\textit{with-procedure}\ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e) \\
\qquad (\lambda p\ .\ (\textit{with-value}\ (\mathcal{E}[\![\texttt{(primop / 1 0)}]\!]\ e)\ \ p))\ ) \\
= (\textit{with-procedure}\ \ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e)\ \ (\lambda p\ .\ (\textit{with-value}\ \textit{error}\ p))) \\
= (\textit{with-procedure}\ \ (\mathcal{E}[\![\texttt{(proc x 3)}]\!]\ e)\ \ (\lambda p\ .\ \textit{error})) \\
= (\textit{with-procedure}\ (\textit{val-to-comp} \\
\qquad\qquad\qquad\qquad (\textit{Procedure} \mapsto \textit{Value} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\delta\ .\ (\mathcal{E}[\![\texttt{3}]\!]\ [\texttt{x} : \delta]e))\ )) \\
\qquad (\lambda p\ .\ \textit{error})\ ) \\
= \textit{error}
\end{array}
$$

7.1.3 Discussion

7.1.3.1 Extensions

Numerous additional features may be layered on top of the above mechanisms to yield further variations in parameter passing for functional languages. For example, it is possible to pass parameters by keyword, to specify optional arguments, or to describe formal parameters that are pattern-matched against arguments that are compound data structures. While these are important ways of capturing common patterns of usage, they are orthogonal to and less fundamental than the CBN vs. CBV distinction. The introduction of side-effects, on

the other hand, will lead to fundamental variations of the above mechanisms. We will explore these in Chapter 8.

It is possible to include more than one parameter passing mechanism within a single language. This possibility is explored in Exercise 7.7.

7.1.3.2 Non-Strict vs. Strict Pairs

Data constructors (such as `pair`) are typically non-strict in a CBN language but strict in a CBV language. Figure 7.7 summarizes the operational and denotational differences between non-strict and strict pairs. In both perspectives, the difference boils down to whether the components of pair values must themselves be values. The figure omits the semantics of the `left` and `right` primitives, which do not differ between the non-strict and strict versions.

7.1.3.3 Denotable vs. Passable Values vs. Component Values

We have assumed in our discussion that every entity that is nameable may be passed as an argument or bundled into a pair. While this tends to be true in (and is a major source of power for) functional languages, it is not true in general. For example, while procedures are almost universally nameable, there are many languages (e.g., FORTRAN, BASIC, PASCAL) in which procedures cannot be passed as arguments, or can only be passed in a limited way. Similarly, many languages do not permit data structure components to be procedures. In order to give an accurate denotational description of such languages, it is necessary to distinguish the class of nameable entities from those which may be passed as arguments and those which can be components of data structures. We could therefore introduce new domains, *Passable* and *Component*, that describe these classes of values.

7.1.3.4 Semantic Derivation of Thunking

The denotational descriptions of parameter passing emphasize that CBN and CBV differ only in their treatment of error and nontermination. They also give another perspective on simulating CBN in a CBV language. From a denotational view, the essence of such a simulation in CBV FL is to find a way of naming *error* and $\perp_{Computation}$. It is not possible to name these directly in a CBV language, but it is always possible to name them indirectly, via procedures. For every computation c, we can construct a procedural value that returns c when called:

$(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ \ (\lambda\delta\ .\ c)))$

Operational:

$V \in \text{ValueExp} = \ldots \cup \{(\texttt{pair}\ E_1\ E_2)\}$

Non-Strict Pairs

$V \in \text{ValueExp} = \ldots \cup \{(\texttt{pair}\ V_1\ V_2)\}$

$$\frac{E_1 \Rightarrow E_1'}{(\texttt{pair}\ E_1\ E_2) \Rightarrow (\texttt{pair}\ E_1'\ E_2)} \quad [\textit{pair-left-progress}]$$

$$\frac{E_2 \Rightarrow E_2'}{(\texttt{pair}\ V_1\ E_2) \Rightarrow (\texttt{pair}\ V_1\ E_2')} \quad [\textit{pair-right-progress}]$$

Strict Pairs

Denotational:

$a \in \mathit{Pair} = \mathit{Computation} \times \mathit{Computation}$

$\mathcal{E}[\![(\texttt{pair}\ E_1\ E_2)]\!] = \lambda e\,.\ (\textit{val-to-comp}\ (\mathit{Pair} \mapsto \mathit{Value}\ \langle(\mathcal{E}[\![E_1]\!]\ e),\ (\mathcal{E}[\![E_2]\!]\ e)\rangle))$

Non-Strict Pairs

$a \in \mathit{Pair} = \mathit{Value} \times \mathit{Value}$

$\mathcal{E}[\![(\texttt{pair}\ E_1\ E_2)]\!] =$
$\quad \lambda e\,.\ (\textit{with-value}\ (\mathcal{E}[\![E_1]\!]\ e)$
$\qquad (\lambda v_1\,.\ (\textit{with-value}\ (\mathcal{E}[\![E_2]\!]\ e)$
$\qquad\quad (\lambda v_2\,.\ (\textit{val-to-comp}\ (\mathit{Pair} \mapsto \mathit{Value}\ \langle v_1, v_2\rangle)))$

Strict Pairs

Figure 7.7: Operational and denotational views of non-strict and strict pairs.

This means that we can effectively put any computation into an environment by transforming it into the above form before it is bound to a name and then perform the inverse transformation when the name is looked up. Since the parameter δ of the above form is ignored, the transform and its inverse are equivalent to, respectively, creating a procedure of no arguments (a so-called **thunk**) and calling that procedure on no arguments.

7.1.3.5 CBV Versions of `rec`

In an operational semantics, `rec` is handled by the same rule regardless of whether the language is CBN or CBV:

$$\texttt{(rec } I \ E\texttt{)} \Rightarrow [\texttt{(rec } I \ E\texttt{)}/I]E \qquad [rec]$$

Unforutnately, things are not so simple in a denotational semantics. In CBN, where *Denotable* = *Computation*, the valuation clause for a CBN version of `rec` is very pretty:

$$\mathcal{E}[\![\texttt{(rec } I \ E\texttt{)}]\!] = \lambda e\,.\ \mathbf{fix}_{Computation}(\lambda c\,.\,(\mathcal{E}[\![E]\!]\ [I : c]e))$$

The fixed point defined by this clause is well-defined as long as *Computation* is a pointed CPO. For this reason, we will always guarantee that *Computation* is a pointed domain.

However, developing a valuation clause for `rec` in a CBV language is rather tricky. In CBV, the corresponding version of the CBN clause is:

$$\mathcal{E}[\![\texttt{(rec } I \ E\texttt{)}]\!] = \lambda e\,.\ \mathbf{fix}_{Value}(\lambda v\,.\,(\mathcal{E}[\![E]\!]\ [I : v]e))$$

But since *Denotable*= *Value* is *not* a pointed domain, the fixed point is not well defined, and the clause is nonsensical.

There are several ways of circumventing this impasse. Here we present two approaches:

1. In `(rec` I E`)`, we can limit E to a subset of expressions that are syntactically guaranteed to be procedures. In CBV,

   $$Proc = Value \rightarrow Computation$$

   is pointed (because *Computation* is always pointed), so it is always possible to fix over *Procedure*. That is, suppose we modify the syntax of FLK as follows:

   E ::= ...
   | `(rec` I_{var} A_{body}`)` [Recursion]

   A ::= `(proc` I_{formal} E_{body}`)` [Abstraction]

If we suppose that $\mathcal{A}$ is a valuation function of signature

$$\text{Abstraction} \to \mathit{Environment} \to \mathit{Procedure}$$

then `rec` is definable as:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(rec } I\ A\texttt{)}]\!] = \\ \quad \lambda e\,.\ (\mathit{Value} \mapsto \mathit{Expressible} \\ \qquad (\mathit{Procedure} \mapsto \mathit{Value} \\ \qquad\quad (\mathbf{fix}_{\mathit{Procedure}} \\ \qquad\qquad (\lambda p\,.\ ((\mathcal{A}\ [\![A]\!])(\mathit{extend}\ e \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad I \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Value} \mapsto \mathit{Denotable} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{Procedure} \mapsto \mathit{Value}\ p)\ ))))))\end{array}$$

Unfortunately, the syntactic restriction results in a restriction of the expressive power of `rec`. It is no longer possible to specify recursions over pairs or over procedures whose describing expression is not a manifest `proc`. The following examples, though contrived, are indicative of useful patterns that are disallowed by an approach that requires the body of a `rec` to be a manifest abstraction in a CBV language:

```
(rec ones (pair 1 (lambda () ones)))

(rec fact
  (let ((fact-of-0 1))
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            fact-of-0
            (* n (fact (- n 1))))))))
```

2. An alternative is to make use of $\perp_{\mathit{Binding}}$ to compute fixed points. Here is a version that works in the case of $\mathit{Computation} = \mathit{Expressible}$:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(rec } I\ E\texttt{)}]\!] = \lambda e\,.\ (\mathbf{fix}_{\mathit{Computation}} \\ \qquad\qquad\qquad\qquad (\lambda c\,.\ (\mathcal{E}[\![E]\!]\ [I :: (\mathit{extract\text{-}value}\ c)]e))\end{array}$$

$$\begin{array}{l}\mathit{extract\text{-}value} : \mathit{Computation} \to \mathit{Binding} \\ = \lambda c\,.\ \mathbf{matching}\ c \\ \qquad \triangleright\,(\mathit{Value} \mapsto \mathit{Expressible}\ v)\ [\!]\ (\mathit{Denotable} \mapsto \mathit{Binding}\ v) \\ \qquad \triangleright\,(\mathit{Error} \mapsto \mathit{Expressible}\ \text{error})\ [\!]\ \perp_{\mathit{Binding}} \\ \qquad \mathbf{endmatching}\end{array}$$

extract-value coerces a computation into a binding. The resulting binding is either an element of $\mathit{Denotable} = \mathit{Value}$, or it is $\perp_{\mathit{Binding}}$. (Recall that

matching is strict, so that *extract-value* maps $\perp_{Computation}$ to $\perp_{Binding}$.) By effectively naming a bottom element in the environment, this trick gives a starting point for the fixed-point iteration. It would also be possible to add a bottom element directly to the *Denotable* domain, but that would not faithfully model the intuition behind CBV. The $\perp_{Binding}$ element helps to clarify the difference between using bottom to solve a recursion equation expressed by a `rec` and allowing bottom to be passed as an argument to a procedure.

It is worth noting that the *extract-value* function works only for *Computation* = *Expressible* and needs to be tweaked if the domain of computations changes.

7.1.3.6 The Perils of Reading Denotational Descriptions Operationally

The valuation clause for `call` in CBN illustrates the potential dangers of giving operational interpretations to denotational definitions. If $(p\ \ (\mathcal{E}[\![E_2]\!]\ e))$ were read as if it were, for example, a LISP or SCHEME program fragment, it would say something like: "First evaluate the expression E_2 in the environment e and then call p on the resulting value." Unfortunately, this reading introduces inappropriate notions of evaluation order and time (based on when the argument is evaluated) that are inherited from the CBV nature of LISP or SCHEME. Such a reading can cause confusion in the cases where E_2 denotes error or bottom; the reader might (incorrectly) think that, as in LISP, errors or bottom in an argument would propagate to errors or bottom for the call.

Rather than reading $\mathcal{E}$ as "evaluate" (in the operational sense), it is safer to read it as "the meaning of." Thus $(p\ \ (\mathcal{E}[\![E_2]\!]\ e))$ means "Apply p to the meaning of E_2 in the environment e." Additionally, it is important to remember that the application of a total function (as opposed to the application of a procedure in a programming language) is well-defined as long as the arguments are in the appropriate domains. In particular, the result is independent of any evaluation strategy that might be associated with the metalanguage expressions used to represent the application. For example, the term

$$(\lambda y\,.\,3)\,((\lambda x\,.\,xx)\,(\lambda x\,.\,xx))$$

denotes 3 even though a CBV-like strategy for equation rewriting would not find this value.

In the case of $(p\ \ (\mathcal{E}[\![E_2]\!]\ e))$, if the meaning of E_2 is error or bottom, these are simply handed to p, which is constructed by the clause for `proc`. The `proc` clause shows that any such argument is simply associated with an identifier in

the environment, while the variable reference clause indicates that this denotable value is retrieved upon lookup without any ado.

It is natural to wonder at this point how error or nontermination in an argument can *ever* lead to error or nontermination for an application in a CBN language. That is, if an argument is simply inserted into the environment upon call and retrieved upon lookup, who ever actually *examines* the value denoted by the argument? There are several spots in the denotational definition where information about the values is required. For example, in the clause for $[\![$`(call` E_1 E_2`)`$]\!]$ the value of E_1 is required to be the denotation of a procedure; the semantics must check the value to ensure this is the case (such checks are hidden in the abstraction *with-procedure*). Similarly, an `if` clause must not only check that the test expression denotes a boolean, but also uses the boolean value in order to determine which arm is denoted by the entire conditional construct. Handling primitive operators in FLK is perhaps the most common case where details of the values must be examined.

These facts imply that in any implementation of CBN FLK, the argument expression E_2 *cannot* be evaluated before the procedure p is invoked. For if E_2 initiated a nonterminating computation, p would never be invoked. The moral of this discussion is that operational conclusions of this sort aren't always obvious from a denotational definition. Indeed, factoring out operational concerns is a source of power for denotational semantics. It is not necessary to worry about details like the practical implications of binding errors or nontermination in an environment. Instead, the denotational approach helps us to focus on high-level descriptions like: "The essential difference between CBN and CBV is that the former allows errors and nontermination to be named whereas the latter does not."

7.1.3.7 Call-by-Denotation

Sometimes a denotational semantics suggests alternative perspectives. A case in point is **call-by-denotation (CBD)**, a parameter passing mechanism that is obtained by tweaking call-by-name semantics in a straightforward way (see Figure 7.8). Whereas call-by-name determines the meaning of an argument expression relative to the environment available at the point of call, call-by-denotation instead determines the meaning of an argument expression relative to the environment where the formal parameter is referenced.

In this case, the domain equation

$$\delta \in \mathit{Denotable} = \mathit{Environment} \to \mathit{Computation}$$

indicates that the nameable entities in the language are functions that map

$\delta \in \mathit{Denotable} = \mathit{Environment} \to \mathit{Computation}$

$\mathcal{E}[\![I]\!] = \lambda e\,.\ (\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda\delta\,.\ (\delta\ e)))$

$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\ (\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\ (p\ \mathcal{E}[\![E_2]\!])))$

Call-By-Denotation

Figure 7.8: Essential semantics of call-by-denotation.

environments to computations. The `call` clause simply embeds the actual parameter in a function of this type. Variable references are handled by applying the function associated with the variable to the environment in effect where the variable is referenced.

CBD is not very useful but does model some of the name capture problems associated with macro expansion. As a somewhat bizarre example of CBD, consider the meaning of the FL expression

```
((let ((x 3))
   (lambda (y) y))
 x)
```

in an environment e_0 in which the identifier `x` is not bound. In both call-by-name and call-by-value, the meaning of this expression is an error because the value of (the outer) `x` is required but nowhere defined. In call-by-denotation, however, the body of the identity procedure — i.e., the variable `y` — will eventually be evaluated in an environment e_1 where `x` is bound to

$$(\mathit{Denotable} \mapsto \mathit{Binding}\ (\lambda e\,.\,(\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))))$$

and `y` is bound to

$$(\mathit{Denotable} \mapsto \mathit{Binding}\ (\lambda e\,.\ (\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ \texttt{x})\ (\lambda\delta\,.\ (\delta\ e)))))$$

(We leave the details of how this point is reached as an exercise.) At this point, the denotation of `y` will be applied to e_1, with the following result:

$$\begin{aligned}
&((\lambda e\,.\ (\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ \texttt{x})\ (\lambda\delta\,.\ (\delta\ e))))\ e_1)\\
&= (\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e_1\ \texttt{x})\ (\lambda\delta\,.\ (\delta\ e_1)))\\
&= (\mathit{with\text{-}denotable}\ (\mathit{Denotable} \mapsto \mathit{Binding}\\
&\qquad\qquad (\lambda e\,.\,(\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))))\\
&\qquad (\lambda\delta\,.\ (\delta\ e_1)))\\
&= ((\lambda\delta\,.\ (\delta\ e_1))\ (\lambda e\,.\,(\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))))\\
&= (\mathit{Value} \mapsto \mathit{Expressible}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))
\end{aligned}$$

Thus, in a CBD semantics, the meaning of this expression is the number 3!

The weird behavior of call-by-denotation in this example is due to a kind of name capture. The evaluation of the outer `x` yields not what we would normally think of as a value but an environment accessor that is eventually applied to an environment with a binding for the inner `x`. Had the inner `x` been named something other than `x` or `y`, no capture would have occurred, and the expression would have denoted an error, as expected. But `x` is not the only outer name which would cause trouble; if we replace the outer `x` by a reference to `y`, the expression diverges! (Check it and see.) Because the semantics of call-by-denotation are so convoluted, it is hardly surprising that this mechanism is not used in any real programming language. (However, call-by-denotation does exhibit some of the behavior of macro languages.)

Then what is our purpose in introducing so contrived a mechanism? First, we wanted to emphasize that in a denotational semantics, names can be bound to entities much more complex than simple values or expressible values. We shall see many examples of this in the future. Second, we wanted to emphasize that just because a mechanism has an elegant denotational description doesn't necessarily mean that it is of any use in real programming languages. Although in some cases denotational descriptions *do* suggest powerful language constructs, other extensions suggested by denotational semantics (like call-by-denotation) turn out to be downright duds.

▷ **Exercise 7.4** For each of the following FL expressions, use the denotational semantics to give the meaning of the expression in CBN, CBV, and CBD. (Recall that `let` and `lambda` inherit their semantics from `proc`.)

a.
```
(let ((x 3)
      (y (/ 1 0)))
  x)
```

b.
```
(let ((x 7))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 10))
      (f x))))
```

c.
```
(let ((x 3))
  ((let ((y 19))
     (lambda (z) y))
   x))
```

d.
```
(let ((x 23))
  (let ((x x))
    x))
```
◁

▷ **Exercise 7.5**

a. Write a single FL expression that has a different meaning in each of the three parameter passing mechanisms. Give the meaning of your expression in each mechanism.

b. Bud Lojack hopes to solve part a. above with an expression that evaluates to one of the symbols `call-by-name`, `call-by-value`, or `call-by-denotation` depending on which parameter passing mechanism is being used. Kindly explain why the expression that Bud desires does not exist. ◁

▷ **Exercise 7.6**

a. Can a CBN FL interpreter be written in CBV FL?

b. Can a CBV FL interpreter be written in CBN FL?

Justify your answers. ◁

▷ **Exercise 7.7** In this exercise we explore combining call-by-name and call-by-value in a single language.

Imagine a language NAVALNAVAL (NAme/VAlue Language) that is just like CBN FLK except that `(proc` *I E*`)` has been replaced by the two constructs `(vproc` *I E*`)` and `(nproc` *I E*`)`. Both of these constructs act like `proc` in that they create single-argument procedures. The only difference between them is that procedures created by `nproc` pass parameters using CBN, while those created by `vproc` use CBV.

For example:

`(call (nproc x 3) (primop / 1 0))` $\xrightarrow[FL]{}$ *3*
`(call (vproc x 3) (primop / 1 0))` $\xrightarrow[FL]{}$ *error:divide-by-zero*

a. Provide a denotational semantics for NAVAL (only give those domain equations and valuation clause that differ from those for CBN FLK). Hint: In a simple approach to this problem, by-name and by-value procedures can both be elements of a single *Procedure* domain. What should *Denotable* be?

b. Just as it was convenient to extend FLK with the notion of multiple argument procedures, it would be nice to extend NAVAL with a similar notion. Some method must be chosen for specifying which parameters are by name and which are by value. For example, parameters might default to the by-value mechanism, but could be declared by-name with the token `name`, as illustrated below:

```
(define unless
  (lambda (test (name default) (name exception))
    (if test exception default)))
```

`(unless (= 1 2) (+ 3 4) (/ 5 0))` $\xrightarrow[eval]{}$ *[int 7]*

Give the rules for desugaring such a multiple-argument `lambda` construct into NAVAL's one-argument `nproc` and `vproc`.

c. We have seen how CBN can be simulated in a CBV language using thunks. Formalize this transformation by showing how to translate NAVAL into CBV FLK. ◁

▷ **Exercise 7.8** Write a program that translates CBN FL into CBV FL. (Note that a very similar program could be used to translate CBN FL into SCHEME.) ◁

▷ **Exercise 7.9** Show that when *Computation* = *Expressible*, the CBV valuation clause for `rec` denotes the same function as the CBN valuation clause for `rec`. ◁

▷ **Exercise 7.10** Use both the operational and denotational `rec` semantics to compute the values of the following expressions in both CBN and CBV versions of FL:

a. `(rec a 4)`

b. `(rec a (1 0))`

c. `(rec a a)`

d. `(rec a b)`

e. `(rec a (if #t 3 a))`

f. `(rec a (lambda (x) a))`

g. `(rec a (pair 1 a))`

h. `(rec a (pair (lambda (x) a)))`

i. `(let ((b 3)) (rec a b))` ◁

7.2 Name Control

The phrase "too much of a good thing" evokes images such as a child getting a stomach ache after eating too much candy or the crew of the starship Enterprise being swamped by the cute but prolific tribbles. The recent explosion in information technology has added a new twist to this phrase. Information consumers,

such as television viewers, magazine subscribers, and readers of electronic mail, now have rapid access to incredible stores of information. But these changes in the information landscape have brought new problems, perhaps the most daunting of which is information overload: there is simply too much information to weed through, to absorb, to remember.

The area of naming in programming languages harbors its own version of the information overload problem. While names are an indispensable means of abstraction, the abundance of names in even modestly sized programs can lead to a host of complications.

From a cognitive point of view, more names can mean more learning, remembering, and model building for programmers. One of the simplest naming strategies, a single global namespace (in which distinct variables are named by distinct identifiers), is also one of the most nightmarish for program writers and readers alike. This approach foists a tremendous amount of mental bookkeeping on the programmer:

- Nonlocality of naming structure impairs readability because there are no constraints on which names the user needs to search or remember in order to understand a given fragment of code. At all times, the reader must potentially be aware of the entire namespace. For this reason, a global namespace is not scalable in a cognitive sense; large programs are much harder to comprehend than shorter ones.

- The reader has to infer structural groupings intended by the writer but not expressed due to the flatness of the namespace.

- Every time a new name is needed, the writer must find one that does not clash with any names already in use.

In order to reduce such unreasonable cognitive demands, programming languages typically provide mechanisms for reusing names and structuring the scope of names. Even when these are not supported by the naming system, programmers often develop naming conventions to simulate such mechanisms.

From an engineering point of view, more names can mean more complex interactions between program parts. One of the chief methods of controlling the complexity of large programs is to break them up into smaller units having well-defined **interfaces** that separate the use of a unit from its implementation. An interface specifies:

- the names defined external to the unit that are to be **imported** for use within the implementation of the unit; and

- the names defined internal to the unit that are to be **exported** for use outside of the unit.

It is desirable to make such interfaces narrow — i.e., importing and exporting few names — to limit dependencies among program parts. Wide interfaces give rise to spaghetti-like dependencies among program units that are difficult for a programmer to keep track of. And the complexities are only exacerbated in the more common situation where a large program is developed in collaboration with others. In this case, wider interfaces imply increased communication and coordination between members of a programming team.

From this engineering perspective, a programming language should provide mechanisms that facilitate the construction of narrow interfaces. The simple approach of a single global namespace strikes out again because it allows every name to be used everywhere throughout a program. A crucial ingredient for narrow interfaces is some means of **name hiding**, whereby names purely local to the implementation of a unit are effectively hidden from the rest of a program.

In this section, we shall investigate techniques for **name control** that address the cognitive and engineering problems outlined above. Unlike our discussion of names up to this point, these issues are largely orthogonal to the choice of denotable values. Rather, they specify the relationship between patterns of name usage and the logical structure of variables in the program.

7.2.1 Hierarchical Scoping: Static and Dynamic

Recall the following terms from our study of variables in FLK:

- A **variable** is an entity that names a value.
- An **identifier** is a name for a variable. Distinct variables may be named by the same identifier.
- A **variable declaration** is a construct that introduces a variable.
- A **variable reference** is a construct that stands for the value of a variable.
- The **scope** of a variable declaration is the portion of the program text over which the declared variable may be referenced.

For example, in FL, variables are declared in `lambda`, `letrec`, and `let` expressions. All variable references are written as unadorned identifiers.

For a given language, it may or may not be possible to determine the scope of a given declaration without running the program. If the scope of a declaration can always be determined from the abstract syntax tree of a program, the scope

of the declaration is said to be **static** or **lexical**. In this case, the variable declaration associated with any variable reference is apparent from the lexical structure of the program. If the scope of the declaration depends on details of the run-time behavior of the program, the declaration is said to have **dynamic scope**. A language in which all declarations have static (dynamic) scope is said to be a **statically (dynamically) scoped** language.

Figure 7.9 summarizes the difference between static and dynamic scoping. We explain these in turn.

$$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Computation}$$

$$\mathcal{E}[\![\texttt{(proc } I\ E\texttt{)}]\!] = \lambda e_{proc}\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta\ .\ (\mathcal{E}[\![E]\!]\ [I:\delta]e_{proc}))))$$

$$\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e_{call}\,.\ (\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e_{call})\ (\lambda p\ .\ (p\ \ (\mathcal{E}[\![E_2]\!]\ e_{call}))))$$

Statically (Lexically) Scoped Procedures

$$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Environment} \to \mathit{Computation}$$

$$\mathcal{E}[\![\texttt{(proc } I\ E\texttt{)}]\!] = \lambda e_{proc}\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta\, e_{call}\ .\ (\mathcal{E}[\![E]\!]\ [I:\delta]e_{call}))))$$

$$\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e_{call}\,.(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e_{call})\ (\lambda p\ .\ (p\ \ (\mathcal{E}[\![E_2]\!]\ e_{call})\ \ e_{call})))$$

Dynamically Scoped Procedures

Figure 7.9: The essential semantics of statically and dynamically scoped CBN procedures (CBV is analogous). In a statically (lexically) scoped procedure, free identifiers appearing in a `proc` body are resolved relative to e_{proc}, the environment determined by the text enclosing the `proc` expression. In a dynamically scoped procedure, free identifiers appearing in a `proc` body are resolved relative to e_{call}, the environment determined by the dynamic chain of procedure calls in which the procedure is being called.

7.2.1.1 Static Scope

All of the languages we have studied so far (other than call-by-denotation FL) have been statically scoped. In fact, all of these languages support a particular discipline of static scoping known as **block structure**. In a block structured language, declarations can be nested arbitrarily, and every variable reference refers to the variable introduced by the nearest lexically enclosing variable declaration of that identifier. The nearest lexically enclosing declaration is found by starting at the identifier and walking up the abstract syntax tree until a declaration introducing the identifier name is found.

As an example, consider the CBV FL expression

```
(let ((x 20))
  ((let ((increment-by-x (lambda (y) (+ x y)))
         (double (lambda (x) (* x 2))))
     (letrec ((x (cons 1 x)))
       (lambda (z)
         (cons (double (increment-by-x (double z)))
               x))))
   (- x 15)))
```

In this expression there are three distinct variables named `x` introduced by three declarations:

1. `(let ((x 20)) ...)` declares `x` and binds it to the number 20.
2. The `(lambda (x) (* x 2))` expression named by `double` declares `x` but does not bind it; `x` will be bound on application of the procedural value of this abstraction. (In fact, the binding of `x` may be different for every distinct application of this procedure.)
3. `(letrec ((x (cons 1 x))) ...)` declares `x` and binds it to an infinite list of 1s.

There are also five variable references involving `x`:

1. In `(+ x y)`, `x` is a reference to the `let`-bound variable named `x`, so its meaning in this context is 20. This means that `increment-by-x` is a procedure that always adds 20 to its argument, regardless of the binding for `x` in whatever environment it happens to be applied.
2. In `(* x 2)`, `x` is a reference to the `lambda`-bound variable named `x`, whose meaning will be determined at application time.
3. In `(cons 1 x)`, `x` is a reference to the `letrec`-bound variable named `x`, so its meaning is an infinite list of 1s.

4. In `(cons (double (increment-by-x (double z))) x)`, `x` refers to the variable introduced by the first lexically enclosing declaration of `x`, which in this case is the `letrec`-bound variable. So here `x` is an infinite list of 1s as well.

5. In `(- x 15)`, `x` is a reference to the variable introduced by the first lexically enclosing declaration of `x`, which in this case is the `let`-bound variable. So here `x` means the number 20.

Putting together all of the above information, the value of the example expression is an infinite list whose first element is 60, and the rest of whose elements are all 1.

Variables in a block structured language have a structure reminiscent of variables in the lambda calculus.

As we noted before for FLK, when the scope of a declaration contains another declaration of the same name, the inner declaration carves out a **hole in the scope** of the outer one. The Stoy diagrams we used to represent the structure of lambda terms could easily be adapted to show declaration/reference relationships in any block structured language.

The essence of block structure is in the way environments are handled by abstractions. Figure 7.9 shows the domains and valuation functions that are crucial for block structure in CBN FLK. The clause for `proc` dictates that the body of the abstraction will be evaluated with respect to the environment in effect *when the procedure was created*. In particular, the environment in which the procedure is called can have no effect on the meaning of names within the abstraction body. This is clear from the domain definition for *Procedure*, which simply maps denotable values to computations and ignores whatever the current environment might be. Though the details of expressible and denotable values might differ under other parameter passing mechanisms, the handling of environments will have this form in any block structured language.

7.2.1.2 Dynamic Scope

SNOBOL4, APL, most early Lisp dialects, and many macro languages are dynamically scoped. In each of these languages, a free variable in a procedure (or macro) body gets its meaning from the environment at the point where the procedure is called rather than the environment at the point where the procedure is created. Thus, in these languages, it is not possible to determine a unique declaration corresponding to a given free variable reference; the effective declaration depends on where the procedure is called. It is therefore generally impossible to determine the scope of a declaration simply by considering the

abstract syntax tree of the program. Instead, the scope of a variable declaration depends on the run-time tree of procedure calls.

Figure 7.9 also shows essential semantics of dynamic scoping for a CBN language. The *Procedure* domain has been modified to indicate that procedures take an extra argument: the dynamic environment (i.e., the call-time environment). In the valuation clause for `proc`, the body of the abstraction is evaluated in the dynamic environment rather than the lexical one. The clause for `call` has been modified to pass the current environment to the procedure being called.

As an example of static vs. dynamic scoping, consider the following expression in CBV FL:

```
(let ((a 1))
  (let ((f (lambda (x) (primop + x a))))
    (let ((a 20))
      (f 300))))
```

Informally, we can reason as follows. The procedure named `f` refers to a free variable `a`. Under static scoping, this variable is bound to the value of `a` where the procedure is defined (i.e., `1`). Thus, the binding between `a` and `20` is irrelevant, and the result of the call `(f 300)` is `301`. On the other hand, under dynamic scoping, the free variable gets its value from whatever binding of `a` is dynamically apparent. In the call `(f 300)`, the binding between `a` and `20` shadows the binding between `a` and `1`, so the value of the call is `320`.

We can use the denotational definitions of scoping to formally analyze this example. The example FL expression desugars into the following FLK program:

```
(call (proc a                                ; E_proc:a1
        (call (proc f                        ; E_proc:f
                (call (proc a (call f 300)) ; E_proc:a20
                      20))
              (proc x (primop + x a))))      ; E_proc:x
      1)
```

The four `proc` expressions have been commented with names that will be used to abbreviate them. Figure 7.10 and 7.11 highlight the key steps for using the denotational definitions to derive the value of the expression under static scoping and dynamic scoping.

A more graphical perspective of these derivations appears in Figure 7.12. Each derivation is summarized by an **environment diagram** that shows key expressions along with the environments they are evaluated in. An environment is represented by a chain of bindings that go up the page; this helps to clarify the relationship between the different environments. The static scoping example is depicted in Figure 7.12(a). The arrow from within the procedural value to the

STATIC SCOPING

$\mathcal{E}[\![\texttt{(call } E_{proc:a1} \texttt{ 1)}]\!]\ e_0$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:a1}]\!]\ e_0)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 1))))$

$\mathcal{E}[\![\texttt{(call } E_{proc:f}\ E_{proc:x}\texttt{)}]\!]\ e_1$
where $e_1 = [\texttt{a:}\ (\textit{Int} \mapsto \textit{Value}\ 1)]\ e_0$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:f}]\!]\ e_1)$
$(\lambda p\ .\ (p\ (\textit{Procedure} \mapsto \textit{Value}\ (\lambda\delta\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e_1))))))$

$\mathcal{E}[\![\texttt{(call } E_{proc:a20}\texttt{ 20)}]\!]\ e_2$
where $e_2 = [\texttt{f:}\ (\textit{Procedure} \mapsto \textit{Value}$
$(\lambda\delta\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e_1))]e_1$

$(\textit{with-procedure}\ (\mathcal{E}[\![E_{proc:a20}]\!]\ e_2)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 20))))$

$\mathcal{E}[\![\texttt{(call f 300)}]\!]\ e_3$
where $e_3 = [\texttt{a} : (\textit{Int} \mapsto \textit{Value}\ 20)]e_2$

$(\textit{with-procedure}\ (\mathcal{E}[\![\texttt{f}]\!]\ e_3)\ (\lambda p\ .\ (p\ (\textit{Int} \mapsto \textit{Value}\ 300))))$

$((\lambda\delta\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e_1))\ (\textit{Int} \mapsto \textit{Value}\ 300))$

$\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : (\textit{Int} \mapsto \textit{Value}\ 300)]e_1$

$\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : (\textit{Int} \mapsto \textit{Value}\ 300)][\texttt{a} : (\textit{Int} \mapsto \textit{Value}\ 1)]e_0$

$= (\textit{Int} \mapsto \textit{Value}\ 301)$

Figure 7.10: `(call` $E_{proc:a1}$ `1)` evaluation using static scoping

DYNAMIC SCOPING

$\mathcal{E}[\![\texttt{(call } E_{proc:a1} \texttt{ 1)}]\!]\ e_0$

$(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_{proc:a1}]\!]\ e_0)\ (\lambda p\ .\ (p\ (\mathit{Int} \mapsto \mathit{Value}\ 1)\ e_0)))$

$\mathcal{E}[\![\texttt{(call } E_{proc:f}\ E_{proc:x}\texttt{)}]\!]\ e_1$
where $e_1 = [\texttt{a} : (\mathit{Int} \mapsto \mathit{Value}\ 1)]e_0$

$(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_{proc:f}]\!]\ e_1)$
$(\lambda p\ .\ (p\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta\, e'\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e')))$
$e_1))$

$\mathcal{E}[\![\texttt{(call } E_{proc:a20}\ \texttt{20)}]\!]\ e_2$
where $e_2 = [\texttt{f}: (\mathit{Procedure} \mapsto \mathit{Value}$
$(\lambda\delta\, e'\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e')))]e_1$

$(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_{proc:a20}]\!]\ e_2)\ (\lambda p\ .\ (p\ (\mathit{Int} \mapsto \mathit{Value}\ 20)\ e_2)))$

$\mathcal{E}[\![\texttt{(call f 300)}]\!]\ e_3$
where $e_3 = [\texttt{a} : (\mathit{Int} \mapsto \mathit{Value}\ 20)]e_2$

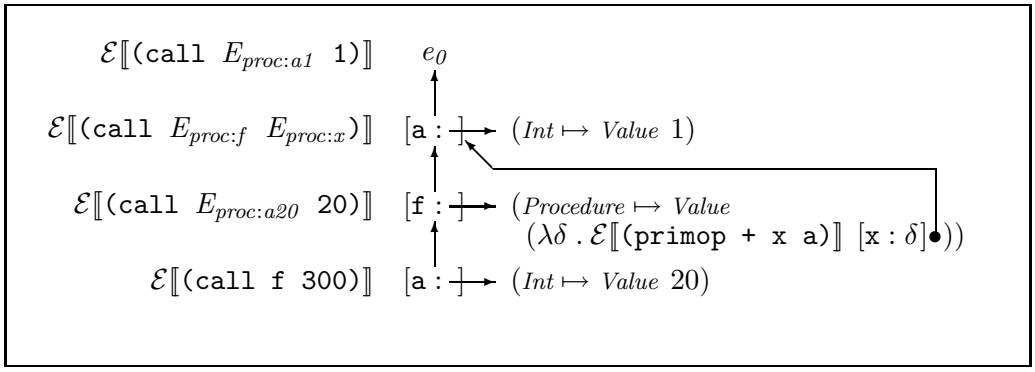
$(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![\texttt{f}]\!]\ e_3)\ (\lambda p\ .\ (p\ (\mathit{Int} \mapsto \mathit{Value}\ 300)\ e_3)))$

$((\lambda\delta\, e'\ .\ (\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\texttt{x} : \delta]e'))\ (\mathit{Int} \mapsto \mathit{Value}\ 300)\ e_3)$

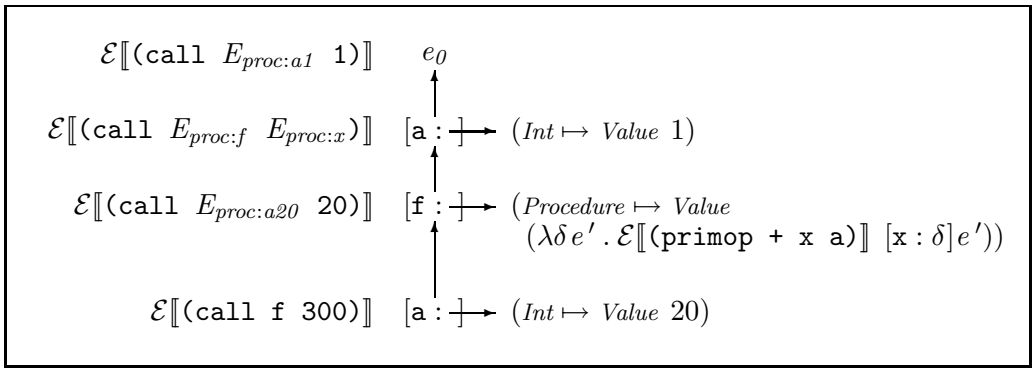
$\mathcal{E}[\![\texttt{(primop + x a)}]\!]\ [\text{x: } (\mathit{Int} \mapsto \mathit{Value}\ 300)][\text{a: } (\mathit{Int} \mapsto \mathit{Value}\ 20)]\ e_2$

$= (\mathit{Int} \mapsto \mathit{Value}\ 320)$

Figure 7.11: `(call` $E_{proc:a1}$ `1)` evaluation using dynamic scoping



(a) Environment diagram for the sample expression under static scoping.



(b) Environment diagram for the sample expression under dynamic scoping.

Figure 7.12: Environment diagrams illustrating the difference between statically and dynamically scoped procedures.

environment starting with [a : 1] emphasizes that a statically scoped procedure "remembers" the environment in which it was created. This **lexical environment** is determined by the text lexically surrounding the `proc` expression that gave rise to the procedure value.

The dynamic scoping example is depicted in Figure 7.12(b). Here, there is no arrow emanating from the procedural value because the environment e' in which the body is evaluated will be the **dynamic environment** in effect when the procedure is called. The dynamic environment is determined by the bindings in the current branch of the tree of procedure calls made during the execution of the program. In this example, it is constructed by the procedure calls associated with the three nested `let` expressions.

Although the environment chains happen to be the same for these two examples, lexically scoped languages tend to give rise to shallow, bushy environment diagrams, while dynamically scoped languages tend to give rise to deep thin ones (see Exercise 7.13).

Dynamic scoping seems rather odd. Is it useful? Yes! Dynamic scope is convenient for specifying the values of implicit parameters that are cumbersome to list explicitly as formal parameters to procedures. For example, consider the `derivative` procedure:

```
(define derivative
  (lambda (f x)
    (/ (- (f (+ x epsilon))
          (f x))
       epsilon)))
```

Note that `epsilon` appears as a free variable in `derivative`. With dynamic scoping, it is possible to dynamically specify the value of `epsilon` via any binding construct. For example, the expression

```
(let ((epsilon 0.001))
  (derivative (lambda (x) (* x x)) 5.0))
```

would evaluate `(derivative (lambda (x) (* x x)) 5.0)` in a context where `epsilon` is bound to `0.001`.

However, with lexical scoping, the variable `epsilon` must be defined at top level, and, without using mutation, there is no way to temporarily change the value of `epsilon` while the program is running. If we really want to abstract over `epsilon` with lexical scoping, we must pass it to `derivative` as an explicit argument:

```
(define derivative
  (lamdba (f x epsilon)
    (/ (- (f (+ x epsilon))
          (f x))
       epsilon)))
```

But then any procedure that uses `derivative` and wants to abstract over `epsilon` must also include `epsilon` as a formal parameter. In the case of `derivative`, this is only a small inconvenience. But in a system with a large number of tweakable parameters, the desire for fine-grained specification of variables like `epsilon` can lead to an explosion in the number of formal parameters throughout a program.

As an example along these lines, consider the huge parameter space of a typical window system (colors, fonts, stippling patterns, line thicknesses, etc.). It is untenable to specify each of these as a formal parameter to every window routine. At the very least, all these parameters need to be bundled up into a data structure that represents the graphics state. But then we still want a means of executing window routines in a temporary graphics state in such a way that the old graphics state is restored when the routines are done. Dynamic scoping is one technique for achieving this effect; side-effects are another.

Another typical use of dynamic scope is to specify error handling routines that are in effect during the execution of an expression. We shall see an example of this in Chapter 9.

Although dynamic scoping is good for allowing the specification of implicit parameters, it is seriously at odds with modularity, especially in a language that has first-class procedure values. We explore this issue in the exercises.

▷ **Exercise 7.11**

a. Can a dynamically scoped FL interpreter be written in statically scoped FL?

b. Can a statically scoped FL interpreter be written in dynamically scoped FL? ◁

▷ **Exercise 7.12** Write a single FL expression that exhibits a different behavior in each of the four following scenarios:

a. statically scoped CBN FL

b. statically scoped CBV FL

c. dynamically scoped CBN FL

d. dynamically scoped CBV FL ◁

▷ **Exercise 7.13** This problem considers a dynamically scoped variant of FL called FLUID. The abstract syntax for FLUID is the same as that for FL except that the

grammar for FLUID does not include any recursion constructs. That is, the FLUID kernel does not contain the `rec` construct (`rec` *I E*); and FLUID does not contain the `letrec` construct (`letrec ((`*I E*`)*)` *E*). The denotational semantics for FLUID is the same for that as FL except for the changes specified in Figure 7.9.

a. For each of the expressions below, show the result of evaluating the expression both in FL and in FLUID. Refer to the denotational semantics as necessary to reason about the evaluation process, but don't get lost in a symbol manipulation quagmire. You may find environment diagrams helpful for thinking about these problems.

i.
```
(let ((a 1))
  (let ((f (lambda (a) (primop + a 20))))
    (f a)))
```

ii.
```
(let ((a 1))
  (let ((f (lambda (a b) (primop + a b))))
    (f 20 300)))
```

iii.
```
(let ((a 1))
  (let ((a 20)
        (b 300))
    (primop + a b)))
```

iv.
```
(let ((a 1))
  (let ((f (lambda (b) (primop + a b))))
    (f (let ((a (f 20)))
         (f 300)))))
```

v.
```
(let ((a 1))
  (let ((f (lambda (b) (primop + a b))))
    (let ((g (lambda (a) (f a))))
      (g (g a)))))
```

b. In FLUID, the usual desugaring of multiple-argument abstractions into single argument abstractions no longer behaves as expected. Explain what goes wrong with the usual desugaring. (You do *not* need to describe how to fix the problem.)

c. In FLK, the factorial procedure is written as the expression:

```
(rec fact (proc n
            (if (primop = n 0)
                1
                (primop * n (fact (primop - n 1))))))
```

FLUID has no recursion constructs, but none are needed to write recursive definitions.

i. Briefly explain why the above claim is true.

ii. Show the definition for factorial procedure in FLUID.

iii. Explain why your FLUID definition for factorial wouldn't work in FL.

d. Consider the factorial procedure from part c. When using the denotational semantics to determine the meaning of `(call fact 3)` in environment e_0, the meaning of `(primop = n 0)` is determined in four distinct environments. For both CBV FL and for FLUID, draw an environment diagram that shows the relationship between these four environments. ◁

▷ **Exercise 7.14** Consider a version of FL called FLAT in which a procedure (a `lambda` or kernel `proc` expression) is not allowed to have free identifiers. Can the meaning of a FLAT expression differ under lexical and dynamic scope? If so, exhibit such an expression; if not, explain why. ◁

▷ **Exercise 7.15** Develop an operational semantics for CBV FL that uses explicit environments instead of substitution. ◁

▷ **Exercise 7.16** Develop an operational semantics for a dynamically scoped version of CBV FL. ◁

▷ **Exercise 7.17** The static scope expressed in Figure 7.9 is typical of block structured languages. However, other kinds of static scope are imaginable. For example, suppose that e_{global} is the top-level FL environment — the one that defines the meanings of all of the standard library names (e.g., `+`, `boolean?`, `cons`, etc.). Then **global scoping** is a static scoping mechanism in which free identifiers in a `proc` expression are resolved relative to e_{global} rather than the environment at the time of procedure creation or at the time of procedure call.

a. Write the valuation clauses for `proc` and `call` for a CBV variant of FL with global scoping.

b. Write a single FL expression whose value is a symbol (one of `global`, `block-structure`, or `dynamic`) indicating the scoping mechanism under which it is evaluated. ◁

▷ **Exercise 7.18** In this problem, we ask you to give a translation from dynamically scoped, call-by-value FLK to PostLisp, the language defined in Exercise 3.45. You are only required to translate a subset of FLK, defined by the following grammar:

E ::= U | I | `(proc` I E`)` | `(call` E_1 E_2`)` | `(primop / ` E_1 E_2`)`

Your translation should map every expression E_{FLK} of the subset to a sequence $Q_{PostLisp}$ of PostLisp commands such that:

$$E_{FLK} \xrightarrow{DCBV} U \quad \text{if and only if} \quad (Q_{PostLisp}) \xrightarrow{PostLisp} U,$$
$$E_{FLK} \xrightarrow{DCBV} error \quad \text{if and only if} \quad (Q_{PostLisp}) \xrightarrow{PostLisp} error, \text{ and}$$
$$E_{FLK} \xrightarrow{DCBV} \infty\text{-}loop \quad \text{if and only if} \quad (Q_{PostLisp}) \xrightarrow{PostLisp} \infty\text{-}loop,$$

where $\xrightarrow{DCBV}$ means dynamically scoped, call-by-value FLK evaluation. ◁

▷ **Exercise 7.19** Alyssa P. Hacker is asked by Analog Equipment Corporation to change their version of FL to be dynamically scoped in response to customer demand. Alyssa is asked to do this over a weekend, but she does not panic. Instead, she realizes that by implementing just a few new primitives, the entire job can be accomplished with clever desugaring.

More specifically, Alyssa added the following three new primitives:

- `(%new)`: Creates a new, empty environment.
- `(%extend` ENV `(symbol` I`)` V`)`: Returns a new environment equal to ENV, except that I is bound to V.
- `(%lookup` ENV `(symbol` I`))`: Returns the value of identifier I in environment ENV. It is is a fatal error if I is not bound in ENV.

In Alyssa's desugaring, the `*dynenv*` variable is always bound to the current dynamic environment. For this problem, consider only single argument procedures and calls. Here is Alyssa's desugaring rule for `call`:

$$\mathcal{D}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \texttt{(call } \mathcal{D}[\![E_1]\!] \texttt{ *dynenv* } \mathcal{D}[\![E_2]\!]\texttt{)}$$

a. What is the desugaring rule for I (variable reference)?

b. What is the desugaring rule for `(lambda (`I`)` E`)`?

c. What is the desugaring rule for `(let ((`I_1 E_1`) ... (`I_n E_n`))` E_{body}`)`?

d. Do all identifiers have to be looked up in the dynamic environment? If not, state what optimizations of the desugarings for identifiers and lambda are possible, and when and how they could be accomplished.

e. Desugar the following expression in this dynamically scoped version of FL:

```
(lambda (g) (call g x))
```

◁

7.2.2 Multiple Namespaces

Sometimes a single environment is not sufficient to model the naming features of a programming language. Languages commonly support **multiple namespaces** — i.e., several different contexts in which names are associated with values of various sorts. For example, Figure 7.13 shows a piece of COMMON LISP code in which the name `x` is used to name five different entities at the same time: an exit point, a special (dynamic) variable, a lexical variable, a procedure, and a tagbody tag.

```
(block x                              ; x1, name of exit point
  (let ((x 2))                        ; x2, declared to be
    (declare (special x))             ;   a special variable
    (let ((x 3))                      ; x3, normal lexical variable
      (flet ((x (y)                   ; x4, names a procedure that
               (+ x y)))              ;   refers to x3 in its body
        (tagbody
           x                          ; x5, a tagbody tag
           (if (> x 6)                ; this x = x3 = 3
               (go x)                 ; go to x5 if we get here
               (return-from
                 x                    ; return from exit point x1
                 (locally (declare (special x))
                                      ; Make 2nd x special below
                          (x x)       ; Apply procedure x4 to
                  ))))))))           ;  special x2
; The value of this expression is 5.
```

Figure 7.13: COMMON LISP code that uses multiple namespaces

There are two typical situations in which multiple namespaces are useful:

1. The language provides multiple scoping mechanisms. In this case, different namespaces can be used for different scoping mechanims. COMMON LISP, for example, supports both lexical and dynamic scoping of variables; variables are ordinarily scoped lexically, but those marked as `special` are dynamically scoped.

2. Different namespaces are used to name different kinds of entities. For example, exit points, tag labels, and procedures are in non-overlapping namespaces in Common Lisp. Namespaces used this way are especially useful for modeling values that are not first-class.

Of course, any language with multiple namespaces must provide methods

for both binding names and accessing names within each namespace. For example, consider the namespaces for exit points and tags in the Common Lisp example above. `block` introduces a name into the exit point namespace, and `return-from` accesses the exit point name, whereas `tagbody` introduces new tag names into the namespace of tags, and `go` can refer to these tags.

Multiple namespaces are modeled in denotational semantics by using multiple environments. For example, we could modify the semantics of FL to

- support both lexical and dynamic scoping;
- make procedures second-class objects.

These modifications are left as exercises.

▷ **Exercise 7.20** DYNALEX Understanding the virtues of both lexical and dynamic scoping, Sam Antix decides to design a language, DYNALEX, that supports both kinds of scoping mechanisms. The kernel of DYNALEX is statically scoped CBV FLK, extended with the following extra constructs to support dynamic scoping:

> `(dylambda (`I_{dyn}*`)` E_{body}`)` is like `lambda`, but binds the names I_{dyn}* in a dynamic environment rather than a static one.
>
> `(dyref` I`)` looks up I in the dynamic environment rather than the lexical one.

The full DYNALEX language includes the usual FL sugar as well as the following sugar for the `dylet` construct:

$$\mathcal{D}_{\text{exp}}[\![\texttt{(dylet ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}]\!] =$$
$$\texttt{((dylambda (}I_1\ \ldots\ I_n\texttt{)}\ \mathcal{D}_{\text{exp}}[\![E_{body}]\!]\texttt{)}\ \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \ldots\ \mathcal{D}_{\text{exp}}[\![E_n]\!]\texttt{)}$$

The following DYNALEX expression illustrates both dynamic and lexical scoping:

```
(let ((a 1) (b 20))
  (let ((f (lambda () (+ a (dyref b)))))
    (dylet ((a 300) (b 4000))
      (f)))) --DYNALEX--> 4001
```

a. Sketch a denotational semantics for DYNALEX that includes the the signature of $\mathcal{E}$, and the valuation clauses for the following constructs: I, `proc`, `call`, `dyref`, and `dylambda`.

b. Explain why Sam chose to make the multi-argument `dylambda` abstraction a kernel form rather than treating `dylambda` as sugar for a single argument abstraction for dynamic variables.

c. Write a set of translation rules for translating the DYNALEX kernel into FLK. ◁

7.2.3 Non-hierarchical Scope

7.2.3.1 Philosophy

The binding constructs we have seen so far are all **hierarchical** in nature. Each construct establishes a parent-child relationship between an outer context in which the declaration is not visible and an inner (body) context in which the declaration is visible. In static scoping, the hierarchy is determined by the abstract syntax tree, while in dynamic scoping, the hierarchy is determined by the tree of procedure calls generated at run-time. In both these scoping mechanisms, there is no natural way to communicate a declaration *laterally* across the tree-structure imposed by the hierarchy.

For small programs, this is not ordinarily a problem, but when a large program is broken into independent pieces, or **modules**, the constraint of hierarchy can be a problem. Modules connect and communicate with each other via collections of bindings; a module provides services by exporting a set of bindings and makes use of other modules' services by importing bindings from those other modules. In a hierarchical language, the scope of a binding is a single region of a program, so all the clients of a module must reside in the region where the module's bindings are in scope.

The traditional solution to the problem of communicating modules is to use a global namespace. All exported bindings from all modules are defined in a single environment, so all exported bindings are available to all modules. This technique is certainly widespread, but it has some major drawbacks:

- In order to avoid accidental name collisions, every module must be aware of all definitions made by all other modules, even those definitions that are completely irrelevant.

- In practice, the dependencies among modules are often poorly documented, making intermodule dependencies difficult to track.

A way for languages to overcome the hierarchical scoping of binding constructs is to provide a value with named subparts. For this purpose, we will introduce a new module value that bundles up a set of bindings at one point in a program and can communicate them to a point that is related neither lexically nor dynamically to the declarations of those bindings. Typically, a module defines a set of named values, especially procedures, that provide a particular function. For example, a matrix module might provide a set of matrix manipulation procedures like `matrix-invert` and `gaussian-elimination`. The modules described here are similar to PASCAL records and C structures.

We will study modules here in the context of a record package for FL. (Chapter 15 will explore a more complete module system.) Figure 7.14 lists new kernel forms for records.

`(record (`*I E*`)*)`	Create a record.
`(select` *I E*`)`	Select field *I* from record *E*.
`(override` E_1 E_2`)`	Append the named components of two records, giving precedence to names in E_2.
`(conceal (`*I*`*) E)`	Return a new record without specified fields.

Figure 7.14: Kernel record constructs.

`record` builds a data structure of name/value bindings. Values can be extracted by name using the `select` construct. Two records can be combined into a new record with `override`, which gives precedence to the bindings in the second record argument. `conceal` returns a new record in which some bindings of the original record have been removed. For example:

```
(define m1 (record
             (a (+ 2 3))
             (square (lambda (x) (* x x)))))

(select a m1) ⟶FL 5

((select square m1) (select a m1)) ⟶FL 25

(select b m1) ⟶FL error:no-such-record-field

(define m2 (record (a 7) (b 11)))

(select a (override m1 m2)) ⟶FL 7

(select a (override m1 (conceal (a) m2))) ⟶FL 5
```

Notice that there is a design choice in the semantics of `conceal`: the language designer must choose whether or not it is an error when `conceal` attempts to hide a name that is not actually a field in the record.

Figure 7.15 shows some convenient sugar constructs for records. Like many desugarings, these capture handy idioms programmers would invent on their own. `recordrec` allows the fields of a record to be mutually recursive. This is useful when records are used to construct separate program modules that contain procedures. `with-fields` provides a lexical scope that binds the specified names exported by a record, saving the programmer the tiresome task of writing `select`

`(recordrec (`I E`)*)`	A record with mutually recursive bindings.
`(with-fields (`I`*) `E_1 E_2`)`	Bind the I* to the corresponding fields in the record value E_1 and then evaluate E_2.
`(restrict (`I`*) `E`)`	Return a new record with only the specified fields. The dual of `conceal`.
`(rename ((`I_{old} I_{new}`)*) `E`)`	Rename I_{old} fields to I_{new} fields in E.

Figure 7.15: Record sugar constructs.

everywhere (or introducing the `let`s manually). Note that `with-fields` requires the list of identifiers to be bound. This allows the bindings in this lexical scope to be apparent,[4] which is necessary in a block structured language, and it also allows the programmer to avoid introducing unnecessary names. `restrict` is the natural dual to `conceal`, useful when exporting comparatively few names. `rename` helps programmers avoid name conflicts.
The desugarings for these constructs appear in Figure 7.16. Just as with `conceal`, the language designer must choose whether names not defined by a record generate errors. In a CBV language, the desugaring rules will generate an error when the undefined name is selected.

$$\mathcal{D}[\![\texttt{(recordrec (}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{))}]\!] =$$
$$\texttt{(letrec ((}I_1\ \mathcal{D}[\![E_1]\!]\texttt{) ... (}I_n\ \mathcal{D}[\![E_n]\!]\texttt{))}$$
$$\texttt{(record (}I_1\ I_1\texttt{) ... (}I_n\ I_n\texttt{)))}$$

$$\mathcal{D}[\![\texttt{(with-fields (}I_1\ \texttt{...}\ I_n\texttt{) }E_1\ E_2\texttt{)}]\!] =$$
$$\texttt{(let ((}I_{fresh}\ \mathcal{D}[\![E_1]\!]\texttt{))}$$
$$\texttt{(let ((}I_1\ \texttt{(select }I_1\ I_{fresh}\texttt{)) ... (}I_n\ \texttt{(select }I_n\ I_{fresh}\texttt{)))}$$
$$\mathcal{D}[\![E_2]\!]\texttt{))}$$

$$\mathcal{D}[\![\texttt{(restrict (}I_1\ \texttt{...}\ I_n\texttt{) }E\texttt{)}]\!] =$$
$$\texttt{(let ((}I_{fresh}\ \mathcal{D}[\![E]\!]\texttt{))}$$
$$\texttt{(record (}I_1\ \texttt{(select }I_1\ I_{fresh}\texttt{)) ... (}I_n\ \texttt{(select }I_n\ I_{fresh}\texttt{))))}$$

$$\mathcal{D}[\![\texttt{(rename ((}I_1\ I_1\texttt{') ... (}I_n\ I_n\texttt{')) }E\texttt{)}]\!] =$$
$$\texttt{(let ((}I_{fresh}\ \mathcal{D}[\![E]\!]\texttt{))}$$
$$\texttt{(override (conceal (}I_1\ \texttt{...}\ I_n\texttt{) }I_{fresh}\texttt{)}$$
$$\texttt{(record (}I_1\texttt{' (select }I_1\ I_{fresh}\texttt{)) ... (}I_n\texttt{' (select }I_n\ I_{fresh}\texttt{)))))}$$

Figure 7.16: Desugarings for the record syntactic sugar.

[4] We will see in Chapters ?? and 15 how a statically typed language could deduce this information.

7.2.3.2 Semantics

Figure 7.17 and 7.18 present a denotational semantics for records. Since both records and environments associate names and values, it is natural to model a record as an environment. You are encouraged to develop an operational semantics for the record constructs.

$$
\begin{array}{rcll}
v & \in & \mathit{Value} & = \ \ldots + \mathit{Record} \\
r & \in & \mathit{Record} & = \ \mathit{Environment}
\end{array}
$$

$\mathit{with\text{-}record} : \mathit{Computation} \to (\mathit{Record} \to \mathit{Computation}) \to \mathit{Computation}$
Defined like *with-boolean* and *with-procedure*.

$\mathit{extend\text{-}env}^* : \mathit{Environment} \to \text{Identifier}^* \to \mathit{Denotable}^* \to \mathit{Environment}$

$$
\begin{array}{l}
= \lambda e I^* \delta^* .\ \textbf{matching}\ \langle I^*,\ \delta^* \rangle \\
\qquad \triangleright\ \langle [],\ [] \rangle \ [\![\ e \\
\qquad \triangleright\ \langle I.I_{rest}{}^*,\ \delta.\delta_{rest}{}^* \rangle \ [\![\ (\mathit{extend}\ (\mathit{extend}^*\ e\ I_{rest}{}^*\ \delta_{rest}{}^*)\ I\ \delta) \\
\qquad \triangleright\ \textbf{else}\ (\mathit{err\text{-}to\text{-}comp}\ \texttt{no-such-record-field}) \\
\qquad \textbf{endmatching}
\end{array}
$$

$\mathit{combine\text{-}env} : \mathit{Environment} \to \mathit{Environment} \to \mathit{Environment}$

$$
\begin{array}{l}
= \lambda e_1\, e_2 .\ \lambda I .\ \textbf{matching}\ (\mathit{lookup}\ e_1\ I) \\
\qquad (\mathit{Denotable} \mapsto \mathit{Binding}\ \delta)\ [\![\ (\mathit{lookup}\ e_1\ I) \\
\qquad (\mathit{Unbound} \mapsto \mathit{Binding}\ \mathit{unbound})[\![\ (\mathit{lookup}\ e_2\ I) \\
\qquad \textbf{endmatching}
\end{array}
$$

Figure 7.17: Domains and auxiliary functions for a denotational semantics of the kernel record constructs in a CBV language.

7.2.3.3 Examples

Figure 7.19 presents two modules for arithmetic: one for integers and one for rational numbers.

The `gcd` routine is used by the `rats` module to remove common factors from the numerator and denominator. Note how `recordrec` (and not `record`) is used to create a recursive scope in which the `rat` constructor is visible to other procedures in the module. Here is a sample use of the two modules:

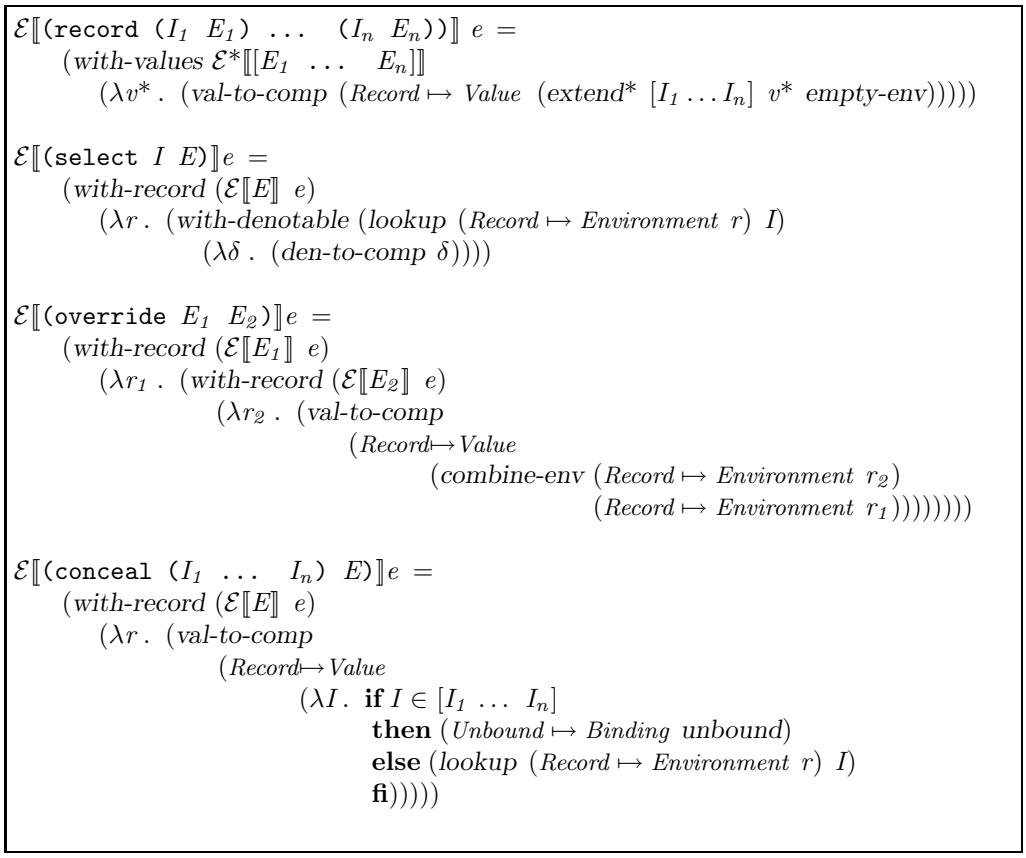


Figure 7.18: Valuation functions for the kernel record constructs in a CBV language.

```
(define ints (record (zero 0) (add +) (sub -) (mul *) (div quotient)
                     (neg (lambda (x) (- 0 x)))
                     (recip (lambda (x) (quotient 1 x)))
                     (eq =) (lt <) (gt >)))

(define gcd (lambda (a b) (if (= b 0)
                              a
                              (gcd b (rem a b)))))
(define rats
  (recordrec
    (rat (lambda (numer denom)
           (let ((common (gcd numer denom)))
             (pair (/ numer common) (/ denom common)))))
    (numer car)
    (denom cdr)
    (zero (rat 0 1))
    (add (lambda (r1 r2)
           (rat (+ (* (numer r1) (denom r2)) (* (denom r1) (numer r2)))
                (* (denom r1) (denom r2)))))
    (sub (lambda (r1 r2) (add r1 (neg r2))))
    (mul (lambda (r1 r2)
           (rat (* (numer r1) (numer r2))
                (* (denom r1) (denom r2)))))
    (div (lambda (r1 r2) (mul r1 (recip r2))))
    (neg (lambda (r) (rat (- 0 (numer r)) (denom r))))
    (recip (lambda (r) (rat (denom r) (numer r))))
    (eq (lambda (r1 r2)
          (and (= (numer r1) (numer r2))
               (= (denom r1) (denom r2)))))
    (lt (lambda (r1 r2) (< (* (numer r1) (denom r2))
                           (* (denom r1) (number r2)))))
    (gt (lambda (r1 r2) (lt r2 r1)))))
```

Figure 7.19: Two modules for arithmetic.

```
(define sum-of-squares
  (lambda (mod)
    (with-fields (add mul) mod
      (lambda (a b)
        (add (mul a a) (mul b b))))))

((sum-of-squares ints) 3 4) ⟶FL 25

(with-fields (rat) rats
  ((sum-of-squares rats) (rat 1 3) (rat 1 4))) ⟶FL ⟨25, 144⟩
```

As a meatier example of using these arithmetic modules, consider the matrix module generator in Figure 7.20. In this example, matrices are represented as lists of rows.[5] `make-matrix-module` takes a number n and an arithmetic module a and constructs a new module that implements $n \times n$ matrices whose components are manipulated by a. For example, if n is 3 and a is `rats`, then `make-matrix-module` returns a module of 3×3 matrices over the rational numbers. The input module a must supply a `zero` constant and binary `add` and `mul` procedures. The resulting module is a matrix module that also exports these names as matrix operations. This means it is possible to use $n \times n$ matrices as elements of another matrix. Figures 7.21–7.23 show some matrix examples that run on a CBV FL interpreter:

7.3 Object-Oriented Programming

Object-oriented programming has emerged as an extremely popular programming paradigm. Definitions of what constitutes object-oriented programming vary, but they typically involve state-based entities called **objects** that communicate via messages. The behavior of an object is defined by its **class**, which specifies the object's state variables and its responses to messages. Classes and objects can be organized into **inheritance hierarchies** that describe how the behavior of one object can be inherited from other objects or classes. Although we will not discuss issues of state until the next chapter, it is worthwhile to introduce object-oriented programming here because most of the issues involved in this paradigm are issues of naming, not issues of state.

We introduce a purely functional object-oriented kernel called HOOK (Humble Object-Oriented Kernel) and its associated full language, HOOPLA (Humble Object-Oriented Programming Language). Figure 7.24 presents an s-expression grammar for HOOK. Figure 7.25 gives the syntax of the syntactic sugar; the

[5] In practice, we would import the list routines from their own list module.

```
(define make-matrix-module
  (lambda (n element-module)
    (with-fields (elt-add elt-mul elt-zero)
      (rename ((add elt-add) (mul elt-mul) (zero elt-zero))
        element-module)
      (conceal (map map2 reduce make-list)
        (recordrec
          (zero (make-list n (make-list n elt-zero)))
          (add (lambda (m1 m2)
                 (map2 (lambda (row1 row2) (map2 elt-add row1 row2))
                       m1
                       m2)))
          (mul (lambda (m1 m2)
                 (map (lambda (row1)
                        (map (lambda (row2)
                               (reduce elt-add
                                       elt-zero
                                       (map2 elt-mul row1 row2)))
                             (transpose m2)))
                      m1)))
          (transpose (lambda (m) (if (null? (car m))
                                     nil
                                     (cons (map car m)
                                           (transpose (map cdr m))))))
          (map (lambda (f lst)
                 (if (null? lst)
                     nil
                     (cons (f (car lst)) (map f (cdr lst))))))
          (map2 (lambda (f lst1 lst2)
                  (if (or (null? lst1) (null? lst2))
                      nil
                      (cons (f (car lst1) (car lst2))
                            (map2 f (cdr lst1) (cdr lst2))))))
          (reduce (lambda (binop identity lst)
                    (if (null? lst)
                        identity
                        (binop (car lst)
                               (reduce binop identity (cdr lst))))))
          (make-list (lambda (n elt)
                       (if (= n 0)
                           nil
                           (cons elt (make-list (- n 1) elt))))))))))
```

Figure 7.20: A generator for NxN matrix modules.

```
(define 2x2-int-matrices (make-matrix-module 2 ints))

(define im1 '((1 2) (3 4)))
(define im2 '((2 3) (4 5)))

fl-CBV> ((select add 2x2-int-matrices) im1 im2)
(list (list 3 5) (list 7 9))

fl-CBV> ((select mul 2x2-int-matrices) im1 im2)
(list (list 10 13) (list 22 29))
```

Figure 7.21: 2×2 matrices of integers.

```
(define 2x2-rat-matrices (make-matrix-module 2 rats))

(define rm1 (with-fields (rat) rats
              (list (list (rat 1 4) (rat 2 4))
                    (list (rat 3 4) (rat 4 4)))))

(define rm2 (with-fields (rat) rats
             (list (list (rat 1 5) (rat 2 5))
                   (list (rat 3 5) (rat 4 5)))))

fl-CBV> ((select add 2x2-rat-matrices) rm1 rm2)
(list (list (pair 9 20) (pair 9 10))
      (list (pair 27 20) (pair 9 5)))

fl-CBV> ((select mul 2x2-rat-matrices) rm1 rm2)
(list (list (pair 7 20) (pair 1 2))
      (list (pair 3 4) (pair 11 10)))
```

Figure 7.22: 2×2 matrices of rationals.

```
(define 2x2-matrices-of-2x2-int-matrices
  (make-matrix-module 2 2x2-int-matrices))

(define im3 '((3 4) (5 6)))
(define im4 '((5 6) (7 8)))

(define imm1 (list (list im1 im2) (list im3 im4)))
(define imm2 (list (list im2 im3) (list im4 im1)))

fl-CBV> ((select add 2x2-matrices-of-2x2-int-matrices) imm1 imm2)
(list (list (list (list 3 5) (list 7 9))
            (list (list 5 7) (list 9 11)))
      (list (list (list 8 10) (list 12 14))
            (list (list 6 8) (list 10 12))))

fl-CBV>  ((select mul 2x2-matrices-of-2x2-int-matrices) imm1 imm2)
(list (list (list (list 41 49) (list 77 93))
            (list (list 24 32) (list 48 64)))
      (list (list (list 89 107) (list 125 151))
            (list (list 52 70) (list 76 102))))
```

Figure 7.23: 2×2 Matrices of matrices of integers.

desugarings themselves are defined in Figure 7.26.

Figure 7.27 contains some sample HOOPLA classes. For example, a point is created by sending `make` to the `point` class. The resulting point responds to the `x`, `y`, and `move` messages. The language does not support side-effects; `move` does not change the existing point but creates a new one. Every method has as its first formal parameter a **self variable** that names the object that originally received the message. This self variable is crucial for getting inheritance to work. The `turtle` class inherits behavior from both `point`s and `direction`s. Figure 7.28 shows some examples of interacting with a HOOPLA interpreter.

7.3.1 Semantics of HOOK

The inheritance structure in HOOK programs is reminiscent of records in FL. In fact, the similarity is so great that we will define the semantics of HOOK programs by compiling them into a version of the FL language extended with records. The key to this transformation is that objects are represented as records that bind message names to procedures that represent methods. A message send is then handled by simply looking up the method/procedure in the receiver record and applying it to the actual arguments.

$E \in$ Exp
$M \in$ Message $=$ Identifier
$I \in$ Identifier
$L \in$ Lit $=$ Intlit + Boollit + ...

$E ::= L$		[Literal]
| I		[Identifier]
| `(method` $M_{message}$ `(`I_{self} I_{formal}`*)` E_{body}`)`		[Simple Object]
| `(object-compose` E_{obj1} E_{obj2}`)`		[Object Composition]
| `(null-object)`		[Null Object]
| `(send` $M_{message}$ $E_{receiver}$ E_{arg}`*)`		[Message Send]

Figure 7.24: The abstract syntax for HOOK.

$P \in$ Program
$D \in$ Def

$P ::=$ `(program` E_{body} D_{def}`*)`	[Program]
$D ::=$ `(define` I_{name} E_{value}`)`	[Definition]
$E ::= \ldots$	
| `(object` E_{object}`*)`	[Object]
| `(class (`I_{init}`*)` $E_{instance}$`*)`	[Class]
| `(lambda (`I_{formal}`*)` E_{body}`)`	[Abstraction]
| `(`E_{rator} E_{rand}`*)`	[Application]
| `(let ((`I_{name} E_{value}`)*)` E_{body}`)`	[Local-Binding]
| `(if` E_{test} $E_{consequent}$ $E_{alternative}$`)`	[Branch]

Figure 7.25: The syntax of the HOOPLA language.

$\mathcal{D}[\![\texttt{(object)}]\!] = \texttt{(null-object)}$
$\mathcal{D}[\![\texttt{(object } E_{fst}\ E_{rest}\texttt{*)}]\!] = \texttt{(object-compose } \mathcal{D}[\![E_{fst}]\!]\ \mathcal{D}[\![\texttt{(object } E_{rest}\texttt{*)}]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(class (}I_{init}\texttt{*) } E_{inst_1}\ \ldots\ E_{inst_n}\texttt{)}]\!] =$
$\texttt{(method make (}I_{ignore}\ I_{init}\texttt{*) } \mathcal{D}[\![\texttt{(object } E_{inst_1}\ \ldots\ E_{inst_n}\texttt{)}]\!]\texttt{)}$
where $I_{ignore} \notin \bigcup_{i=1}^{n} \mathit{FreeIds}[\![E_{inst_i}]\!]$

$\mathcal{D}[\![\texttt{(lambda (}I_{formal}\texttt{*) } E_{body}\texttt{)}]\!] = \texttt{(method call (}I_{ignore}\ I_{formal}\texttt{*) } \mathcal{D}[\![E_{body}]\!]\texttt{)}$
where $I_{ignore} \notin \mathit{FreeIds}[\![E_{body}]\!]$

$\mathcal{D}[\![\texttt{(}E_{rator}\ E_{rand}\texttt{*)}]\!] = \texttt{(send call } \mathcal{D}[\![E_{rator}]\!]\ \mathcal{D}[\![E_{rand}\texttt{*}]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(let ((}I_{var}\ E_{val}\texttt{)*) } E_{body}\texttt{)}]\!] = \mathcal{D}[\![\texttt{((lambda (}I_{var}\texttt{*) } E_{body}\texttt{) } E_{val}\texttt{*)}]\!]$

$\mathcal{D}[\![\texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!] = \texttt{(send if-true } \mathcal{D}[\![E_{test}]\!]$
$\mathcal{D}[\![\texttt{(lambda () } E_{con}\texttt{)}]\!]$
$\mathcal{D}[\![\texttt{(lambda () } E_{alt}\texttt{)}]\!]\texttt{)}$

Figure 7.26: Rules for desugaring HOOPLA into HOOK.

We formally define the transformation from HOOK code to CBV FL code in terms of the compilation function $\mathcal{T} : \mathrm{Exp}_{HOOK} \to \mathrm{Exp}_{FL}$. This function is defined in Figure 7.29. To be complete, we also would need functions that map HOOK programs to FL programs and HOOK definitions to FL definitions, but since these are straightforward, we will leave them out.

The core of the compilation is the handling of methods, objects, and message sends. A HOOK `method` construct is transformed into an FL `record` construct with a single binding of the message name to a procedure that does the work of the method. A HOOK `object-compose` construct compiles into an FL `override` construct; the semantics of `override` are such that methods from E_{obj1} will take precedence over methods from E_{obj2}. A HOOK message send compiles to a procedure application in FL; the procedure is found by looking up the message name in the record that represents the receiver.

The handling of literals (via $\mathcal{T}_{lit}$) is perhaps the trickiest part of the compilation. HOOK literals stand not for simple values but for full-fledged message-passing objects. A HOOK number object, for instance, must compile into an FL record that has methods for all the numeric operations. In addition, such a record must also be able to supply the unadorned version of the value it is holding onto; this is the purpose of the binding involving I_{int}. The identifier I_{int} must be the same for all literal objects. Note that operations returning the

```
(define point
  (class (init-x init-y)
    (method x (self) init-x)
    (method y (self) init-y)
    (method move (self dx dy)
      (object (send make point
                    (send + (send x self) dx)
                    (send + (send y self) dy))
              self)                          ; Allows mixins
      )))

(define direction
  (class (init-angle)
    (method angle (self) init-angle)
    (method turn (self delta)
      (object (send make direction
                    (send + (send angle self) delta))
              self))                         ; Allows mixins
    ))

(define turtle
  (class (x y angle)
    (method home (self)
      (object (send make turtle x y angle)
              self))                         ; Allows mixins
    (send make point x y)
    (send make direction angle)))

(define color
  (class (clr)
    (method color (self) clr)
    (method new-color (self new)
      (object (send make color new)
              self))))

(define colored-point
  (class (x y col)
    (send make point x y)
    (send make color col)))
```

Figure 7.27: Sample HOOPLA classes.

```
;;; Define a turtle T1
(define t1 (send make turtle 0 0 0))
(send x t1) ⟶HOOPLA 0    {This is the object 0}
(send y t1) ⟶HOOPLA 0
(send angle t1) ⟶HOOPLA 0

;;; Define T2 as a rotated and translated version of T1.
(define t2 (send move (send turn t1 45) 17 23))
(send x t2) ⟶HOOPLA 17
(send y t2) ⟶HOOPLA 23
(send angle t2) ⟶HOOPLA 45)

;;; Note that T1 is unchanged. E.g.:
(send x t1) ⟶HOOPLA 0

;;; Now define T3 as a version of T2 sent home.
(define t3 (send home t2))
(send x t3) ⟶HOOPLA 0
(send y t3) ⟶HOOPLA 0
(send angle t3) ⟶HOOPLA 0
```

Figure 7.28: Example interactions with a HOOPLA interpreter.

same kind of object being defined (e.g., `+`, `*`) return an extended version of `self` rather than just a fresh instance of the object. This means that the returned object retains all the behavior of the receiver that is not explicitly specified by the definition.

Booleans, symbols, and whatever other literals or standard identifiers we might support are handled like integers.

▷ **Exercise 7.21** What is the value of the following HOOPLA expression?

```
(let ((ob1 (object (method value (self) 1)))
      (ob2 (object (method value (self) 2)))
      (ob3 (object (method value (self) 3)
                   (method evaluate (self) (send value self)))))
  (send evaluate (object ob1 ob2 ob3)))
```
◁

▷ **Exercise 7.22** Following the example for integer literals, show how boolean literals in HOOK compile into FL. HOOK boolean objects handle the messages `not?`, `and?`, `or?`, `if-true`, and `if-false`. ◁

$\mathcal{T} : \text{Exp}_{HOOK} \rightarrow \text{Exp}_{FL}$
$\mathcal{T}_{lit} : \text{Lit}_{HOOK} \rightarrow \text{Exp}_{FL}$

$\mathcal{T}[\![I]\!] = I$

$\mathcal{T}[\![L]\!] = \mathcal{T}_{lit}[\![L]\!]$, where $\mathcal{T}_{lit}$ is described below.

$\mathcal{T}[\![$`(method` $M_{message}$ `(`I_{self} I_{formal}*`)` E_{body}`)`$]\!] =$
`(record (`$M_{message}$ `(lambda (`I_{self} I_{formal}*`)` $\mathcal{T}[\![E_{body}]\!]$`))))`

$\mathcal{T}[\![$`(object-compose` E_{obj_1} E_{obj_2}`)`$]\!] =$`(override` $\mathcal{T}[\![E_{obj_2}]\!]$ $\mathcal{T}[\![E_{obj_1}]\!]$`)`

$\mathcal{T}[\![$`(null-object)`$]\!] =$`(record)`

$\mathcal{T}[\![$`(send` $M_{message}$ $E_{receiver}$ E_{arg_1} $\ldots E_{arg_n}$`)`$]\!] =$
`(let ((`$I_{receiver}$ $\mathcal{T}[\![E_{receiver}]\!]$`))`
`((select` $M_{message}$ $I_{receiver}$`)` $I_{receiver}$ $\mathcal{T}[\![E_{arg_1}]\!]$ $\ldots\mathcal{T}[\![E_{arg_n}]\!]$`))`
where $I_{receiver} \notin \bigcup_{i=1}^{n} FreeIds[\![E_{arg_i}]\!]$

$\mathcal{T}_{lit}[\![N]\!] =$
```
(letrec ((make-integer
          (lambda (n)
            (record
              (I_int n)
              (+ (lambda (self arg)
                   (override self
                     (make-integer
                       (primop + n (select I_int arg))))))
              (* (lambda (self arg)
                   (override self
                     (make-integer
                       (primop * n (select I_int arg))))))
                 ⋮
              ))))
   (make-integer N))
```
where I_{int} is the same for all integers but distinct from all other message names.

Similarly for other literals.

Figure 7.29: The rules for translating HOOK to FL.

▷ **Exercise 7.23** Anoop Hacker is confused about namespace issues in HOOPLA. In the syntax of the full language, there are several binding constructs: `class`, `lambda`, `let`, and `method`. The first three constructs all bind formal parameters; the last one binds a message name and a name for *self* in addition to the formal parameters of the method. You have volunteered to help Anoop answer the following questions. Carefully study the definitions of HOOPLA to HOOK desugaring and HOOK to FL translation to justify your answers. Give examples where appropriate.

a. How many distinct namespaces are there in HOOPLA?

b. Is it possible for a method formal parameter named `x` to be shadowed by a message named `x`?

c. Is it possible for a message named `x` to be shadowed by a method formal parameter named `x`?

d. Do the answers to parts b and c change if the `record` in the translation for `object-compose` becomes a `recordrec` instead? If so, how? ◁

▷ **Exercise 7.24** Does HOOPLA's `lambda` construct support currying? Explain. ◁

▷ **Exercise 7.25** Paula Morwicz doesn't like the fact that it's always necessary to explicitly name *self* within a HOOPLA method. She decides to implement a version of HOOPLA called SELFISH in which a reserved word `self` is implicitly bound within every method body. For example, in SELFISH the point class would be written as follows:

```
(define point
  (class (init-x init-y)
    (method x () init-x)
    (method y () init-y)
    (method move (dx dy)
      (object (send make point
                    (send + (send x self) dx)
                    (send + (send y self) dy))
              self)                          ; Allows mixins
      )))
```

In this example, the instances of `self` within the `move` method evaluate to the receiver of the `move` message. Because `self` is a reserved word in SELFISH, it is illegal to use it as a formal parameter to a method.

a. Describe what modifications would have to be made to the following in order to specify the semantics of SELFISH:

   i. The HOOK grammar.

   ii. The HOOPLA to HOOK desugarer.

   iii. The HOOK to FL translator.

b. Unfortunately, SELFISH doesn't always give the behavior Paula expects. For example, she makes a simple modification to the definition of the point class:

```
(define point
  (class (init-x init-y)
    (method x () init-x)
    (method y () init-y)
    (method move (dx dy)
      (let ((new-x (send + (send x self) dx))
            (new-y (send + (send y self) dy)))
        (object (method x () new-x)
                (method y () new-y)
                self)
        ))))
```

After this change, turtle objects (which are implemented in terms of points) no longer work as expected. Explain what has gone wrong.

c. Show how to get Paula's new point definition to work as expected. You can add new code, but not remove any. You may perform alpha-renaming where necessary.

d. Which do you think is better: the explicit `self` approach of HOOPLA or the implicit `self` approach of SELFISH? Explain your answer. ◁

Chapter 8

State

Man's yesterday may ne'er be like his morrow;
Nought may endure but Mutability

— Mutability, st. 4, Percy Bysshe Shelley

I woke up one morning and looked around the room. Something wasn't right. I realized that someone had broken in the night before and replaced everything in my apartment with an exact replica. I couldn't believe it...I got my roommate and showed him. I said, "Look at this — everything's been replaced with an exact replica!" He said, "Do I know you?"

— Steven Wright

8.1 What is State?

8.1.1 Time, State, Identity, and Change

We naturally view the world around us in terms of objects. Each object is characterized by a set of attributes that can vary with time. The **state** of an object is the set of particular attributes it has at a given point in time. For example, the state of a box of chocolates includes its size, shape, color, location, whether its lid is on or off, and the number, types, and positions of the chocolates inside.

Every object has a unique, time-independent attribute that distinguishes it from other objects: its **identity**. The notion of identity is at the very heart of

objectness, for it formalizes the intuition that objects exist over extents of time rather than just at instants of time. Identity allows us to say that an object at one point in time is the "same" as that at another point, regardless of any changes of state that may have taken place in between. It also gives us a way of saying that two objects with otherwise indistinguishable states are "different."

Consider our box of chocolates again. If we open the lid, the state of the box has changed, but we still consider it to be the same box of chocolates. Even after we eat all the goodies inside, we think that the box has become empty, not that we have a different box of chocolates.

On the other hand, suppose we leave an unopened box of chocolates on the kitchen table one day and find an unopened box there the next day. We are likely to assume that it's the same box. However, a housemate might later confess to consuming the entire original box in a fit of the munchies, but then buying a replacement box after feeling pangs of guilt. In light of this confession, we concede that the box on the table is not the same as the one we bought, even though, from our perspective, its state is indistinguishable from that of the box we left there the day before.

How could we monitor similar situations in the future without the help of explicit confessions? Before placing an unopened box of chocolates on the table we could alter the box in some irreversible way. The next day we could check if the box on the table had the same alteration. If the box on the table the next day does not exhibit the alteration, we are sure that the new box is not the same as the original. If it does have the alteration, we aren't 100% sure (our housemate might have diabolically copied our alteration, or a new box by chance might exhibit the same alteration), but there is reasonable evidence that the box is in fact the same one we left the previous day.

This example emphasizes that the notions of time, state, identity, and change are all inextricably intertwined.[1] The purpose of this chapter is to see how these notions are expressed in a computational framework. We shall see that state and its friends provide new ways to decompose problems but can greatly complicate reasoning about programs.

8.1.2 FL Does Not Support State

Computing with time-varying state-based entities is an extremely popular programming paradigm, both in traditional **imperative languages**, such as FORTRAN, COBOL, PASCAL, C, and ADAas well as in **object-oriented languages** like SMALLTALK, C++, C#, and JAVA. We shall call such languages **stateful**.

[1] For a further discussion of this philosophical point in a computational framework, see Chapter 3 of [ASS96].

One reason that stateful languages are so popular is that they resonate with the experience that many programmers have in interacting with objects that change over time in the world. At the opposite side of the spectrum are **stateless** languages like the so-called **purely functional** programming languages such as HASKELL and MIRANDA. **Mostly functional** languages are those, like COMMON LISP, SCHEME, and ML, that add stateful features on top of a stateless function-oriented core.

The FL language we have studied thus far is a stateless language – it provides no support for expressing computational objects with identity and state. In particular, neither variables nor data structures (pairs) may exhibit time-dependent behavior. To underscore this point, we will show the difficulties encountered in modeling a classic example of state – bank accounts – within FL. The goal is to implement the following bank account procedures in FL:

- `(make-account` *amount*`)`: Creates an account with *amount* as the initial balance.
- `(balance` *account*`)`: Returns the balance in *account*.
- `(deposit!` *amount account*`)`: If *amount* is non-negative, increases the balance of *account* by *amount* and returns the symbol `succeeded`. If *amount* is negative, leaves the balance unchanged and returns the symbol `failed`.
- `(withdraw!` *amount account*`)`: If *amount* is less than or equal to the balance of *account*, decreases the balance of *account* by *amount*, and returns the symbol `succeeded`. If *amount* is negative or is greater than the balance of *account*, leaves the balance unchanged and returns the symbol `failed`.

We adopt the convention that names of procedures that change the state of an object (such as `deposit!` and `withdraw!`) end in the '`!`' character (pronounced "bang").

Note that the specifications of `deposit!` and `withdraw!` indicate not only what **value** the procedures return (in both cases, one of the symbols `succeeded` or `failed`) but also what **effect** the procedure has on the state of the account (increasing or decreasing the balance). Even `make-account` has the effect of updating the banking system to include a new account. Such changes in state are referred to as **side effects** or **mutations**. In programming languages supporting state, the specification of a procedure includes both its return value and its side effects.[2]

[2]This is true for languages like SCHEME and C in which procedure calls are *expressions* —

It turns out that it is impossible to write a set of FL procedures that satisfy the above specifications. We will demonstrate this fact by studying a nullary (i.e., zero-argument) procedure `test-deposit!` that performs the following steps in order:

- create an account *acct* with a balance of 100;
- determine the balance *bal* of *acct*;
- deposit 17 dollars into *acct*;
- determine the new balance *bal′* of *acct*;
- return the difference *bal′* − *bal*.

In a stateful language, `(test-deposit!)` should return 17. However, we can show that in FL `test-deposit!` must return 0!

If we try to write `test-deposit!` in FL, we immediately run into a stumbling block. The specified actions are clearly ordered by time, but FL provides no explicit construct for specifying that expressions should be evaluated in any particular order. To get around this problem, we assume the existence of a construct `(begin` E_1 E_2`)` that evaluates E_1 before E_2. Since all FL expressions must return a value, we dictate that the value returned by a `begin` expression is the value of E_2. The formal semantics of `begin` are specified by the operational rewrite rules and the denotational valuation clause in Figure 8.1.

Operational Semantics

$$\texttt{(begin}\ V\ E\texttt{)} \Rightarrow E \qquad [\textit{begin-return}]$$

$$\frac{E_1 \Rightarrow E_1'}{\texttt{(begin}\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(begin}\ E_1'\ E_2\texttt{)}} \qquad [\textit{begin-progress}]$$

Denotational Semantics

$$\mathcal{E}[\![\texttt{(begin}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,(\textit{with-value}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda v\,.\,(\mathcal{E}[\![E_2]\!]\ e)))$$

Figure 8.1: Operational and denotational semantics of `begin`.

Using `begin`, we can write `test-deposit!` in FL as follows:

constructs that appear in value-accepting contexts. But in many languages, procedure calls are *commands* — constructs that do not produce values but are executed for effect only. In such languages, procedure specifications do not describe a return value.

```
(define test-deposit!
  (lambda ()
    (let ((acct (make-account 100)))
      (let ((old (balance acct)))
        (begin (deposit! 17 acct)
               (- (balance acct) old))))))
```

The abstraction can be translated into FLK as follows:

```
(proc ignore
  (call (proc acct
          (call (proc old
                  (begin (call (call deposit! 17) acct)
                         (primop - (call balance acct) old)))
                (call balance acct)))
        (call make-account 100)))
```

We can now use our semantics frameworks to show that the FLK call

```
(call test-deposit! #u)
```

must evaluate to 0 regardless of how `deposit!` is defined. We will assume a CBN version of FLK; since termination is not an issue here, the result will be the same for CBV.

An operational trace of `(call test-deposit! #u)` appears in Figure 8.2. Note the three copies of the expression `(call make-account 100)` generated by substitution. In FL's operational semantics, an expression representing a data structure for all intents and purposes *is* the data structure. Since the second operand of `deposit!` and the operand of the two calls of `balance` are syntactically distinct copies of the `make-account` expression, any operations performed by `deposit!` can't possibly affect the operands of the `balance` calls. If we make the assumption that

$$\texttt{(call balance (call make-account 100))} \stackrel{*}{\Rightarrow} \texttt{100}$$

(this would seem to be required of any reasonable bank account implementation), then the trace shows that `(test-deposit!)` indeed evaluates to 0.

Denotational semantics offers another perspective on this example. Recall that FL's valuation function $\mathcal{E}$ maps expressions and environments to expressible values. If an environment $e_{context}$ is specified, then a given expression must always denote the same meaning relative to $e_{context}$ because $\mathcal{E}$ is a mathematical function. Suppose that e_1 is an environment in which `acct` is bound to a representation of an account with a balance of b dollars (b need not be 100). Then the following must be true:

$$(\mathcal{E}[\![\texttt{(call balance acct)}]\!]\ e_1) = b$$

Now consider the meaning of the following expression E_{core} with respect to e_1:

```
(call (proc ignore
        (call (proc acct
                (call (proc old
                        (begin (call (call deposit! 17) acct)
                               (primop - (call balance acct) old)))
                      (call balance acct)))
              (call make-account 100)))
      #u)
⇒ (call (proc acct
            (call (proc old
                    (begin (call (call deposit! 17) acct)
                           (primop - (call balance acct) old)))
                  (call balance acct)))
          (call make-account 100))
⇒ (call (proc old
            (begin (call (call deposit! 17) (call make-account 100))
                   (primop - (call balance (call make-account 100))
                             old)))
          (call balance (call make-account 100)))
⇒ (begin (call (call deposit! 17) (call make-account 100))
           (primop -
                   (call balance (call make-account 100))
                   (call balance (call make-account 100))))
⇒* (primop -
            (call balance (call make-account 100))
            (call balance (call make-account 100)))
⇒* (primop - 100 100)
⇒ 0
```

Figure 8.2: Operational trace showing that `(call test-deposit! #u)` evaluates to 0.

```
E_core = (call (proc old
                  (begin (call (call deposit! 17) acct)
                         (primop - (call balance acct) old)))
               (call balance acct))
```

This expression contains two occurrences of `(call balance acct)` that are evaluated in environments containing the same binding for `acct`. So both of these occurrences denote the same number b. But then the meaning of `old` is clearly b, so the meaning of

```
(primop - (call balance acct) old)
```

must be 0. Thus, we have shown that E_{core} denotes 0, regardless of which account is denoted by `acct`. So `(test-deposit!)` must also denote 0.

In both the operational and denotational analyses, the fundamental insight is that `(test-deposit!)` returns the difference of two occurrences of the expression `(call balance acct)`, and these must necessarily have the same value. A language in which distinct occurrences of any expression always have the same meaning within a given naming context is said to be **referentially transparent**. Of course, the notion of "naming context" needs to be fully specified. Intuitively, two occurrences of an expression are in the same naming context if they share the same Stoy diagram — i.e., if every occurrence of a free identifier in one refers to the same binding occurrence as the corresponding identifier in the other. Stateless languages, such as our mini-language FL and the real language HASKELL, are referentially transparent, while stateful languages are not.

Referential transparency is a property that we frequently use in mathematical reasoning in the form of "substituting equals for equals." But it is seriously at odds with the notions of state and time. State is predicated on the idea that observable properties of an object can change. But if we make the reasonable assumption that a property of an object can be accessed by applying a single-argument procedure to that object (as in `(balance acct)` above), referential transparency dictates that all occurrences of such an expression within a given environment must denote the same value. Thus, the observable properties of an object cannot change. And if changes to the state of objects cannot be observed, how meaningful is it to talk about one action happening before or after another? We shall have more to say about referential transparency and state in Section 8.2.5.

Finally, suppose we actually try to write the definition of `deposit!` in FL. What kind of difficulties do we run into? Below is a skeleton for such a procedure:

```
(define deposit!
  (lambda (amount account)
    (if (< amount 0)
        'failed
        (begin E_IncreaseBalance
               'succeeded))))
```

The body of `deposit!` returns the right value (one of the symbols `failed` or `succeeded`). But how do we write $E_{IncreaseBalance}$? By the same reasoning used above, no FL expression can possibly alter the state of the account. Obviously, we are missing something. Shortly, we will introduce constructs that allow us to fill in the blanks here, and we will explore how the semantics of FL needs to be changed to accommodate their introduction.

But before we do that, consider the following. Since FL is a universal language, it is capable of expressing *any* computation. So surely examples such as the bank account scenario must be expressible within FL, albeit not necessarily in a way that corresponds to our intuitions about the physical world. Next, we'll examine some ways in which state can be simulated in FL. The purpose of this exploration is to give us insight into the nature of state. Later, we will be able to apply what we learn to the semantics for our modified dialect of FL.

8.1.3 Simulating State In FL

8.1.3.1 Iteration

The simulation of state in FL is exemplified by the handling of iteration. An **iteration** is a computation that characterizes the state of a system in terms of the values of a set of variables known as its **state variables**. The value of each state variable in an iteration at time t is a function of the values of the state variables at time $t - 1$.

As an example of an iteration, consider the problem of reversing the order of cards in a deck of playing cards. A natural solution is to use two piles, called *old* and *new*, where *old* is initially the original deck and *new* is an empty pile. Then, one by one, cards can be moved from the old pile to the new pile until the old pile is empty. At this point, the new pile contains the reversed deck of cards. In this example the state variables are the (ordered) contents of the old and new piles. These two variables completely characterize the state of the system. If a person performing the reversal for some reason had to leave before completing the task, someone else could take over as long as it was apparent which was the old pile and which was the new.

It is straightforward to express iterations in FL. For example, the above

technique can be applied to list reversal as follows:

```
(define reverse
  (lambda (lst)
    (letrec ((iterate
              (lambda (old-pile new-pile)
                (if (null? old-pile)
                    new-pile
                    (iterate (cdr old-pile)
                             (cons (car old-pile) new-pile)))))))
      (iterate lst '()))))
```

In this case, the state variables are the arguments `old-pile` and `new-pile` to the internal procedure `iterate`. For example, here is a trace of the reversal of a three-element list (where *REVERSE* and *ITERATE* stand for the appropriate expressions):

```
(REVERSE (list 1 2 3))
⇒* (ITERATE (list 1 2 3) (list)       )
⇒* (ITERATE (list 2 3)   (list 1)     )
⇒* (ITERATE (list 3)     (list 2 1)   )
⇒* (ITERATE (list)       (list 3 2 1) )
⇒* (list 3 2 1)
```

The above example suggests a general approach for expressing iterations in FL. State variables simply become the arguments to an iterating procedure, and updating the state variables is expressed by calling the iterating procedure on values computed from the previous values of the state variables.

Note carefully how an iteration manages to circumvent the constraints of referential transparency to represent state and time. The state at any point in time is represented by the values of formal parameter names associated with a particular application of the iterating procedure. In the list reversal example, the state variables correspond to the formal parameters `old-pile` and `new-pile`. The value of a particular variable named `old-pile` or `new-pile` never changes. However, each application of the `iterate` procedure effectively creates new variables that happen to be named by these same identifiers. So for each point in time t, there are distinct variables $\texttt{old-pile}_t$ and $\texttt{new-pile}_t$. State is encoded not as the changing value of a variable, but rather as the values of a sequence of immutable variables.

Events in time are ordered by the only means available for ordering in a stateless language: data dependency. If the value of E_1 is needed to compute E_2, then E_2 is said to have a **data dependency** on E_1. In the list reversal example, since $\texttt{old-pile}_t$ is equal to $\texttt{(cdr old-pile}_{t-1}\texttt{)}$, it has a data dependency on

`old-pile`$_{t-1}$; `new-pile`$_t$ is dependent on both `old-pile`$_{t-1}$ and `new-pile`$_{t-1}$. Data dependencies can be interpreted as a kind of time: if E_2 depends on the result of E_1, it is natural to view the evaluation of E_1 as happening *before* the evaluation of E_2.

8.1.3.2 Single-Threaded Data Flow

Iteration is an instance of a general technique for simulating state in a stateless language. State can always be simulated by adding state variables both as arguments and return values to every procedure in a program whose body either accesses or changes the state variables. The state of the program upon entering a procedure is encoded in the values of the state variable arguments, and the state of the program upon exiting a procedure is encoded in the values of the state variables returned as results. Because state is based on a notion of linearly-ordered time, we must guarantee that the data dependencies among the state variables form a linear chain. State variables satisfying this constraint are said to be passed through the program in a **single-threaded** fashion.

From this perspective, the problem with the bank account procedures is that the state of the system is not appropriately threaded through calls to these procedures. Suppose the state of the banking system is modeled by an entity called a *bank-state*. Then we can simulate state with the bank account procedures by extending each procedure to accept an additional bank-state argument and to return a pair of its usual return value and a (potentially updated) bank-state.

Suppose that every bank account bears a unique *account number*. Then we can represent a bank-state as a list of account-number/current-balance pairs. For example, the bank-state $[\langle 1729, 200\rangle, \langle 6821, 17\rangle]$ indicates that account 1729 has a current balance of 200 dollars and account 6821 has a current balance of 17 dollars. We will allow the same account number to appear more than once in a bank-state; in this case, the leftmost pair with a given account number indicates the current balance of that account. For example, in bank-state $[\langle 6821, 52\rangle, \langle 1729, 200\rangle, \langle 6821, 17\rangle]$, account 6821 has 52 dollars.

Here is an implementation of the `deposit!` procedure in this approach:

```
(define deposit!
  (lambda (amount account bank-state)
    (if (< amount 0)
        (pair 'failed bank-state)
        (let ((old&bank1 (balance account bank-state)))
          (let ((old (left old&bank1))
                (bank1 (right old&bank1)))
            (pair 'succeeded
                  (cons (pair account (+ old amount)) bank1)))))))
```

We assume that accounts are represented by their account numbers and that `balance` has been similarly modified to accept and return a bank-state. When it succeeds, `deposit!` creates a new bank state by prepending a new account-number/current-balance pair to the old one. A bank-state can be threaded through `make-account`[3], `balance`, and `withdraw!` in a similar fashion.

The `test-deposit!` procedure can also be modified to take a bank-state and thread it through each of the bank account operations:

```
(define test-deposit!
  (lambda (bank)
    (let ((acct&bank1 (make-account 100 bank)))
      (let ((acct (left acct&bank1))
            (bank1 (right acct&bank1)))
        (let ((old&bank2 (balance acct bank1)))
          (let ((old (left old&bank2))
                (bank2 (right old&bank2)))
            (let ((sym&bank3 (deposit! 17 acct bank2)))
              (let ((sym (left sym&bank3))
                    (bank3 (right sym&bank3)))
                (let ((new&bank4 (balance acct bank3)))
                  (let ((new (left new&bank4))
                        (bank4 (right new&bank4)))
                    (pair (- new old)
                          bank4)))))))))))
```

Given any initial bank-state, the new version of `test-deposit!` will return a pair of 17 (the desired result) and an updated bank-state.

▷ **Exercise 8.1** Provide definitions of `make-account`, `balance`, and `withdraw!` in which a bank-state is single-threaded through each procedure. ◁

▷ **Exercise 8.2** It is only necessary to single-thread a store through procedures that may update the store. For procedures that only access the store without updating it, it is sufficient to pass the store as an argument; such a procedure need not return a store as its result. An example of such a procedure is `balance`, which reads the balance of a bank account but does not write it.

- Write a version of `balance` that takes an account and a bank-state and returns only the balance of the account.
- Modify the state-simulating definitions of `deposit!` and `test-deposit!` to use

[3] `make-account` must also create a new, previously unused account number. Asking the caller to specify the number is an option, but it is better to include the next available account number as part of the bank state. If we don't care about wasting computational resources, we can compute a fresh account number from the current bank-state representation by adding `1` to the largest account number in the bank.

the new version of `balance`. ◁

8.1.3.3 Monadic Style

The bank-state threading details make the `test-deposit!` code hard to read, but some well-chosen abstractions can significantly increase readability. It helps to have a `with-pair` procedure that decomposes a pair into its component parts and passes these to a receiver procedure that names them:

```
(define with-pair
  (lambda (pair receiver)
    (receiver (left pair) (right pair))))
```

Using `with-pair`, `test-deposit!` can be simplified as follows:

```
(define test-deposit!
  (lambda (bank)
    (with-pair (make-account 100 bank)
      (lambda (acct bank1)
        (with-pair (balance acct bank1)
          (lambda (old bank2)
            (with-pair (deposit! 17 acct bank2)
              (lambda (sym bank3)
                (with-pair (balance acct bank3)
                  (lambda (new bank4)
                    (pair (- new old) bank4))))))))))))
```

Readability can be increased even further by hiding the threading of the bank-state altogether. Suppose that we define an **action** as any procedure that takes a bank-state and returns a pair of a value and a bank-state. In order to **perform** an action, we apply the action to a bank-state, which returns a value/bank-state pair. Such actions can be glued together by the `after` procedure in Figure 8.3, which takes a first action and a procedure that maps the value from performing the first action to a second action and returns a single action that performs the first action followed by the second. The figure also contains a `return` procedure that converts a value into an action and curried versions of `make-account`, `balance`, and `deposit!` that return actions when supplied with their non-bank-state arguments. With these abstractions, the `test-deposit!` procedure can be composed using four occurrences of `after` and one `return` (Figure 8.4).

This final version of `test-deposit!` illustrates a technique for threading state through a program that is known as **monadic style**. This style is based on gluing together state-threading components like the bank account actions in

```
(define after
  (lambda (action receiver)
    (lambda (bank)
      (with-pair (action bank)
        (lambda (val bank1)
          ((receiver val) bank1))))))

(define return
  (lambda (val)
    (lambda (bank) (pair val bank))))

(define *make-account
  (lambda (amount)
    (lambda (bank) (make-account amount bank))))

(define *deposit!
  (lambda (amount acct)
    (lambda (bank) (deposit! amount acct bank))))

(define *balance
  (lambda (acct)
    (lambda (bank) (balance acct bank))))
```

Figure 8.3: Procedures supporting a monadic style of threading bank-states through a program.

```
(define test-deposit!
  (after (*make-account 100)
    (lambda (acct)
      (after (*balance acct)
        (lambda (old)
          (after (*deposit! 17 acct)
            (lambda (sym)
              (after (*balance acct)
                (lambda (new)
                  (return (- new old))))))))))))
```

Figure 8.4: A version of `test-deposit!` written in monadic style.

a way that hides the details of the "plumbing." We have already seen monadic style in the denotational semantics of FL in Section 6.5. There, the *Computation* domain and functions like *with-value* are used to hide the messy details of propagating errors. In Section 8.2.4, we will extend the *Computation* domain to include a threaded store. By changing the meanings of a few functions like *with-value*, it is possible to thread the state through the semantics without changing many of the existing valuation functions. This illustrates the power of the monadic style.

In stateless languages, monadic style is commonly used to express stateful computations. The awkwardness of using a combiner like `after` can be avoided by syntactic sugar. For example, HASKELL supports a "`do` notation" in which the bank account testing function can be written as:

```
testDeposit =
  do a <- makeAccount 100
     b1 <- balance a
     deposit 17 a
     b2 <- balance a
     return (b2-b1)
```

As we shall see, this notation is not far from the way that stateful computations are expressed in stateful languages.

The name "monadic style" is derived from an algebraic structure, the **monad**, that captures the essence of manipulating state-threading components. For more information on monads and how monadic style can be used to express stateful computations in stateless languages like HASKELL, see [Wad95] and [JW93].

8.1.4 Imperative Programming

The bank account example demonstrates how it is possible to simulate state within a stateless language. However, even in monadic style, such simulations can be cumbersome. An alternative strategy is to develop a language paradigm that abstracts over the notion of state in such a way that the details of single-threading are automatically managed by the language. This is the essence of the **imperative programming paradigm**. In the imperative paradigm, all program state is conceptually bundled into a single entity called a **store** that is implicitly single-threaded through the program execution. Elements of the store are addressed by **locations**, unique identifiers that serve as unchanging names for time-dependent values. In the bank account example, bank-states correspond to stores and account numbers correspond to locations.

The advantage of the imperative programming paradigm is that programs can be shorter and more modular when the details of single-threading are im-

plicitly handled by the language. However, implicit single-threading has a down side: making explicit state variables implicit destroys referential transparency and thus makes programs harder to reason about.

The rest of this chapter explores how to model languages that exhibit state. We will see that the notions of store, location, and single-threading crop up in both operational and denotational descriptions of stateful languages.

8.2 Mutable Data: FL!

8.2.1 Mutable Cells

A one-slot cons is called a cell,
A two-slot cons makes pairs as well.
But I would bet a coin of bronze
There isn't any three-slot cons.

— *Guy L. Steele, Jr.*

Data structures whose components can change over time are said to be **mutable**. The simplest kind of mutable data is the **mutable cell**, a data structure characterized by a single time-dependent value called its **content**. A mutable cell corresponds to a one-slot cons cell in SCHEME or a pointer variable in languages like C and PASCAL. We will study mutable data in the context of FL!, a version of FL that supports mutable cells. We will use CBV as the default evaluation strategy for FL! because, as we shall see later, it makes more sense than CBN in languages that support mutation.

We begin by extending CBV FLK with features for supporting mutable cells. The modified kernel, FLK!, has the following syntax:

$$
\begin{array}{lll}
E_{FLK!} ::= \ldots & & \text{[FLK expressions]} \\
\quad \mid \texttt{(cell } E_{content}\texttt{)} & & \text{[Cell]} \\
\quad \mid \texttt{(begin } E_{sequent1}\ E_{sequent2}\texttt{)} & & \text{[Begin]} \\
O \in \text{Primop}_{FLK!} & = & \text{Primop}_{FLK} \cup \{\texttt{cell-ref}, \texttt{cell-set!}, \texttt{cell=?}, \texttt{cell?}\}
\end{array}
$$

Here is an informal description of the extensions:

- `(cell` E`)` returns a new mutable cell whose initial content is the value of E. We shall write cells as *id*:*val*, where *id* is a number that uniquely identifies the cell, and *val* is the content of the cell. In the following example, the expression allocates a cell with *id* number 1729 and content 3:

  `(cell (+ 1 2))` $\xrightarrow[FLK!]{}$ *1729*:*3*

- `(primop cell-ref` E`)` fetches the content of the cell computed by E. If the value of E is not a cell, this expression yields an error.

  `(primop cell-ref (cell (+ 1 2)))` $\xrightarrow[FLK!]{}$ *3*
  `(primop cell-ref (+ 1 2))` $\xrightarrow[FLK!]{}$ *error:not-a-cell*

- `(primop cell-set!` E_1 E_2`)` stores the value of E_2 in the cell computed by E_1. If the value of E_1 is not a cell, this expression yields an error. Since every FLK! expression must return a value, we shall arbitrarily specify that the value returned by a cell assignment expression is the unit value.

  `(primop cell-set! (cell (+ 1 2)) 4)` $\xrightarrow[FLK!]{}$ *unit*

- `(primop cell=?` E_1 E_2`)` returns *true* if E_1 and E_2 evaluate to the same cell and *false*if they evaluate to different cells. If at least one of E_1 or E_2 is not a cell, the expression yields an error.

```
(let ((c1 (cell 1))
      (c2 (cell 1)))
  (let ((c3 c1))
    (list (primop cell=? c1 c1)
          (primop cell=? c1 c2)
          (primop cell=? c1 c3))))
```
$\xrightarrow[FLK!]{}$ [*true*, *false*, *true*]

- `(primop cell?` E`)` returns *true* if E evaluates to a cell and *false*if it evaluates to some other value.

  `(pair (primop cell? 0) (primop cell? (cell 0)))` $\xrightarrow[FLK!]{}$ $\langle$*false*, *true*$\rangle$

- `(begin` E_1 E_2`)` first evaluates E_1, then evaluates E_2, and then returns the value of E_2. The value of E_1 is discarded.

```
(call (proc c
        (begin (primop cell-set! c 4)
               (primop cell-ref c)))
      (cell (+ 1 2)))
```
$\xrightarrow[FLK!]{}$ *4*

FL! is built on top of the kernel provided by FLK!. It has the same syntax as FL except that it includes the `cell` construct and a version of `begin` that can have an arbitrary number of sequents.

$E_{FL!}$ ::=	...	[FL expressions]
	| `(cell` E`)`	[Cell]
	| `(begin` $E_{sequent}$`*)`	[Begin]

The extended `begin` construct is defined by the following desugarings:

$\mathcal{D}_{\text{exp}}[\![\texttt{(begin)}]\!] = \texttt{\#u}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![E]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E_1\ E_2\ E_{rest}\texttt{*)}]\!] =$
$\quad\texttt{(begin } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![\texttt{(begin } E_2\ E_{rest}\texttt{*)}]\!]\texttt{)}$

This is the first time we've seen a sugar construct that has the same phrase tag as a kernel construct. This situation is common in practice. Of course, the desugaring for such a construct must guarantee that the general sugar form rewrites to the more restricted kernel form.

Like other primitive operator names, `cell-ref` and `cell-set!` are standard identifiers in FL!, where, respectively, they stand for procedures that access the content of a cell and change the content of a cell. Because these names are verbose, we will also introduce shorter synonyms in the standard environment: the name `^` is a synonym for `cell-ref`, and `:=` is a synonym for `cell-set!`.

8.2.2 Examples of Imperative Programming

The imperative programming paradigm is characterized by the use of side effects to perform computations. Because it is equipped with mutable cells, FL! supports the imperative paradigm. In this section, we present a few FL! programs that illustrate the imperative programming style.

8.2.2.1 Factorial

Here is an imperative version of an iterative factorial procedure written in FL!:

```
(define factorial
  (lambda (n)
    (let ((num (cell n))
          (ans (cell 1)))
      (letrec ((loop
                (lambda ()
                  (if (= (cell-ref num) 0)
                      (cell-ref ans)
                      (begin
                        (cell-set! ans (* (cell-ref num)
                                          (cell-ref ans)))
                        (cell-set! num (- (cell-ref num) 1))
                        (loop))))))
        (loop)))))
```

`num` and `ans` are cells that serve as the state variables of the iteration. The nullary `loop` procedure corresponds to a `while` loop in traditional imperative languages. On each round through the loop, the contents of the state variables are updated appropriately. The loop terminates when the content of `num` becomes zero.

It is instructive to compare the imperative version to a purely functional version:

```
(define factorial
  (lambda (n)
    (letrec ((loop (lambda (num ans)
                     (if (= num 0)
                         ans
                         (loop (- num 1) (* num ans))))))
      (loop n 1))))
```

In the functional version, every call to `loop` creates a new pair of variables named `num` and `ans`. In contrast, the imperative version shares one `num` and one `ans` variable across all the calls to `loop`. The correctness of the imperative version depends crucially on the order of the assignment expressions `(cell-set! ans ...)` and `(cell-set! num ...)`. If these expressions are swapped, then the imperative `factorial` no longer computes the right answer. This bug is due purely to the time-based nature of the imperative paradigm; the functional version does not exhibit the potential for this bug since all expressions have time-independent values. This illustrates one of the dangers of imperative programming: since many dependencies are implicit rather than explicit, subtle bugs are more likely, and they are harder to locate.

8.2.2.2 Bank Accounts

Using mutable cells, it is straightforward to implement the bank account scenario introduced in Section 8.1.2 and examined further in Section 8.1.3.

```
(define make-account
  (lambda (amount)
    (if (< amount 0)
        'failed
        (cell amount))))

(define balance
  (lambda (account) (cell-ref account)))
```

```
(define deposit!
  (lambda (amount account)
    (if (< amount 0)
        'failed
        (begin
          (cell-set! account (+ amount (cell-ref account)))
          'succeeded))))

(define withdraw!
  (lambda (amount account)
    (let ((bal (cell-ref account)))
      (if (or (< amount 0) (> amount bal))
          'failed
          (begin
            (cell-set! account (- bal amount))
            'succeeded)))))
```

Each account is represented by a distinct cell, and the bank account operations examine and change the content of this cell. Figure 8.5 shows the transcript of an interpreter session testing bank account objects.

```
(define a (make-account 100))
(define b (make-account 100))

(balance a) ⟶FL! 100
(balance b) ⟶FL! 100

(deposit! 17 b) ⟶FL! 'succeeded

(balance a) ⟶FL! 100
(balance b) ⟶FL! 117

(deposit! 23 a) ⟶FL! 'succeeded
(deposit! -23 b) ⟶FL! 'failed
(withdraw! 120 a) ⟶FL! 'succeeded
(withdraw! 120 b) ⟶FL! 'failed


(balance a) ⟶FL! 3
(balance b) ⟶FL! 117
```

Figure 8.5: Sample interactions with bank account objects.

While it is natural to represent accounts directly as cells, it is also somewhat insecure to do so. Every account should maintain the invariant that the balance

never slips below zero. But if an account is just a cell, then it is possible to violate this invariant by using `cell-set!` to directly store a negative number into an account. In general, it is wise to package up mutable data in a way that guarantees that important invariants cannot be violated (either accidentally or maliciously) by some other part of a software system.

First-class procedures provide an elegant means of encapsulating state so that it can only be manipulated in constrained ways. Figure 8.6 presents an alternate implementation in which bank accounts are represented as procedures that dispatch a message. The advantage to this approach is that the procedure provides a security wall for accessing and updating the account balance. In particular, the alternate implementation guarantees that the balance can never fall below zero.

8.2.2.3 Pattern Matching Revisited

The pattern matcher presented in Section 6.2.4.3 passes a dictionary through the computation in a single-threaded fashion. This means that the time-based sequence of dictionary values can alternately be represented as the changing content of a mutable cell. Figure 8.7 presents an imperative version of the `match-sexp` procedure that is based on this idea. (Procedures not defined in the figure are assumed to be the same as before.)

The `match-with-dict` procedure from Section 6.2.4.3 has been replaced by the internal `match!` procedure. Rather than taking a dictionary as its third argument, `match!` implicitly takes the current value of `dict-cell` as its argument. The `failed-flag` cell is used to simplify the handling of unsuccessful pattern matches.

8.2.3 An Operational Semantics for FLK!

In order to model the state exhibited by FLK!, we will use the notions of a **location** and a **store** introduced above. A location is a unique identifier for a mutable entity, and a store is a structure that associates each location with its value at a particular point in time. There are many ways to represent locations and stores. In our operational treatment, we will represent locations as numeric literals and stores as a sequence of location/value pairs:

$$\begin{array}{lllll} L & \in & \text{Location} & = & \mathit{Nat} \\ S & \in & \text{Store} & = & \text{Assignment}^* \\ Z & \in & \text{Assignment} & = & \text{Location} \times \text{ValueExp} \end{array}$$

We will assume the existence of a partial function *get* that finds the first value associated with a location in a store:

```
(define make-account
  (lambda (amount)
    (if (< amount 0)
        'failed
        (let ((account (cell amount)))
          (lambda (message)
            (cond ((sym=? message 'balance)
                   (cell-ref account))
                  ((sym=? message 'deposit!)
                   (lambda (amount)
                     (if (< amount 0)
                         'failed
                         (begin
                           (cell-set! account
                                      (+ amount (cell-ref account)))
                           'succeeded))))
                  ((sym=? message 'withdraw!)
                   (lambda (amount)
                     (let ((bal (cell-ref account)))
                       (if (or (< amount 0) (> amount bal))
                           'failed
                           (begin
                             (cell-set! account (- bal amount))
                             'succeeded)))))))))))

(define balance (lambda (account) (account 'balance)))

(define deposit!
  (lambda (amount account) ((account 'deposit!) amount)))

(define withdraw!
  (lambda (amount account) ((account 'withdraw!) amount)))
```

Figure 8.6: A message passing implementation of bank accounts.

```
;;; Imperative version of the MATCH-SEXP program. Procedures not defined
;;; here are the same as before.
(define match-sexp
  (lambda (pat sexp)
    (let ((dict-cell (cell (dict-empty)))
          (failed-flag (cell #f)))
      (letrec ((match!
                ;; MATCH! sets FAILED-FLAG true upon failure, and
                ;; updates the content of the DICT-CELL otherwise.
                ;; It always returns unit.
                (lambda (pat sexp)
                  (cond
                   ((failed-flag?) #u)
                   ((null? pat)
                     (if (null? sexp)
                         #u
                         (fail!)))
                   ((null? sexp) (fail!))
                   ((pattern-constant? pat)
                    (if (sexp=? pat sexp) #u (fail!)))
                   ((pattern-variable? pat)
                    (dict-bind! (pattern-variable-name pat) sexp))
                   (else
                    (begin (match! (car pat) (car sexp))
                           (match! (cdr pat) (cdr sexp)))))))

               (failed-flag? (lambda () (cell-ref failed-flag)))

               (fail! (lambda () (cell-set! failed-flag #t)))

               (dict-bind!
                (lambda (sym sexp)
                  (let ((new-dict (dict-bind
                                   pat sexp (cell-ref dict-cell))))
                    (if (failed? new-dict)
                        (fail!)
                        (cell-set! dict-cell new-dict))))))
        (begin
          (match! pat sexp)
          (if (cell-ref failed-flag)
              'failed
              (cell-ref dict-cell)))))))
```

Figure 8.7: Version of `match-sexp` written in an imperative style.

get : Location $\rightarrow$ Store $\rightharpoonup$ ValueExp
$(get\ L\ \langle L, V\rangle\ .\ S) = V$
$(get\ L_1\ \langle L_2, V\rangle\ .\ S) = (get\ L_1\ S)$, where $L_1 \neq L_2$

The FLK! SOS uses the syntactic domain $E_{mixed} \in$MixedExp, which has a grammar isomorphic to FLK! except for the addition of a `(*cell*` L`)` construct that is used to represent cell values:

E_{mixed} ::= ... [FLK! expressions]
| `(*cell*` L`)` [Cell Value]

The `*cell*` construct may not appear in a user program. ValueExp is the same as that for CBV FLK except that it also contains cell values:

$$V \in \text{ValueExp} = \text{Lit} \cup \{\texttt{(proc}\ I\ E\texttt{)}\} \cup \{\texttt{(pair}\ V_1\ V_2\texttt{)}\} \cup \{\texttt{(*cell*}\ L\texttt{)}\}$$

An operational semantics for FLK! is specified by

$$\langle CF_{FLK!}, \Rightarrow, FC_{FLK!}, IF_{FLK!}, OF_{FLK!}\rangle,$$

where the rewrite rules defining $\Rightarrow$ are specified later and

$$\begin{aligned} CF_{FLK!} &= \text{MixedExp} \times \text{Store} \\ FC_{FLK!} &= \text{ValueExp} \times \text{Store} \\ IF_{FLK!} &= \lambda E\,.\,\langle E, [\,]_{Assignment}\rangle \\ OF_{FLK!} &= \lambda\langle V, S\rangle\,.\,(output\ V) \end{aligned}$$

$$\begin{aligned} (output\ L) &= L \\ (output\ \texttt{(proc}\ I\ E\texttt{)}) &= \texttt{procedure} \\ (output\ \texttt{(pair}\ V_1\ V_2\texttt{)}) &= \texttt{(pair}\ (output\ V_1)\ (output\ V_2)\texttt{)} \\ (output\ \texttt{(*cell*}\ L\texttt{)}) &= \texttt{cell}. \end{aligned}$$

The first component (code component) of an FLK! configuration is a mixed expression that serves the same role as an entire FLK configuration. An FLK! configuration has an additional state component: a store that models the current mapping of locations to value expressions. A computation begins with the initial expression and an empty store, $[\,]_{\text{Assignment}}$, and runs until the code component becomes a value (or the configuration becomes stuck). At this point, an approximation to the final value is returned as the result of the original FLK! expression.

The core rewrite rules for the FLK! semantics appear in Figure 8.8. The `cell` construct and the primitive operators `cell-ref` and `cell-set!` are the only constructs that directly manipulate the store. The [*cell-alloc*] axiom allocates a new location, L_{fresh}, which does not appear in the store, and extends the

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle \texttt{(cell } E\texttt{)}, S\rangle \Rightarrow \langle \texttt{(cell } E'\texttt{)}, S'\rangle} \quad [\textit{cell-progress}]$$

$$\langle \texttt{(cell } V\texttt{)}, S\rangle \Rightarrow \langle \texttt{(*cell* } L_{fresh}\texttt{)}, (\langle L_{fresh}, V\rangle \texttt{ . } S)\rangle, \text{ where } L_{fresh} \text{ is a location that does not appear in } S. \quad [\textit{cell-alloc}]$$

$$\langle \texttt{(primop cell-ref (*cell* } L\texttt{))}, S\rangle \Rightarrow \langle V, S\rangle, \text{ where } (\textit{get } L\ S) = V \quad [\textit{cell-ref}]$$

$$\langle \texttt{(primop cell-set! (*cell* } L\texttt{) } V\texttt{)}, S\rangle \Rightarrow \langle \texttt{\#u}, \langle L, V\rangle \texttt{ . } S\rangle \quad [\textit{cell-set!}]$$

$$\langle \texttt{(primop cell=? (*cell* } L\texttt{) (*cell* } L\texttt{))}, S\rangle \Rightarrow \langle \texttt{\#t}, S\rangle \quad [\textit{cell=?-true}]$$

$$\langle \texttt{(primop cell=? (*cell* } L_1\texttt{) (*cell* } L_2\texttt{))}, S\rangle \Rightarrow \langle \texttt{\#f}, S\rangle, \text{ where } L_1 \neq L_2 \quad [\textit{cell=?-false}]$$

$$\langle \texttt{(primop cell? (*cell* } L\texttt{))}, S\rangle \Rightarrow \langle \texttt{\#t}, S\rangle \quad [\textit{cell?-true}]$$

$$\langle \texttt{(primop cell? } V\texttt{)}, S\rangle \Rightarrow \langle \texttt{\#f}, S\rangle, \text{ where } V \neq \texttt{(*cell* } L\texttt{)} \quad [\textit{cell?-false}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle \texttt{(begin } E_1\ E_2\texttt{)}, S\rangle \Rightarrow \langle \texttt{(begin } E_1\,'\ E_2\texttt{)}, S'\rangle} \quad [\textit{begin-first}]$$

$$\langle \texttt{(begin } V\ E\texttt{)}, S\rangle \Rightarrow \langle E, S\rangle \quad [\textit{begin-rest}]$$

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle \texttt{(rec } I\ E\texttt{)}, S\rangle \Rightarrow \langle \texttt{(rec } I\ E'\texttt{)}, S'\rangle} \quad [\textit{rec-body}]$$

$$\langle \texttt{(rec } I\ V\texttt{)}, S\rangle \Rightarrow \langle [\texttt{(rec } I\ V\texttt{)}/I]\, V, S\rangle \quad [\textit{cbv-rec}]$$

Figure 8.8: Core rewrite rules for FLK!.

store with a new binding between L_{fresh} and the given value. The result of this operation is a `*cell*` value that maintains an index into the store. The [*cell-ref*] rule uses *get* to extract the binding at the location specified by the `*cell*` value. Even though *get* is only a partial function, a `cell-ref` expression can never get stuck because every location appearing in a `*cell*` value must appear in the store. The [*cell-set!*] rule returns a unit value but also prepends a new location/value pair to the store to reflect the assignment.

We have chosen to represent stores as explicit sequences of bindings, but other representations are certainly possible (e.g., representing stores as functions that map locations to values). In our approach, the number of bindings in a store is equal to the number of allocations and assignments performed by the program. An implementation based on such a strategy would be disastrously inefficient: the size of the store would grow throughout the computation, and cell references would take time linear in the size of the growing store. But our goal here is to give a simple semantics for stores, not to implement them efficiently. Any reasonable implementation of FLK! would represent stores in a way that takes advantage of the state-based nature of addressable memory in physical computers.

Of the remaining rules in Figure 8.8, the `begin` rules are straightforward, but the `rec` rules deserve some explanation. Handling `rec` is a bit tricky in the presence of side effects. The basic problem is illustrated by the following FL! example:

```
(let ((counter (cell 0)))
  (begin ((rec fact
            (begin (cell-set! counter
                     (+ (cell-ref counter) 1))
                   (lambda (n)
                     (if (= n 0)
                         1
                         (* n (fact (- n 1)))))))
          5)
         (cell-ref counter)))
```

Here the value computed by the `rec` expression is a factorial procedure. But we're not so much interested in the value of the `rec` as we are in the value of the `counter` cell at the end of the expression. This value tells us how many times `counter` is incremented during the evaluation of the `rec` expression. Presumably, the content of `counter` should be 1. However, if the CBN rule

$$\langle (\texttt{rec}\ I\ E), S\rangle \Rightarrow \langle [(\texttt{rec}\ I\ E)/I]E, S\rangle \qquad [\textit{cbn-rec}]$$

were used, then the value of the above expression would be 6 because the `begin`

expression that is the body of the `rec` would be copied in each unwinding and would be evaluated six times.

To avoid this behavior, the [*cbv-rec*] rule only unwinds the `rec` when the body is a value. The [*rec-body*] rule takes care of rewriting the body of the `rec` into a member of ValueExp. This means that any side effects encountered during the evaluation of the `rec` body are performed only once. In a CBV semantics, the rewriting of the `rec` body will only terminate in a non-stuck state when all uses of the formal parameter introduced by `rec` that appear in the body are "shielded" from immediate evaluation by a `proc`.

The rest of the rules for FLK! appear in Figure 8.9. These rules never actually examine or update the store. Rather, they just specify the "plumbing" that passes the store through the computation in a single-threaded fashion. This guarantees that any changes made by `cell` or `cell-set!` are visible to later uses of `cell-ref`. Except for the additional shuffling of the store component, these rules are the same as those for CBV FLK. The [*FLK-prim*] rule says that the behavior of primitive applications from FLK (as specified by $\Rightarrow_{FLK}$) is inherited by FLK! (as specified by $\Rightarrow$), where the store is unaffected by all such applications.

As a simple example of the FLK! SOS, consider the operational evaluation of the expression `(call` E_{proc} `(cell 3))` where E_{proc} is:

```
(proc c
  (begin (primop cell-set! c
           (primop + 1
            (primop cell-ref c)))
         (primop cell-ref c)))
```

Figure 8.10 shows the transition sequence associated with this expression. Note how the cell value `(*cell* 0)` serves as an unchanging index into the time-dependent store.

▷ **Exercise 8.3** The `begin` construct need not be primitive in FLK!. A desugaring of `begin` into other FLK! constructs must take advantage of the fact that the only notion of time in FL has to do with data dependency. That is, the only thing that forces an expression to be evaluated is that its value is used in the evaluation of another expression. This suggests the following desugaring for `begin` into other FLK! expressions.

$$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E_1 \ E_2\texttt{)}]\!] = \texttt{(call (proc (}I_{ignore}\texttt{) } \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{) } \mathcal{D}_{\text{exp}}[\![E_1]\!]\texttt{)}$$

where $I_{ignore} \notin FreeIds[\![E_2]\!]$.

a. The desugaring for `begin` given above uses constructs only from FLK, which does not support state. Is it possible to determine whether `begin` actually works as advertised (i.e., evaluates E_1 before E_2) in a language that does not support state? Explain your answer, using examples where appropriate.

$$\langle (\texttt{call (proc } I\ E)\ V), S\rangle \Rightarrow \langle [V/I]E, S\rangle \qquad [\textit{cbv-call}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle (\texttt{call } E_1\ E_2), S\rangle \Rightarrow \langle (\texttt{call } E_1\,'\ E_2), S'\rangle} \qquad [\textit{rator-progress}]$$

$$\frac{\langle E_2, S\rangle \Rightarrow \langle E_2\,', S'\rangle}{\begin{array}{l}\langle (\texttt{call (proc } I\ E_1)\ E_2), S\rangle \\ \Rightarrow \langle (\texttt{call (proc } I\ E_1)\ E_2\,'), S'\rangle\end{array}} \qquad [\textit{rand-progress}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle (\texttt{if } E_1\ E_2\ E_3), S\rangle \Rightarrow \langle (\texttt{if } E_1\,'\ E_2\ E_3), S'\rangle} \qquad [\textit{test-progress}]$$

$$\langle (\texttt{if \#t } E_1\ E_2), S\rangle \Rightarrow \langle E_1, S\rangle \qquad [\textit{if-true}]$$

$$\langle (\texttt{if \#f } E_1\ E_2), S\rangle \Rightarrow \langle E_2, S\rangle \qquad [\textit{if-false}]$$

$$\frac{\langle E_{left}, S\rangle \Rightarrow \langle E_{left}\,', S'\rangle}{\langle (\texttt{pair } E_{left}\ E_{right}), S\rangle \Rightarrow \langle (\texttt{pair } E_{left}\,'\ E_{right}), S'\rangle} \qquad [\textit{left-progress}]$$

$$\frac{\langle E_{right}, S\rangle \Rightarrow \langle E_{right}\,', S'\rangle}{\langle (\texttt{pair } V_{left}\ E_{right}), S\rangle \Rightarrow \langle (\texttt{pair } V_{left}\ E_{right}\,'), S'\rangle} \qquad [\textit{right-progress}]$$

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle (\texttt{primop } O\ E), S\rangle \Rightarrow \langle (\texttt{primop } O\ E'), S'\rangle} \qquad [\textit{unary-arg}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle (\texttt{primop } O\ E_1\ E_2), S\rangle \Rightarrow \langle (\texttt{primop } O\ E_1\,'\ E_2), S\rangle} \qquad [\textit{binary-arg1}]$$

$$\frac{\langle E_2, S\rangle \Rightarrow \langle E_2\,', S'\rangle}{\langle (\texttt{primop } O\ V_1\ E_2), S\rangle \Rightarrow \langle (\texttt{primop } O\ V_1\ E_2\,'), S'\rangle} \qquad [\textit{binary-arg2}]$$

$$\frac{(\texttt{primop } O_{FLK}\ V^*) \Rightarrow_{FLK} V_{result}}{\langle (\texttt{primop } O_{FLK}\ V^*), S\rangle \ \Rightarrow\ \langle V_{result}, S\rangle,} \qquad [\textit{FLK-prim}]$$

where $O_{FLK} \in \text{Primop}_{FLK!} - \{\texttt{cell-ref}, \texttt{cell-set!}, \text{ and } \texttt{cell?}\}$

Figure 8.9: FLK! rewrite rules for single-threading the store.

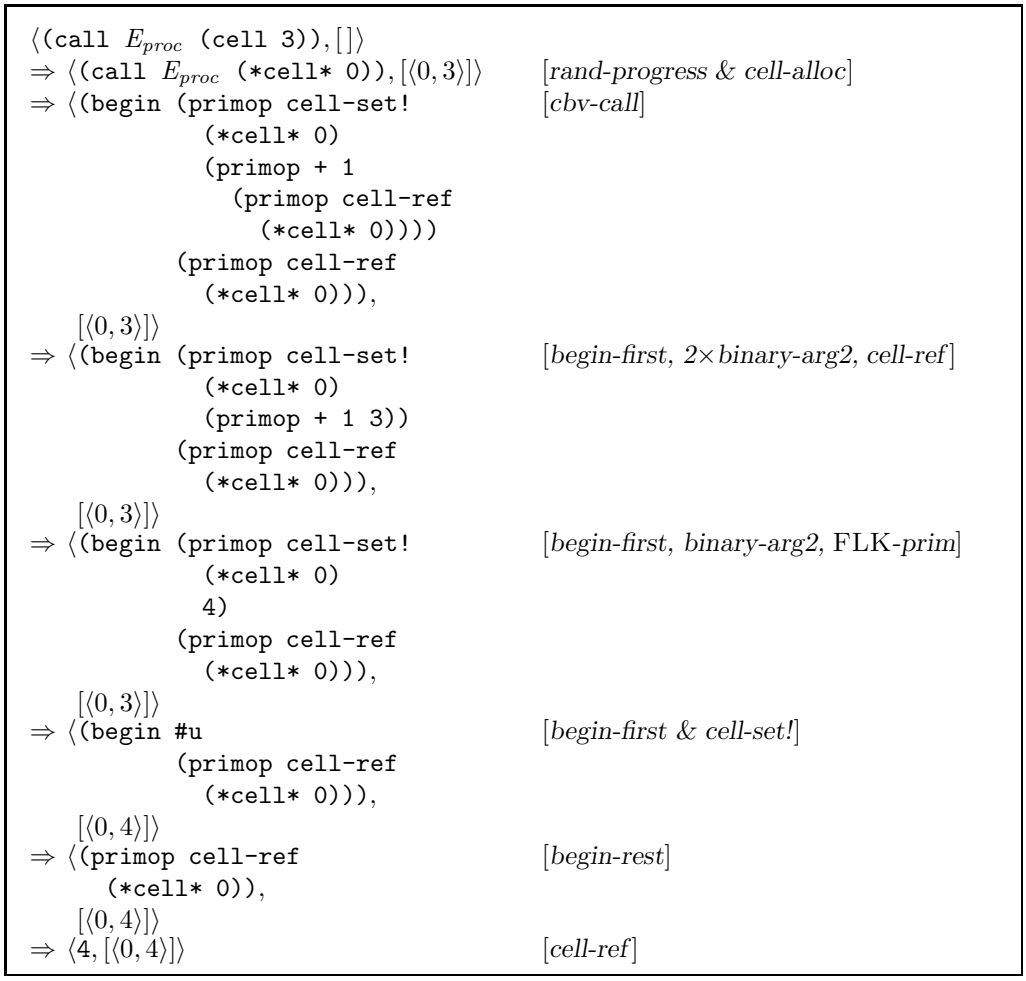


Figure 8.10: Operational evaluation of a sample FLK! expression.

b. Explain why the above desugaring would not work for a CBN version of FL!.

c. Write a desugaring for `begin` in CBV FLK! that does not require any condition involving the free variables of E_1 or E_2. (Hint: use thunks!)

d. Is it possible to write a desugaring for `begin` that works in both CBV and CBN FLK! ? If so, give the desugaring; if not, explain why not. ◁

▷ **Exercise 8.4** The introduction of side effects can complicate reasoning about programs. For example, program transformations that are safe in FL aren't necessarily safe in FL!.

- List three transforms that are safe in FL but not in FL!. Provide counter-examples to demonstrate why they are not safe in FL!.
- List three transforms that are safe in both FL and FL!.
- Consider transforms that do not mention any of the new features of FLK!. Are there any such transforms that are safe in FL! but not in FL? If so, exhibit such a transform. If not, explain. ◁

8.2.4 A Denotational Semantics for FLK!

Now we'll study the semantics of FLK! from the denotational perspective. As in the operational approach, notions of location and store will be used to model state. The notion of computation will be modified so that stores flow through a computation in a single-threaded fashion. The power of the computation abstraction will be illustrated by the fact that only those constructs that explicitly refer to the store need new valuation clauses; other constructs are described by their (unmodified) FLK valuation clauses.

8.2.4.1 Stores

The denotational treatment of stores and locations is summarized in Figure 8.11.

Here locations are represented as natural numbers and stores are represented as functions that map locations to elements of the *Assignment* domain. Stores do not map locations directly to values because it is necessary to encode the fact that not all locations have values assigned to them. The distinguished element *unassigned* in the lifted sum domain *Assignment* is used to indicate that a location is unassigned. *unassigned* serves the same purpose for stores that *unbound* serves for environments.

The domain *Storable* of storable entities varies from language to language. In FLK!, which is a CBV language, *Storable* = *Value*, but a CBN version of

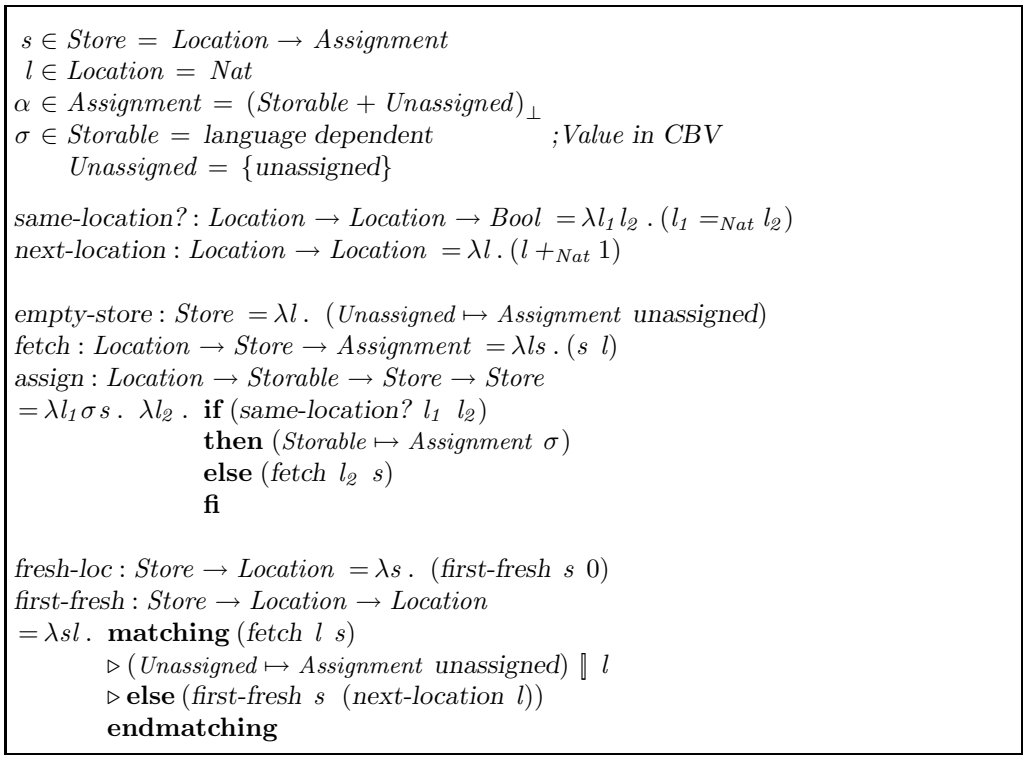


Figure 8.11: Denotational treatment of stores.

FLK! would have *Storable* = *Computation*. In both CBV and CBN FLK!, it happens that *Storable* = *Denotable*, but this need not be the case in general. For example, in PASCAL, procedures can be named and (with certain restrictions) be passed as arguments, but they may not be assigned to variables or stored as the components of data structures.

There are several auxiliary functions for manipulating stores. *fetch* and *assign* are functions on stores that are reminiscent of *lookup* and *extend* on environments. The purpose of *fresh-loc* is to return an unassigned location from the given store. Since locations are natural numbers, one way of doing this is by scanning the store starting with location 0 and incrementing the location until an unassigned location is found. We assume an unbounded store, so that *fresh-loc* never fails to return a fresh location. To model a bounded store (which would be more realistic), *fresh-loc* could potentially return an indication that the attempt to find a fresh location failed.

8.2.4.2 Computations

Previously, a computation was just an expressible value. But in the presence of state, a computation needs to embody the single-threaded nature of stores. The following domain definition captures this idea:

$$c \in \mathit{Computation} = \mathit{Store} \rightarrow (\mathit{Expressible} \times \mathit{Store})$$

Here, a computation accepts an initial store and returns two entities:

- The expressible value computed by the computation.
- A final store that reflects all the allocations and assignments performed by the computation.

When composing two computations, single-threadedness can be achieved by supplying the final store of the first computation as the initial store of the second.

It is not difficult to show that the new *Computation* domain is pointed. This means that it is possible to find fixed points over computations, as required in the semantics of `rec`.

Recall that numerous auxiliary functions must be defined as part of the computation abstraction. Figure 8.12 shows the definitions of these functions for the store-based version of *Computation*. *val-to-comp* injects a value into a computation by injecting it into an expressible value and passing a store around it unchanged. Similarly, *error-comp* passes a store around an error expressible value.

The main means of gluing computations together is *with-value*. It takes a computation c_1 and a function f that maps a value to a computation c_2 and

$c \in \mathit{Computation} = \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store})$

$\mathit{expr\text{-}to\text{-}comp} : \mathit{Expressible} \to \mathit{Computation} = \lambda x\,.\ \lambda s\,.\ \langle x, s\rangle$

$\mathit{val\text{-}to\text{-}comp} : \mathit{Value} \to \mathit{Computation} = \lambda v\,.\ (\mathit{expr\text{-}to\text{-}comp}\ (\mathit{Value} \mapsto \mathit{Expressible}\ v))$

$\mathit{err\text{-}to\text{-}comp} : \mathit{Error} \to \mathit{Computation} = \lambda I\,.\ (\mathit{expr\text{-}to\text{-}comp}\ (\mathit{Error} \mapsto \mathit{Expressible}\ I))$

$\mathit{error\text{-}comp} : \mathit{Computation} = (\mathit{err\text{-}to\text{-}comp}\ \texttt{error})$

$\mathit{with\text{-}value} : \mathit{Computation} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda c f\ .\ \lambda s_1\ .$ **matching** $(c\ s_1)$
$\quad \triangleright \langle(\mathit{Value} \mapsto \mathit{Expressible}\ v), s_2\rangle \Vert (f\ v\ s_2)$
$\quad \triangleright \langle(\mathit{Error} \mapsto \mathit{Expressible}\ \mathit{error}), s_2\rangle \Vert (\mathit{error\text{-}comp}\ s_2)$
$\quad$ **endmatching**

with-values, *with-boolean*, *with-procedure*, etc. can be written in terms of *with-value*.

$\mathit{check\text{-}location} : \mathit{Value} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda v f\ .$ **matching** v
$\quad \triangleright (\mathit{Location} \mapsto \mathit{Value}\ l) \Vert (f\ l)$
$\quad \triangleright$ **else** *error-comp*
$\quad$ **endmatching**

check-boolean, *check-procedure*, etc. are similar.

Figure 8.12: Store-based implementation of the computation abstraction.

returns the computation that results from composing c_1 and c_2. Like the action-combining `after` procedure in Section 8.1.3.3, the main purpose of *with-value* is to support the monadic style of threading state by handling the "plumbing" between computations: the value argument to f is the (non-error) expressible value produced by c_1 and the initial store of c_2 is the final store of c_1. In the case where c_1 produces an error rather than a value, f is ignored and the resulting computation is equivalent to c_1. It is instructive to unwind the type of f:

$$\mathit{Value} \to \mathit{Computation} \;=\; \mathit{Value} \to \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store})$$

This makes it clear that f can be viewed as a function that maps two (curried) arguments (a value and store) to two (paired) results (an expressible value and store).

Other *with-* functions, like *with-values*, *with-procedure*, *with-boolean* can be written in the same style as *with-value*. There is a parallel collection of *check-* functions that differ from the *with-* functions only in that their initial argument is a value rather than a computation.

In the presence of state, there are a few more auxiliary functions involving computations that are especially handy. These are defined in Figure 8.13. *allocating* allocates a location for a storable value and passes it (and the up-

$$\begin{array}{l}
\mathit{allocating} : \mathit{Storable} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda\sigma f \;.\; \lambda s \;.\; (f \;\; (\mathit{fresh\text{-}loc}\;\; s) \;\; (\mathit{assign}\;\; (\mathit{fresh\text{-}loc}\;\; s) \;\; \sigma \;\; s)) \\
\\
\mathit{fetching} : \mathit{Location} \to (\mathit{Storable} \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda l f \;.\; \lambda s \;.\; \mathbf{matching}\;(\mathit{fetch}\;\; l \;\; s) \\
\qquad\qquad \triangleright\,(\mathit{Storable} \mapsto \mathit{Assignment}\;\sigma) \;[\!]\; (f \;\; \sigma \;\; s) \\
\qquad\qquad \triangleright\,\mathbf{else}\;(\mathit{error\text{-}comp}\;\; s) \\
\qquad\qquad \mathbf{endmatching} \\
\\
\mathit{update} : \mathit{Location} \to \mathit{Storable} \to \mathit{Computation} \\
= \lambda l \sigma \;.\; \lambda s \;.\; \langle (\mathit{Value} \mapsto \mathit{Expressible}\;\; (\mathit{Unit} \mapsto \mathit{Value}\;\; \mathit{unit})), (\mathit{assign}\;\; l \;\; \sigma \;\; s) \rangle \\
\\
\mathit{sequence} : \mathit{Computation} \to \mathit{Computation} \to \mathit{Computation} \\
= \lambda c_1\, c_2 \;.\; (\mathit{with\text{-}value}\;\; c_1 \;\; (\lambda v \;.\; c_2))
\end{array}$$

Figure 8.13: Auxiliary functions for store-based computations.

dated store) to a computation-producing function. *fetching* finds the storable value at a location and passes it (and the unchanged store) to a computation-producing function. *update* takes a location and storable value and returns a unit-producing computation whose final store includes an assignment between the location and value. *sequence* glues two computations together by supplying

the final store of the first as the initial store of the second; the expressible value produced by the first is ignored.

Reasoning about computations directly in terms of the auxiliary functions can be very tedious. Figure 8.14 presents a number of high-level equalities that greatly facilitate reasoning about computations. We leave the proofs of these

1. $(\textit{with-value}\ (\textit{val-to-comp}\ v)\ f) = (f\ v)$
2. $(\textit{with-value}\ c\ (\lambda v\,.\,(\textit{val-to-comp}\ v))) = \mathrm{c}$
3. $(\textit{with-procedure}\ (\textit{val-to-comp}\ (\textit{Procedure} \mapsto \textit{Value}\ p))\ f) = (f\ p)$
   Similarly for *with-boolean*, *with-integer*, etc.
4. $(\textit{with-value}\ (\textit{with-value}\ c\ f)\ g) = (\textit{with-value}\ c\ (\lambda v\,.\,(\textit{with-value}\ (f\ v)\ g)))$
5. $(\textit{with-value}\ (\textit{check-location}\ v\ f)\ g)$
   $= (\textit{check-location}\ v\ (\lambda l\,.\,(\textit{with-value}\ (f\ l)\ g)))$
   similarly for *check-boolean*, *check-integer*
6. $(\textit{with-value}\ (\textit{allocating}\ \sigma\ f)\ g) = (\textit{allocating}\ \sigma\ (\lambda l\,.\,(\textit{with-value}\ (f\ l)\ g)))$
7. $(\textit{with-value}\ (\textit{fetching}\ l\ f)\ g) = (\textit{fetching}\ l\ (\lambda\sigma\,.\,(\textit{with-value}\ (f\ \sigma)\ g)))$
8. $(\textit{with-value}\ (\textit{update}\ l\ \sigma)\ f)$
   $= (\textit{sequence}\ (\textit{update}\ l\ \sigma)\ (f\ (\textit{Unit} \mapsto \textit{Value}\ \textit{unit})))$
9. $(\textit{with-value}\ (\textit{sequence}\ c1\ c2)\ f) = (\textit{sequence}\ c1\ (\textit{with-value}\ c2\ f))$

Figure 8.14: Useful equalities on computations. It is assumed that newly introduced variables do not conflict with free identifiers elsewhere in the expression.

equalities as exercises for the reader. We require the first four equalities in Figure 8.14 to be true of any notion of computation that we introduce, and equalities 5–9 to be true of any notion of computation that supports state.

8.2.4.3 Valuation Clauses

The denotational specification of FLK! is summarized in Figure 8.15. The *Value* domain has been extended with locations, which represent cell values. Since FLK! is a CBV language, both *Denotable* and *Storable* equal *Value*. As always, $\mathcal{E}$ has the signature Exp $\to$ *Environment* $\to$ *Computation*. There are two semantic functions for primitives. $\mathcal{P}_{FLK!}$ is the version for FLK!, while $\mathcal{P}_{FLK}$ is the version inherited from FLK.

With the help of the auxiliary functions, the valuation clauses are surprisingly compact. In fact, only one clause (`rec`) explicitly mentions the store! `begin` sequences two computations. `cell` allocates a location for its content and returns

$c \in \mathit{Computation} = \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store})$
$v \in \mathit{Value} = \mathit{Unit} + \mathit{Bool} + \mathit{Int} + \mathit{Sym} + \mathit{Pair} + \mathit{Procedure} + \mathit{Location}$
$\delta \in \mathit{Denotable} = \mathit{Value}$
$\sigma \in \mathit{Storable} = \mathit{Value}$
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Computation}$

$\mathcal{E} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation}$
$\mathcal{P}_{FLK} : \mathrm{Primop} \to \mathit{Value}^* \to \mathit{Expressible}$
$\mathcal{P}_{FLK!} : \mathrm{Primop} \to \mathit{Value}^* \to \mathit{Computation}$

$\mathcal{E}[\![\texttt{(begin } E_1 \ E_2\texttt{)}]\!] = \lambda e\,.\ (\mathit{sequence}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\mathcal{E}[\![E_2]\!]\ e))$

$\mathcal{E}[\![\texttt{(cell } E\texttt{)}]\!]$
$= \lambda e\,.\ (\mathit{with\text{-}value}\ (\mathcal{E}[\![E]\!]\ e)$
$\qquad (\lambda v\,.\ (\mathit{allocating}\ v\ (\lambda l\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l))))))$

$\mathcal{P}_{FLK!}[\![\texttt{cell-ref}]\!] = \lambda[v]\,.\ (\mathit{check\text{-}location}\ v\ (\lambda l\,.\ (\mathit{fetching}\ l\ (\lambda v\,.\ (\mathit{val\text{-}to\text{-}comp}\ v)))))$

$\mathcal{P}_{FLK!}[\![\texttt{cell-set!}]\!] = \lambda[v_1, v_2]\,.\ (\mathit{check\text{-}location}\ v_1\ (\lambda l\,.\ (\mathit{update}\ l\ v_2)))$

$\mathcal{P}_{FLK!}[\![\texttt{cell=?}]\!] =$
$\quad \lambda[v_1, v_2]\,.\ (\mathit{check}\text{-location}\ v_1$
$\qquad (\lambda l_1\ .\ (\mathit{check}\text{-location}\ v_2$
$\qquad\quad (\lambda l_2\ .\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Bool} \mapsto \mathit{Value}\ (l_1 = l_2)))))))$

$\mathcal{P}_{FLK!}[\![\texttt{cell?}]\!] = \lambda[v]\,.$ **matching** v
$\qquad \triangleright\, (\mathit{Location} \mapsto \mathit{Value}\ l)\ [\![\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Bool} \mapsto \mathit{Value}\ \mathit{true}))$
$\qquad \triangleright$ **else** $(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Bool} \mapsto \mathit{Value}\ \mathit{false}))$
$\qquad$ **endmatching**

$\mathcal{P}_{FLK!}[\![O]\!]$; $O \in$Primop $- \{\texttt{cell-ref}, \texttt{cell-set!}, \texttt{cell?}\}$
$= \lambda v^*\,.\ (\mathit{expr\text{-}to\text{-}comp}\ (\mathcal{P}_{FLK}[\![O]\!]\ v^*))$

$\mathcal{E}[\![\texttt{(rec } I\ E\texttt{)}]\!] = \lambda e\,.\ \mathbf{fix}_{\mathit{Computation}}\ (\lambda c\,.\ \lambda s\,.\ \mathcal{E}[\![E]\!]\ [I :: (\mathit{extract\text{-}value}\ c\ s)]e\ s)$

$\mathit{extract\text{-}value} : \mathit{Computation} \to \mathit{Store} \to \mathit{Binding}$
$= \lambda cs\,.$ **matching** $(c\ s)$
$\qquad \triangleright < (\mathit{Value} \mapsto \mathit{Expressible}\ v),\ s' > [\![\ (\mathit{Denotable} \mapsto \mathit{Binding}\ v)$
$\qquad \triangleright < (\mathit{Error} \mapsto \mathit{Expressible}\ \mathit{error}),\ s' > [\![\ \bot_{\mathit{Binding}}$
$\qquad$ **endmatching**

Figure 8.15: Essential valuation clauses for FLK!. Clauses not shown here inherit their definition from FLK.

the location as its resulting value. `cell-ref` fetches the value of a location and returns it, while `cell-set!` updates a location to contain a new value. `cell?` simply checks the tag on a value. Other primitives are handled by passing them off to $\mathcal{P}_{FLK}$ and converting the result into a computation. This works because none of the primitives inherited from FLK has any effect on the store.

The only really tricky clause is the one for `rec`. The valuation clause presented here is a variant of the CBV version presented in Section 7.1.3. The only difference is that it is necessary to supply *extract-value* with the current store in order to coerce the computation into a binding.

And that's it! By the magic of the monadic style, all the other valuation clauses are inherited unchanged from the denotational definition of CBV FLK. For example, the clause for `call` is still:

$$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \\ \lambda e\,.\ (\textit{with-procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\ (\textit{with-value}\ (\mathcal{E}[\![E_2]\!]\ e)\ p)))$$

The valuation clauses are very concise, but their level of abstraction can make them difficult to understand. To get a better feel for the valuation clauses, it can be helpful to strip away the abstractions by "in-lining" the auxiliary functions. For example, here is a version of the `call` clause without any auxiliary functions:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \\ \quad \lambda e s_0\,.\ \textbf{matching}\ (\mathcal{E}[\![E_1]\!]\ e\ s_0) \\ \qquad \triangleright \langle (\textit{Value} \mapsto \textit{Expressible}\ (\textit{Procedure} \mapsto \textit{Value}\ p)), s_1\rangle\ [\![\\ \qquad\ \textbf{matching}\ (\mathcal{E}[\![E_2]\!]\ e\ s_1) \\ \qquad\ \triangleright \langle (\textit{Value} \mapsto \textit{Expressible}\ v), s_2\rangle\ [\![\ (p\ v\ s_2) \\ \qquad\ \triangleright \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_2\rangle\ [\![\ \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_2\rangle \\ \qquad\ \textbf{endmatching} \\ \qquad \triangleright \langle (\textit{Value} \mapsto \textit{Expressible}\ v), s_1\rangle\ [\![\ \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_1\rangle \\ \qquad \triangleright \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_1\rangle\ [\![\ \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_1\rangle \\ \qquad \textbf{endmatching}\end{array}$$

The single-threaded nature of the store that is implicit in the original clause is explicit in the expanded clause. Evaluating E_1 in e with s_0 yields an expressible value (call it x_1) and a store s_1. If the x_1 is a procedure value p, E_2 is evaluated in e with s_1 to yield a second expressible value (call it x_2) and another store, s_2. If x_2 is a value v, then p, whose signature is

$$\textit{Value} \to \textit{Store} \to (\textit{Expressible} \times \textit{Store})$$

is applied to the value and the store. In error situations (x_1 is not a procedure or x_2 is not a value), expressible error values are propagated along with the updated store.

You may find it helpful to perform this sort of expansion on other valuation clauses. After you have done several, you may start to appreciate the purpose

of the auxiliary functions! As usual, it is also instructive to make sure that all of the valuation clauses type check.

8.2.5 Referential Transparency, Interference, and Purity

We noted earlier (page 319) that stateless languages like FL are **referentially transparent**. Referential transparency is an important property when reasoning about programs, especially when analyzing and transforming programs.

For example, consider the following program transformations:

T1: `(+` E_a E_a`)` $\longrightarrow$ `(*` `2` E_a`)`

T2: `(+` E_b E_c`)` $\longrightarrow$ `(+` E_c E_b`)`

Under what conditions are such transformations **safe**, i.e., guaranteed to preserve the meaning of a program?[4]

In a referentially transparent language like FL, these two transformations are always safe. In **T1**, E_a always has the same value no matter how many times it is evaluated. In **T2**, reordering E_b and E_c cannot change their values because they are still in the same naming context as before.

However, in a stateful language like FL!, neither of these transformations is always safe. For example, in **T1**, suppose that E_a increments a counter in addition to returning a result. Then `(+` E_a E_a`)` will increment the counter twice, but `(*` `2` E_a`)` will only increment it once. In **T2**, suppose that E_b increments a counter whose value is returned by E_c. Then swapping E_b and E_c changes the value returned by E_c. The problem in these cases is that expressions can depend on the implicit store threaded through their evaluation, so it is generally not safe to replace them by a value or change their relative positions. In particular, an expression can depend on the store by:

- allocating a location in the store (which includes initialization in our semantics),
- reading the value stored at a location, or
- writing a value into a location.

Nevertheless, there are still many situations in which the transformations are safe, even in a stateful language. Let us say that an expression E_1 **interferes** with E_2 when E_1 allocates or writes a store location that is read and/or written by E_2. Then **T1** is safe as long as E_a does not interfere with itself or the rest

[4]For the purposes of this discussion, we choose to treat all errors and divergence as observationally equivalent. That is, we do not care if a transformation changes the error signaled by a program or changes an error-signaling program to a diverging one (or vice versa).

of the program and **T2** is safe as long as E_b or E_c do not interfere with each other. Classical compiler optimizations like code motion, common subexpression elimination, and dead code removal require reasoning about the interference between expressions.

A particularly simple form of non-interference involves expressions that do not depend on the store at all. An expression is **pure** when it does not allocate, read, or write any store locations. A pure expression does not interfere with any other expression, and so it can be treated as if it were in a referentially transparent language. For instance, it is safe to replace a pure expression by an expression having the same value or to move a pure expression to a different position in the same naming context.

Neither interference nor purity is a computable property. However, there are conservative approximations to these properties that are computable. For example, a common syntactic technique for approximating purity is to observe the following in a language with cells:

- variable references and abstractions (`lambda` expressions) do not depend on the store and so are syntactically pure;
- conditionals, `let` expressions, `pair` expressions, and primitive applications (except those involving cell primitives) are syntactically pure if all their subexpressions are syntactically pure;
- all other expressions, including primitive applications of cell primitives and procedure applications, are assumed to be impure.

Expressions that are pure by these rules are called **syntactic values**. We shall use this notion later in our discussion of polymorphic types, type reconstruction, and abstract types (Chapters **??** and 15). Chapter 16 will present a more flexible mechanism for statically determining the side effects (and therefore interference properties) an expression may have.

▷ **Exercise 8.5** Show that the store-based definition of *Computation* is pointed. ◁

▷ **Exercise 8.6**

a. Prove that the first four equalities in Figure 8.14 hold when $Computation = Expressible$.

b. Prove that all nine equalities in Figure 8.14 hold when $Computation = Store \to (Expressible \times Store)$. ◁

▷ **Exercise 8.7**

a. What is the value of `(rec a a)` under the call-by-value denotational semantics for FLK in the previous chapter?

b. What is the value of `(rec a a)` under the operational semantics for FLK!?

c. What is the value of `(rec a a)` under the denotational semantics for FLK!?

d. Explain any discrepancy in your answers to the first three parts of this question.

◁

▷ **Exercise 8.8** The FL! language definition includes a simple immutable data structure called the *pair*. In this problem, we introduce a mutable pair. Mutable pairs are a simple kind of mutable structure similar to the mutable records found in many imperative languages. (See Section 10.1.4 for a discussion of mutable data structures.)

Suppose the FLK! language is extended in the following way:

E ::= ...
| `(mpair` E_l E_r`)` | `(mfst` E_{mp}`)` | `(msnd` E_{mp}`)`
| `(set-mfst!` E_{mp} E_l`)` | `(set-msnd!` E_{mp} E_r`)`

The new constructs have the following informal semantics:

- `(mpair` E_l E_r`)` creates a new mutable pair value with two fields called *mfst* and *msnd*. The values of E_l and E_r are stored in the *mfst* and *msnd* fields, respectively.
- If E_{mp} evaluates to a mutable pair, then `(mfst` E_{mp}`)` returns the content of the *mfst* field of the pair. Otherwise, `mfst` produces an error. Similarly for `msnd`.
- If E_{mp} evaluates to a mutable pair, then `(set-mfst!` E_{mp} E_l`)` mutates the mutable pair so that the *mfst* field contains the value of E_l. If E_{mp} evaluates to anything else, or if evaluating E_l gives an error, then `set-mfst!` generates an error. Similarly for `set-msnd!`.

For example, here are some expressions involving mutable pairs:

```
(let ((foo (mpair 1 2)))
  (begin
    (set-mfst! foo 6)
    (+ (mfst foo) (msnd foo)))) ⟶eval 8

(let ((bar (mpair 8 (mpair 4 3))))
  (begin
    (set-mfst! bar (msnd bar))
    (set-msnd! (msnd bar) (mfst (mfst bar)))
    (+ (mfst (msnd bar)) (msnd (msnd bar))))) ⟶eval 8
```

a. Extend the denotational semantics of FLK! to handle `mpair`, `mfst`, `msnd`, `set-mfst!`, and `set-msnd!`.

   i. Describe any additions or modifications you make to the semantic domains of FLK!.

ii. Give valuation clauses for the five constructs. (You should not have to modify any of the existing valuation clauses.)

iii. Define any auxiliary functions necessary for your valuation clauses.

b. Consider the following potential desugaring for `mpair`, `mfst`, and `set-mfst!` (`msnd` and `set-msnd!` would be handled similarly):

$$\mathcal{D}[\![\texttt{(mpair } E_l \ E_r\texttt{)}]\!] = \texttt{(pair (cell } \mathcal{D}[\![E_l]\!]\texttt{) (cell } \mathcal{D}[\![E_r]\!]\texttt{))}$$

$$\mathcal{D}[\![\texttt{(mfst } E_{mp}\texttt{)}]\!] = \texttt{(cell-ref (left } \mathcal{D}[\![E_{mp}]\!]\texttt{))}$$

$$\mathcal{D}[\![\texttt{(set-mfst! } E_{mp} \ E_l\texttt{)}]\!] = \texttt{(cell-set! (left } \mathcal{D}[\![E_{mp}]\!]\texttt{) } \mathcal{D}[\![E_l]\!]\texttt{)}$$

Is this desugaring consistent with the semantics of mutable pairs? If it is, explain why; if not, show an expression whose meaning differs under this desugaring. ◁

▷ **Exercise 8.9** A common problem when working with state is data consistency. For example, consider a database application that manages the accounts of a bank. Transferring an amount of money between two accounts implies subtracting the amount from the first account and adding it to the second one. If we transfer money only between accounts of the same bank, the total amount of money present in all the accounts should remain the same. However, if something bad occurs between the subtraction and the addition (e.g., a system crash), a certain amount might simply vanish! To prevent this, in database programming, all modifications to the database are required to occur within a *transaction*.

Intuitively, a transaction is a series of modifications to a database that become permanent only when the transaction is successfully terminated (the technical term is *committed*). If the user decides to *abort* (i.e., cancel) the transaction, or the system crashes before the transaction is committed, all the modifications are "undone."

Abe Stract, president and CEO of Intrusive Databases, Inc., decides to add transactions to FL!. In Abe's language, the store will act as the database: queries of the database are `cell-ref`s, and modifications are performed by `cell-set!`. It is an error to perform a `cell-set!` when there is no active transaction.

Abe extends the grammar of FLK! by the following clauses:

$E ::= \ldots$	[As before]
| `(begin-transaction!)`	[Begin Transaction]
| `(commit!)`	[Commit Transaction]
| `(abort!)`	[Abort Transaction]

The informal semantics of transactions are:

- `(begin-transaction!)` begins a transaction. The transaction continues until either a `commit!` or an `abort!` is encountered — it is an error if the program ends and a transaction has not been ended or aborted.
- `(commit!)` successfully terminates the current transaction. It is an error if no transaction is in progress.

- `(abort!)` ends the current transaction and undoes all of its modifications. It is an error if no transaction is in progress.

Like `cell-set!`, the three transaction operations all return *unit*.

Transactions may be nested, in which case `abort!` and `commit!` only end the current (innermost) transaction. An `abort!` of a transaction undoes the modifications of the transaction including modifications made by nested transactions.

Here is how Abe might write a transfer between two bank accounts (represented as cells) using transactions:

```
(define transfer
  (lambda (from to amount)
    (begin (begin-transaction!)
           (cell-set! from (- (cell-ref from) amount))
           (cell-set! to   (+ (cell-ref to)   amount))
           (if (< (cell-ref from) 0)
               (begin (abort!)
                      'failed)
               (begin (commit!)
                      'succeeded)))))
```

Here are more examples of the behavior of transactions; we assume the expressions are evaluated in order.

```
(define cell-1 (cell 0))
(define cell-2 (cell 10))

(define inc!
  (lambda (a-cell) (cell-set! a-cell (+ (cell-ref a-cell) 1))))

(define current-state
   (lambda ()
      (list (cell-ref cell-1) (cell-ref cell-2))))
```

`(current-state)` $\xrightarrow[FL!]{}$ $[0, 10]$

```
(begin (begin-transaction!)
       (inc! cell-1)
       (commit!)
       (current-state))
```
$\xrightarrow[FL!]{}$ $[1, 10]$

```
(begin (begin-transaction!)
       (inc! cell-2)
       (abort!)
       (current-state))
```
$\xrightarrow[FL!]{}$ $[1, 10]$

```
(begin (begin-transaction!)
       (inc! cell-1)
       (begin (begin-transaction!)
              (inc! cell-2)
              (abort!))
       (commit!)
       (current-state)) ⟶_{FL!} [2, 10]

(begin (begin-transaction!)
       (inc! cell-1)
       (begin (begin-transaction!)
              (inc! cell-2)
              (commit!))        ;; End inner transaction,
       (abort!)                 ;; but abort! outer transaction.
       (current-state)) ⟶_{FL!} [2, 10]

(begin (begin-transaction!)     ;; commit! returns #u
       (inc! cell-2)
       (commit!)) ⟶_{FL!} unit

(current-state) ⟶_{FL!} [2, 12]
```

Abe also points out some programs that generate errors (each interacts with the database in a completely independent session):

```
(begin-transaction!) ⟶_{FL!} error : transaction − not − terminated

(commit!) ⟶_{FL!} error : no − current − transaction

(let ((a-cell (cell 0)))
   (begin (cell-set! a-cell 5)
          (cell-ref a-cell))) ⟶_{FL!} error : not − in − a − transaction

(let ((a-cell (cell 0)))
  (begin (begin-transaction!)
         (cell-set! a-cell 5)
         (commit!)
         (cell-set! a-cell 7)   ;; commit! ends transaction,
                                ;; so invalid modification
         (cell-ref a-cell))) ⟶_{FL!} error : not − in − a − transaction
```

a. Extend the operational semantics of FLK! (Section 8.2.3) to handle transactions:

   i. Define the configurations, the set of final configurations, the input function, and the output function.

   ii. Provide transition rules for `begin-transaction!`, `commit!`, `abort!`, and `cell-set!`:

b. Modify the denotational semantics of FLK! to handle transactions.

   i. Give the necessary additions or modifications to the semantic domains of FLK!.

ii. Some auxiliary functions used by the FLK! denotational semantics might need to be modified (e.g., as a result of the changes in the semantic domains). Give their new definitions.

iii. Write the valuation clauses for the three new constructs. ◁

▷ **Exercise 8.10** Clark Smarter of the Photocopy Research Center has developed a new backtracking construct for FLK! called `try`:

E ::= ...
| `(try` E_1 E_2`)` [Backtracking]

The informal semantics of `(try` E_1 E_2`)` is as follows: First E_1 is evaluated, and if E_1 returns *true*, then `try` ignores E_2 and returns *true*. If E_1 evaluates to *false*, then the side effects of E_1 are discarded, and the value of `try` is the value of E_2. If the value of E_1 is neither *true* nor *false*, then the `try` expression yields an error. `try` is thus an elementary backtracking construct. It allows the exploration of one alternative, and, if that does not work, restores the initial state and tries a second alternative.

Here's an example of a program that uses `try`:

```
(let ((balance (cell 200)))
  (let ((withdraw (lambda (n)
                    (begin (cell-set! balance
                                      (- (cell-ref balance) n))
                           (> (cell-ref balance) 0)))))
    ; First try to withdraw 250; if that fails, withdraw 10 from
    ; the original balance.
    (begin (try (withdraw 250)
                (withdraw 10))
           (cell-ref balance))))
```

FL› 190

Clark knows the pitfalls of informal semantics. When writing up the documentation for `try`, he decides to give an operational and denotational semantics for his new construct.

a. First Clark tries to find an operational semantics for `try`:

   i. In attempting to give an operational semantics for `try`, Clark realizes that he must extend the configuration space CF, so he adds a new intermediate expression to E. Describe the new intermediate form and its purpose. (Hint: you may want to think about the next part before answering this one.)

   ii. Provide all of the rewrite rules which are necessary to handle the `try` construct.

b. Next Clark wants to find a denotational semantics for `try`. Help Clark by writing the valuation clause that handles the `try` expression.

c. Clark shows his operational and denotational semantics definitions of `try` to language implementor Hardy Ware. Hardy says, "These semantic definitions are all well and good, but implementing `try` efficiently is going to be tough."

i. Explain what Hardy means by describing what difficulties would be encountered in implementing `try` efficiently on physical computers where state-based memory devices implement the binding of locations to values.

ii. Sketch a strategy for implementing `try` that does not require making a copy of the entire store. ◁

8.3 Mutable Variables: FLAVAR!

In FLK!, the only entity that can change over time is the contents of a mutable cell. So-called "variables" are actually constants whose value cannot change during the execution of a program. While mutable cells are sufficient for implementing any state-based program, they are not always convenient to use. Here we explore a variant of FLK! called FLAVAR! in which every variable becomes a mutable entity. We will also revisit the issue of parameter-passing in the context of state by examining four parameter-passing mechanisms for FLAVAR!.

8.3.1 Mutable Variables

In FL!, it can be difficult to modify a program to make a previously constant quantity mutable. For example, suppose an FLK program binds the variable `addresses` to a list of names and addresses. Since both variables and pairs are immutable in FL!, the meaning of `addresses` cannot change during the execution of the program. Suppose that we later decide to modify the program so that it dynamically updates the address list. Then it is necessary to rebind `addresses` to a mutable cell whose contents is a list. Furthermore, we must find all references to `addresses` in the existing program and replace them by `(cell-ref addresses)`.[5]

Most programming languages offer a more convenient way of making such changes: **mutable variables**. A variable is mutable if the value it is bound to can change over time. The variables of FL and FL! are somewhat misnamed, because their values can't vary over time; rather, they are names for constants. In contrast, variables in languages like SCHEME, C, PASCAL, and FORTRAN can have their values changed by assignment during the execution of the program. In these languages, modifying the address program would not require finding and updating all references to `addresses`, because all variables are assignable by default. On the other hand, programs in these languages can be tougher to reason about because it can be hard to determine which variables change

[5] We shall see in Section 17.7 that compilers often perform a program transformation like this called **assignment conversion.**

over time and which do not. This situation can be improved if the languages provide a mechanism for declaring that certain named entities are immutable (e.g., constant declarations).

We have two motivations for studying mutable variables:

- Many real languages support mutable variables.
- Mutable variables shift the way we think about naming. In languages with mutable variables, names do not denote values, but instead denote locations in the store at which values are stored.

8.3.2 FLAVAR!

We will study mutable variables in FLAVAR!, a dialect of FL! that supports assignments to variables. The syntax of FLAVAR! (and its kernel, FLAVARK!) is the same as that for FL! (and its kernel, FLK!) except for the addition of a SCHEME-like `set!` construct:

$E_{FLAVARK!}$ ::= ... [FLK! Expressions]
| `(set!` *I E*`)` [Assignment]

Informally, `(set!` *I E*`)` assigns the value of the expression *E* to the variable named by *I*. For example,

```
(let ((a 3))
  (begin (set! a 4)
         a))
```
$\xrightarrow[FLAVAR!]{}$ *4*

Note the differences between the cell assignment operator, `cell-set!`, and the variable assignment construct, `set!`. The former changes the value of a first-class data value (a cell), while the latter changes the value of a variable (which is not a first-class value). In `(cell-set!` E_1 E_2`)`, E_1 can be any expression that evaluates to a cell, while in `(set!` *I E*`)`, *I* is constrained to be an identifier visible in the current scope. Mutable cells and mutable variables are orthogonal language features. FLAVAR! contains both.

The semantics of FLAVARK! is based on the denotational semantics of FLK! presented in the previous section. We will only note the ways in which the semantics for FLAVARK! differs from that for FLK!. Some of the differences are highlighted in Figure 8.16. The key feature of FLAVARK! is that variables, like mutable cells, are represented as locations in the store. This means that locations are the only entity in the language that can be named; i.e., *Denotable* = *Location*. The association between a name *I* and a value *v* that is represented by a single environment binding in FL! is represented by *two* bindings in FLAVAR!:

$\delta \in \mathit{Denotable} = \mathit{Location}$
$\sigma \in \mathit{Storable} = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$\mathit{val\text{-}to\text{-}storable} : \mathit{Value} \to \mathit{Storable} = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$\mathcal{E}[\![I]\!] = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$$\mathcal{E}[\![\texttt{(set! } I \ E\texttt{)}]\!] = \lambda e\,.\ (\mathit{with\text{-}value}\ \ (\mathcal{E}[\![E]\!]\ e)\ (\lambda v\,.\ (\mathit{with\text{-}denotable}\ \ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\ (\mathit{update}\ l\ \ (\mathit{val\text{-}to\text{-}storable}\ v))))))$$

Figure 8.16: Semantics of mutable variables. The definitions of *Storable*, *val-to-storable*, and the valuation clauses for I and `call` depend on the parameter passing mechanism.

an environment binding between a name I and a location l, and an assignment between l and v. The indirection through l allows the value associated with the name to be changed. The details of how the locations are allocated, how they are looked up, and what values may legally be stored in them are determined by the parameter passing mechanism of the language. We shall discuss several mechanisms shortly.

The other interesting aspect of the FLAVAR! semantics is the valuation clause for `set!`. In `(set!` I E`)`, E is evaluated and stored in the location named by I. The auxiliary function *val-to-storable*, which depends on the definition of *Storable*, is needed to inject the value into the *Storable* domain. Note that in the expression `(set! a a)`, the left and right occurrences of `a` are treated differently. A location is found for the left occurrence, but the value stored at that location is found for the right occurrence. For this reason, the location is called the **L-value** (left value) of the variable, and the value stored at that location is called the **R-value** (right value) of the variable. Determining the R-value associated with an L-value is called **dereferencing** the variable. The notions of L-value and R-value can be extended to expressions. Variables can be viewed as cells in which dereferencing corresponds to automatically performing a `cell-ref` upon every variable reference, and `(set!` I E`)` performs a `cell-set!` of the L-value of I to the R-value of E.

8.3.3 Parameter Passing Mechanisms for FLAVAR!

Parameter passing mechanisms for languages with mutable variables are determined by the domain *Storable*, the function *val-to-storable*, and the valuation clauses for `call` and *I*. Figures 8.17 and 8.18 summarize four parameter passing mechanisms for FLAVAR!. These are explained in the following sections.

$\sigma \in \mathit{Storable} = \mathit{Value}$

$\mathit{val\text{-}to\text{-}storable} = \lambda v\,.\,v$

$$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \lambda e\,.\,(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\,(\mathit{with\text{-}value}\ (\mathcal{E}[\![E_2]\!]\ e)\ (\lambda v\,.\,(\mathit{allocating}\ v\ p)))))$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{fetching}\ l\ \mathit{val\text{-}to\text{-}comp})))$$

Call-by-Value

$\sigma \in \mathit{Storable} = \mathit{Computation}$

$\mathit{val\text{-}to\text{-}storable} = \mathit{val\text{-}to\text{-}comp}$

$$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \lambda e\,.\,(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\,(\mathit{allocating}\ (\mathcal{E}[\![E_2]\!]\ e)\ p)))$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{fetching}\ l\ (\lambda c\,.\,c))))$$

Call-by-Name

Figure 8.17: Parameter passing mechanisms in FLAVAR!, part I.

8.3.3.1 Call-by-value

The CBV mechanism for FLAVAR! is similar to CBV for FL and FLK! except that a procedure call allocates a new location for the argument value and passes this location (rather than the value) to the procedure. Since the meaning of an identifier is a location and not a value, every variable reference requires both a lookup in the environment (to find the location) and a fetch from the store (to dereference the location). In CBV, only elements of the domain *Value* are storable. For example:

$$\begin{array}{rcll} \sigma & \in & \mathit{Storable} & = & \mathit{Memo} \\ \mathit{mm} & \in & \mathit{Memo} & = & \mathit{Computation} + \mathit{Value} \end{array}$$

$\mathit{val\text{-}to\text{-}storable} = \lambda v\,.\,(\mathit{Value} \mapsto \mathit{Memo}\ v)$

$$\begin{aligned} \mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,&(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e) \\ &(\lambda p\,.\,(\mathit{allocating}\ (\mathit{Computation} \mapsto \mathit{Memo}\ (\mathcal{E}[\![E_2]\!]\ e))\ p))) \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\![I]\!] = \lambda e\,.\,&(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I) \\ &(\lambda l\,.\,(\mathit{fetching}\ l \\ &\quad(\lambda \mathit{mm}\,.\ \textbf{matching}\ \mathit{mm} \\ &\quad\quad\triangleright\,(\mathit{Computation} \mapsto \mathit{Memo}\ c) \\ &\quad\quad\ [\!]\ (\mathit{with\text{-}value}\ c \\ &\quad\quad\quad(\lambda v\,.\,(\mathit{sequence}\ (\mathit{update}\ l\ (\mathit{Value} \mapsto \mathit{Memo}\ v)) \\ &\quad\quad\quad\quad(\mathit{val\text{-}to\text{-}comp}\ v)))) \\ &\quad\quad\triangleright\,(\mathit{Value} \mapsto \mathit{Memo}\ v)\ [\!]\ (\mathit{val\text{-}to\text{-}comp}\ v) \\ &\quad\quad\textbf{endmatching}\ )))) \end{aligned}$$

Call-by-Need (Lazy Evaluation)

$$\sigma \in \mathit{Storable} = \mathit{Value}$$

$\mathcal{E} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation}$
$\mathcal{LV} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation}$

$\mathit{val\text{-}to\text{-}storable} = \lambda v\,.\,v$

$$\begin{aligned} \mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,&(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e) \\ &(\lambda p\,.\,(\mathit{with\text{-}location}\ (\mathcal{LV}[\![E_2]\!]\ e)\ p))) \end{aligned}$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{fetching}\ l\ \mathit{val\text{-}to\text{-}comp})))$$

$$\mathcal{LV}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l))))$$

$$\begin{aligned} &\mathcal{LV}[\![E_{other}]\!]\ ;\ \text{where } E_{other} \text{ is not } I \\ &= \lambda e\,.\,(\mathit{with\text{-}value}\ (\mathcal{E}[\![E_{other}]\!]\ e) \\ &\qquad(\lambda v\,.\,(\mathit{allocating}\ v\ (\lambda l\,.\,(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l))))) \end{aligned}$$

Call-by-Reference

Figure 8.18: Parameter passing mechanisms in FLAVAR!, part II.

```
(let ((a 0)
      (f (lambda (x) (+ x x))))
  (f (begin (set! a (+ a 1)) a)))
```
$\xrightarrow[CBV\ \ FLAVAR!]{}$ *2*

```
((lambda (x) 3) (/ 1 0))
```
$\xrightarrow[CBV\ \ FLAVAR!]{}$ *error*

8.3.3.2 Call-by-name

CBN in FLAVAR! is similar to CBN in FL, except that here it is *Storable* (not *Denotable*) that equals *Computation*. The `call` clause indicates that the computation of the argument expression (not its value) is stored at a newly allocated location. In FLAVAR!, computations are functions that accept a store, so the current store is supplied to a computation every time the variable that names it is referenced. If the computation performs a side effect, this side effect will be performed every time the variable is looked up. Consider the following example:

```
(let ((a 0)
      (f (lambda (x) (+ x x))))
  (f (begin (set! a (+ a 1)) a)))
```
$\xrightarrow[CBN\ \ FLAVAR!]{}$ *3*

In the example, calling `f` binds `x` to a location that holds the computation ($\mathcal{E}[\![$`(begin (set! a (+ a 1)) a)`$]\!]\ e_1$), where e_1 is an environment with bindings for `a` and `f`. Each variable reference to `x` within the procedure body `(+ x x)` performs this computation with the current store. So the left reference to `x` increments `a` and returns 1, while the right reference to `x` increments `a` again and returns 2. This behavior illustrates the perils of mixing state with CBN parameter passing.

As in FL, certain computations in FLAVAR! correspond to errors or nontermination. Because such computations are nameable in CBN (by an indirection through the store), procedures can be non-strict:

```
((lambda (x) 3) (/ 1 0))
```
$\xrightarrow[CBN\ \ FLAVAR!]{}$ *3*

8.3.3.3 Call-by-need (Lazy Evaluation)

The presence of state in FLAVAR! suggests a parameter passing mechanism based on the memoization trick introduced in the FL interpreter. That is, a formal parameter name can be bound to a location that originally stores the computation of the argument expression. The first time the parameter is referenced, the computation is performed, but the resulting value is cached at the location and is used on every subsequent reference. Thus, the argument expression is evaluated at most once and is never evaluated at all if the parameter

is never referenced. This mechanism is called **call-by-need** or **lazy evaluation**. Because the acronym CBN is already taken, we will abbreviate call-by-need as CBL (call-by-lazy).

Call-by-need can exhibit the desirable behavior of both CBV and CBN:

```
(let ((a 0)
      (f (lambda (x) (+ x x))))
  (f (begin (set! a (+ a 1)) a)))
```
$\xrightarrow[CBL\ \ FLAVAR!]{}$ *2*

```
((lambda (x) 3) (/ 1 0))
```
$\xrightarrow[CBL\ \ FLAVAR!]{}$ *3*

However, because side effects in argument expressions are performed at the time of lookup rather than at the time of call, CBL can exhibit different behavior from CBV. For example, consider the following expression:

```
(let ((a 0))
  (let ((f (lambda (x)
             (begin (set! a 17)
                    (+ x x)))))
    (f (begin (set! a (+ a 1)) a))))
```

Under CBV, the call to `f` first increments `a` and then binds `x` to a location holding `1`. The assignment of `17` to `a` does not affect `x`, so the result is *2*. However, under CBL, the call to `f` binds `x` to a location that holds the computation of `(begin (set! a (+ a 1)) a)`. This computation is not performed until the first reference of `x`, which occurs *after* `a` has been set to `17`. So CBL returns *36* for this expression.

8.3.3.4 Call-by-reference

So far, all the parameter passing mechanisms we have discussed allocate a new location for every argument. But in the case where the argument expression is a variable reference, there is already a location associated with the variable. This suggests a mechanism that uses the existing location rather than allocating a new one. Such a mechanism is termed **call-by-reference (CBR)**. FORTRAN and PASCAL and are examples of languages that support CBR.

In CBR, there is the question of what to do with an argument that is not a manifest identifier. For example, in the application `(test (+ 1 2))`, the value of `(+ 1 2)` has no associated location. Languages handle this situation in different ways. In PASCAL, it is an error to supply anything but an identifier as a CBR argument. In FORTRAN, however, a new location will be allocated for any argument that is not a manifest identifier. The semantics in Figure 8.18 takes this latter approach. In fact, this is the *only* mechanism for creating

new variables in CBR FLAVAR!. This is a somewhat unrealistic aspect of our language; real CBR languages have special declarations for introducing new variables.

The denotational semantics for CBR models the special handling of variable arguments by providing two valuation functions for expressions: $\mathcal{E}$ and $\mathcal{LV}$. $\mathcal{LV}$ finds the L-value of an expression, while $\mathcal{E}$ finds the R-value of an expression. For an expression that is an identifier, $\mathcal{LV}$ returns the location of that identifier. For any other expression, $\mathcal{LV}$ allocates a new location for the R-value of the expression and returns this location. The key feature of the CBR semantics is that $\mathcal{LV}$ (rather than $\mathcal{E}$) is used to evaluate the operand of a procedure application.

In FLAVAR!, procedure calls are expressions that return results, but in many imperative languages, procedure calls are commands that do not return results. In such languages, CBR is useful as a means of extracting a result from a procedure call. One (or more) arguments to a procedure can be a variable that the procedure uses to communicate the result(s) back to the caller. Here is an example of this idiom in CBR FLAVAR!:

```
(let ((a 0)
      (double (lambda (in out)
                (set! out (+ in in)))))
  (begin
   ;; A is 0 here.
   (double 17 a)
   ;; Now A is 34.
   (+ a 1))) ⟶ 35
```

The `double` procedure takes a numeric argument (`in`) and variable (`out`) for returning the result of doubling `in`. In the example, the variable `a` is used to communicate the result of the doubling operation back to the point of call.

One characteristic of CBR (or any paradigm that allows mutable entities to be passed as arguments) is that two different names may refer to the same location. This situation is known as **aliasing**. Consider the following example:

```
(let ((x 1))
  (let ((test (lambda (a)
                (begin
                  (set! x 20)
                  (+ a x)))))
    (test x))) ⟶ 40
```

Within the call `(test x)`, both `x` and `a` are aliases for the same location, so the assignment to `x` changes `a`. Aliasing is often considered undesirable because it complicates reasoning about programs.

CBR is similar to passing a mutable cell as an argument to a procedure. The difference is that variables are more restricted than cells. A mutable cell is a first-class value: it may be named, passed as an argument to a procedure, returned as a result from a procedure, and stored in any data structure, including another cell. On the other hand, while a variable may be named by an identifier and passed as an argument to a procedure, it cannot be returned as a result from a procedure, and it cannot be stored in a data structure (including another variable). Unlike cells, therefore, variables are not first-class values. Although this restricts the expressive power of variables, it permits variables to be implemented more efficiently than cells. A variable may be allocated on a stack, while cells generally must be allocated from a garbage-collected heap. We will have much more to say about tradeoffs between expressiveness and efficiency when we study pragmatic issues later on.

▷ **Exercise 8.11**

a. Give a translation of call-by-value FLAVARK! into call-by-value FLK!. You do not need to translate `rec`.

b. Give a translation of call-by-reference FLAVARK! into call-by-value FLK!. You do not need to translate `rec`. ◁

Chapter 9

Control

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I —
I took the one less traveled by,
And that has made all the difference.

— The Road Not Taken, st. 4, Robert Frost

"Did he ever return, no he never returned
And his fate is still unlearned"

— MTA, performed by the Kingston Trio,
written by Bess Hawes & Jacqueline Steiner

9.1 Motivation: Control Contexts and Continuations

So far, we have studied two different kinds of contexts important in the evaluation of programming language expressions:

- A naming context that determines the meaning of free variable names within an expression.
- A state context that specifies the time-dependent behavior of mutable entities.

By objectifying both of these contexts as mathematical entities — environments and stores — the denotational approach provides significant leverage for us to investigate the space of language features that depend on these contexts. In the case of naming, environments help us to understand issues like parameter passing, scoping, and inheritance. In the case of state, stores help us to understand issues involving mutable variables and data structures.

There is a third major context that is still missing from our toolbox: a **control** context. Informally, control describes the path taken by a programmer's eyes and fingertips when hand-simulating the code in a listing. For example, when simulating a `while` or `for` loop in an imperative language, it is often necessary to refocus attention on the beginning of the loop after the end of the loop code is reached. Conditional expressions and procedure calls are other simple examples of control constructs that we have seen.

What does it mean for expressions to have a control *context*? As an example, consider the following FL! expression:

```
(let ((square (lambda (x) (* x x))))
  (+ (square 5) (* (+ 1 2) (square 5))))
```

There are two different occurrences of the `(square 5)` expression. What is the difference between them? Both are evaluated in the same environment and the same store, so they are guaranteed to yield the same value. What distinguishes them is how their value is used by the rest of the program. Reading from left to right, the first `(square 5)` returns 25 to a process that is collecting the first of two arguments to the procedural value of `+`. The second `(square 5)` yields its result to a process that is collecting the second of two arguments to the procedural value of `*`; this, in turn, is a subtask of the process that is collecting the second of two arguments to `+`, which itself is a subtask of the process that is waiting for the answer to the entire `let` expression. What distinguishes the occurrences of `(square 5)` is their control context: the part of the computation that remains to be done after the expression is evaluated.

The denotational descriptions we have employed so far have not explicitly represented the notion of "the rest of the computation." A denotational semantics without an explicit control model is said to be a **direct semantics**. A direct semantics for a programming language cannot deal elegantly with interruptions of the normal flow of control of a program. As long as valuation clauses are recursive in the obvious way, the flow of control in the clauses has no choice but to follow the structure of the program's parse tree.

A simple example of the limitation of direct semantics can be seen in its clumsy handling of error conditions in the languages that we have already encountered. An error is detected in one part of the semantics, and every other

part of the semantics must be able to cope with the possibility that some subexpression has produced an error instead of a normal result. This approach to error checking does not capture the intuition that a computation encountering an error immediately aborts without further processing. Abstractions like *with-value* help to hide this error checking, but they do not remove it. Indeed, interpreters based on the direct semantics of FL and its variants expend considerable effort performing such checks.

More generally, a direct semantics cannot easily explain constructs that interrupt the "normal" flow of control:

- *non-local exits* as provided by C's `break` and `continue` or COMMON LISP's `throw` and `catch`.
- *unrestricted jumps* permitted in numerous languages via `goto`.
- sophisticated *exception handling* as seen in CLU, ML, COMMON LISP, DYLAN, and JAVA.
- *coroutines* such as iterators in CLU and communicating sequential processes in many languages, notably OCCAM and even JAVA (JCSP).
- *backtracking,* which is used to model nondeterminism, e.g., to search a tree of possibilities, as in PROLOG and other logic programming languages.

In each of these cases, a program phrase does not simply return some value and/or an updated store, but instead bypasses the control context that invoked it and transfers control to some other place in the program.

The notion of **continuation** addresses this problem and provides a mathematical model of such transfers of control. A continuation is an entity that explicitly represents the "rest" of some computation. In implementation terms, it corresponds to the part of the machine state that comprises the current configuration of the runtime stack, together with a return address that specifies what code to run when the current computation returns a value. The continuation corresponding to the textually subsequent code in a program is usually referred to as the **normal continuation**. Many control constructs achieve their effect by substituting some other continuation for the normal one.

This chapter shows how continuations simplify the descriptions of the languages we have studied so far and allow the modeling of advanced control features in these languages. Be forewarned that control constructs are notoriously hard to think about. Even though many of the formal descriptions of control constructs are surprisingly concise, this does not imply that they are proportionately easier to understand. The often convoluted nature of control can lead

the reader into mental gymnastics that are likely to leave the brain a little bit sore at first. Luckily, with sufficient practice, the concepts can begin to seem natural.

To help build up some intuitions about continuations, we will first discuss how to achieve some sophisticated control behavior using only first class procedures. Then we will be better prepared to understand the use of continuations in denotational definitions.

9.2 Using Procedures to Model Control

We said before that continuations represent the rest of a computation. In a functional language, the continuation for an expression E is "waiting for" the value of E. It is therefore natural to think of continuations in a functional language as being procedures of one argument. For example, in the FL expression

```
(let ((square (lambda (x) (* x x))))
  (+ (square 5) (* (+ 1 2) (square 5))))
```

the continuation of the first `(square 5)` might be thought of as

```
(lambda (v1) (+ v1 (* (+ 1 2) (square 5))))
```

and the continuation for the second `(square 5)` might be thought of as

```
(lambda (v2) (+ 25 (* 3 v2)))
```

The above approximations indicate that operands to an FL application are evaluated in left-to-right order. When the first call to `square` is being evaluated, the second argument to `+` is the unevaluated `(* (+ 1 2) (square 5))`. But by the time the second call to `square` is evaluated, the first `(square 5)` has been evaluated to `25` and the `(+ 1 2)` argument has been evaluated to `3`.

Even in languages that do not support mutation, continuations require a computation to be viewed in a purely sequential way; some expressions are evaluated "before" other expressions. In fact, it is really control, not state, that must be linearly threaded through a sequential computation. State is just a piece of information carried along by the control in its linear walk. This separation of control and state makes it easier to think about sophisticated control constructs like backtracking, where a computation may revert to a previous state even though it is progressing in time.

First-class procedures are powerful enough to implement some fancy control behavior. In this section, we show how first-class procedures can be used to implement procedures returning multiple values, non-local exits, and coroutines.

9.2.1 Multiple-value Returns

It is often useful for a procedure to return more than one result. A classic example of the utility of multiple-value returns concerns integer division and remainder. Languages often provide two primitives for these operations even though the same algorithm computes both. It would make more sense to have a single operation that returns two values.

As another example, suppose that we want to write an FL program that, given a binary tree with integers as leaves, computes both the depth and the sum of the leaves in the tree and returns their product. One approach is to apply two different procedures to the tree and combine the results as in Figure 9.1. Notice that `depth*sum`$_1$ requires two walks over the given tree.

```
(define depth*sum1
  (lambda (tr)
    (letrec ((depth (lambda (tree)
                      (if (leaf? tree)
                          0
                          (+ 1 (max (depth (tree-left tree))
                                    (depth (tree-right tree)))))))
             (sum (lambda (tree)
                    (if (leaf? tree)
                        tree
                        (+ (sum (tree-left tree))
                           (sum (tree-right tree)))))))
      (* (depth tr) (sum tr)))))
```

Figure 9.1: The first version of `depth*sum` performs two tree traversals.

A procedure that returns multiple values allows one to perform the computation in a single tree walk. A simple method of doing this is to return a pair at each node of the tree as in Figure 9.2. However, the bundling and unbundling of values makes this approach to multiple values messy and hard to read.

An alternate approach to returning multiple values is to use first-class procedures. If procedure M is supposed to return multiple values, we can modify it to take an extra argument R[1], called the **receiver**. The receiver is a procedure that expects the multiple values as its arguments and will combine them into some result. M returns its results by calling R on them. We have already seen numerous examples of this strategy in metalanguage functions and interpreter procedures (e.g., *with-* functions have this form). Figure 9.3 shows how to apply this idea to our example.

[1]By convention, the extra argument usually comes last.

```
(define depth*sum2
  (lambda (tr)
    (letrec ((inner
              (lambda (tree)
                (if (leaf? tree)
                    (cons 0 tree)
                    (let ((depth&sum1 (inner (tree-left tree)))
                          (depth&sum2 (inner (tree-right tree))))
                      (cons (+ 1 (max (car depth&sum1)
                                      (car depth&sum2)))
                            (+ (cdr depth&sum1) (cdr depth&sum2)))))))))
      (let ((depth&sum (inner tr)))
        (* (car depth&sum) (cdr depth&sum))))))
```

Figure 9.2: The second version of `depth*sum` uses pairs to return multiple values.

```
(define depth*sum3
  (lambda (tr)
    (letrec ((inner
              (lambda (tree receiver)
                (if (leaf? tree)
                    (receiver 0 tree)
                    (inner
                      (tree-left tree)
                      (lambda (depth1 sum1)
                        (inner (tree-right tree)
                               (lambda (depth2 sum2)
                                 (receiver (+ 1 (max depth1 depth2))
                                           (+ sum1 sum2))))))))))
      (inner tr *))))
```

Figure 9.3: The third version of `depth*sum` passes multiple values to procedural continuations.

This style of code can be difficult to read. The `receiver` argument to the `inner` procedure acts as a continuation encoding what computation needs to be performed on the two values that `inner` "returns." For example, the call `(inner tr *)` starts off the process by applying `inner` to the tree `tr` with a receiver `*` that will take the two results and return their product.

Even though the receiver is an argument, it is typical to ignore its argument status and view it as a different entity when reading a call like `inner`. So `(inner` E_1 `(lambda (`I_1 I_2`)` E_2`))` can be read as "Call `inner` on E_1 and apply the procedure `(lambda (`I_1 I_2`)` E_2`)` to the results" or "Evaluate E_2 in an environment where I_1 and I_2 are bound to the two results of applying `inner` to E_1." Note that these readings treat `inner` as a procedure of one argument that returns two results, not a procedure of two arguments. Viewing continuation argument(s) as different entities from other arguments is important for getting a better working understanding of them.

Unlike the other two approaches, using a receiver forces us to choose a particular order for examining the branches of the binary tree. The main advantage of a receiver is that it allows the multiple returned values to be named using the standard naming construct, `lambda`. It is not necessary to invent a new syntax for naming intermediate values: `lambda` suffices.

As a concrete example, consider the following application of `depth*sum`$_3$:

```
(depth*sum3 '((5 7) (11 (13 17))))
```

An operational trace of the evaluation of this expression appears in Figures 9.4 and 9.5. Here, a tree node is represented by a list of the left and right subtrees, while a leaf is represented by an integer. Note how the continuation argument to `inner` acts like a stack that keeps track of the pending operations.

9.2.2 Non-local Exits

A continuation represents all the pending operations that are waiting to be done after the current operation. When continuations are implicit, the computation can only terminate successfully when all of the pending operations have been done. Yet we sometimes want a computation buried deep in pending operations to terminate immediately with a result or at least circumvent a number of pending operations. We can achieve these so-called **non-local exits** by using explicit procedure objects representing continuations.

For example, consider the task of multiplying a list of numbers. Figure 9.6 shows the natural recursive solution to this problem. E.g.,

```
(product-of-list1 (list 2 4 8)) ⟶eval 64
```

```
(depth*sum3 ((5 7) (11 (13 17))))
⇒ (inner ((5 7) (11 (13 17))) *)
⇒ (inner (5 7) (lambda (d1 s1)
                 (inner (11 (13 17)) (lambda (d2 s2)
                                       (* (+ 1 (max d1 d2))
                                          (+ s1 s2))))))
⇒ (inner 5 (lambda (d3 s3)
              (inner 7 (lambda (d4 s4)
                         ((lambda (d1 s1)
                            (inner (11 (13 17))
                              (lambda (d2 s2)
                                (* (+ 1 (max d1 d2)) (+ s1 s2)))))
                          (+ 1 (max d3 d4))
                          (+ s3 s4))))))
⇒ (inner 7 (lambda (d4 s4)
              ((lambda (d1 s1)
                 (inner (11 (13 17)) (lambda (d2 s2)
                                       (* (+ 1 (max d1 d2))
                                          (+ s1 s2)))))
               (+ 1 (max 0 d4))
               (+ 5 s4))))
⇒ ((lambda (d1 s1)
     (inner (11 (13 17)) (lambda (d2 s2)
                           (* (+ 1 (max d1 d2)) (+ s1 s2)))))
   (+ 1 (max 0 0))
   (+ 5 7))
⇒ (inner (11 (13 17)) (lambda (d2 s2)
                        (* (+ 1 (max 1 d2)) (+ 12 s2))))
⇒ (inner 11 (lambda (d5 s5)
              (inner (13 17) (lambda (d6 s6)
                               ((lambda (d2 s2)
                                  (* (+ 1 (max 1 d2)) (+ 12 s2)))
                                (+ 1 (max d5 d6))
                                (+ s5 s6))))))
```

Figure 9.4: Stylized operational trace of a procedural implementation of multiple-value return, part 1.

```
⇒ (inner (13 17) (lambda (d6 s6)
                   ((lambda (d2 s2)
                      (* (+ 1 (max 1 d2)) (+ 12 s2)))
                    (+ 1 (max 0 d6))
                    (+ 11 s6))))
⇒ (inner 13 (lambda (d7 s7)
               (inner 17 (lambda (d8 s8)
                           ((lambda (d6 s6)
                              ((lambda (d2 s2)
                                 (* (+ 1 (max 1 d2)) (+ 12 s2)))
                               (+ 1 (max 0 d6))
                               (+ 11 s6)))
                            (+ 1 (max d7 d8))
                            (+ s7 s8))))))
⇒ (inner 17 (lambda (d8 s8)
               ((lambda (d6 s6)
                  ((lambda (d2 s2)
                     (* (+ 1 (max 1 d2)) (+ 12 s2)))
                   (+ 1 (max 0 d6))
                   (+ 11 s6)))
                (+ 1 (max 0 d8))
                (+ 13 s8))))))
⇒ ((lambda (d6 s6)
      ((lambda (d2 s2)
         (* (+ 1 (max 1 d2)) (+ 12 s2)))
       (+ 1 (max 0 d6))
       (+ 11 s6)))
    (+ 1 (max 0 0))
    (+ 13 17))))))
⇒ ((lambda (d2 s2)
      (* (+ 1 (max 1 d2)) (+ 12 s2)))
    (+ 1 (max 0 1))
    (+ 11 30))
⇒ (* (+ 1 (max 1 2))
      (+ 12 41))
⇒ 159
```

Figure 9.5: Stylized operational trace of a procedural implementation of multiple-value return, part 2.

```
(define product-of-list1
    (lambda (nums)
        (if (null? nums)
            1
            (* (car nums) (product-of-list1 (cdr nums))))))
```

Figure 9.6: A first cut at the product of a list.

Figure 9.7 shows a continuized version of `product-of-list`$_1$. The behavior is exactly the same; we have just made the continuations explicit. For an empty list, `product-of-list`$_1$ continues with the value 1. For a non-empty list, we compute the product of the tail of the list passing along a new continuation that passes the product of the list tail and the current element to the current continuation.

```
(define product-of-list2
  (lambda (nums cont)
    (if (null? nums)
        (cont 1)
        (product-of-list2 (cdr nums)
                          (lambda (v)
                            (cont (* v (car nums))))))))
```

Figure 9.7: A continuized procedure for computing the product of a list.

Notice that `product-of-list`$_1$ and `product-of-list`$_2$ dutifully multiply all the elements of the list even if it contains a zero element. This is a waste since the answer is known to be 0 the moment a 0 is encountered. There is no need to look at any other list elements or to perform any more multiplications. `product-of-list`$_3$ in Figure 9.8 performs this optimization.

To accomplish a non-local exit, `product-of-list`$_3$ distinguishes the continuation passed to the initial call from continuations generated by recursive calls. The escape continuation is kept in `final-cont`. The local recursive procedure `prod` behaves like `product-of-list`$_2$ except that it jumps immediately to `final-cont` upon encountering a 0, thus avoiding unnecessary recursive calls by-passing all pending multiplications.

As a more complicated example, consider the pattern matching program for FL presented in Section 6.2.4.3. The core of the program is the `match-with-dict` procedure in Figure 9.9, where we have unraveled the failure abstractions to make the present discussion more concrete.

As written, `match-with-dict` performs a left-to-right depth-first walk simul-

```
(define product-of-list3
  (lambda (nums final-cont)
    (letrec ((prod
              (lambda (nums normal-cont)
                (if (null? nums)
                    (normal-cont 1)
                    (let ((thisnum (car nums)))
                      (if (= thisnum 0)
                          (final-cont 0)
                          (prod (cdr nums)
                                (lambda (val)
                                  (normal-cont
                                    (* val thisnum))))))))))
      (prod nums final-cont))))
```

Figure 9.8: Computing the product of list, exiting as soon as the answer is apparent.

```
(define match-sexp
  (lambda (pat sexp)
    (match-with-dict pat sexp (dict-empty))))

(define match-with-dict
  (lambda (pat sexp dict)
    (cond ((eq? dict '*failed*)       ; Propagate failures
           '*failed*)
          ((null? pat)
           (if (null? sexp)
               dict                   ; Pat and sexp both ended
               '*failed*))            ; Pat ended but sexp didn't
          ((null? sexp) '*failed*)    ; Sexp ended but pat didn't
          ((pattern-constant? pat)
           (if (sym=? pat sexp) dict '*failed*))
          ((pattern-variable? pat)
           (dict-bind (pattern-variable-name pat) sexp dict))
          (else (match-with-dict (cdr pat)
                                 (cdr sexp)
                                 (match-with-dict (car pat)
                                                  (car sexp)
                                                  dict))))))
```

Figure 9.9: Core of the pattern matching program.

taneously over the `pat` and `sexp` trees. It carries along a dictionary representing bindings for variables that have already been matched. Failure is represented in a rather ad hoc manner by replacing the dictionary with the symbol `*failed*`. Since failure may occur deep in the tree where many pending matches are waiting to be performed, each call of `match-with-dict` must check for and propagate failure tokens that appear as the dictionary argument.

It would be more desirable to handle failures by by-passing all the pending activations and simply returning the symbol `*failed*` as the value of `match-sexp`. This effect can be achieved by passing two extra arguments to `match-with-dict`: a success continuation and a failure continuation. A success continuation is a procedure of one argument, a dictionary, that continues a thus-far successful match with the given dictionary. A failure continuation is a procedure of no arguments that effectively returns `*failed*` for the initial call to `match-with-dict`. It is necessary to package up both continuations so that the program has the option of ignoring one. This strategy is implemented in Figure 9.10. Note in the final clause of the `cond`, `match-inner` works from the outside in while `match-with-dict` works from the inside out. This explains why the calls to `car` and `cdr` appear differently in the two programs, even though both walk the tree in a left-to-right depth-first manner.

In the modified version of the pattern matcher, the interface to `match-sexp` would be cleaner if it took success and failure continuations as well. Then we could more easily specify the behavior we want in these cases.

```
(define match-sexp
  (lambda (pat sexp succeed fail)
    (match-inner pat sexp (dict-empty) succeed fail)))

(match-sexp '((? a) (? a)) '(x x)
            (lambda (dict) #t)
            (lambda () #f))
```

$\xrightarrow[FL!]{}$ *true*

```
(match-sexp '((? a) (? a)) '(x y)
            (lambda (dict) #t)
            (lambda () #f))
```

$\xrightarrow[FL!]{}$ *false*

▷ **Exercise 9.1**

a. Modify the code in `product-of-list`$_3$ to return an error symbol if there is a non-integer element in the list.

b. Suppose the final continuation of `product-of-list`$_3$ must receive an integer value. How would you handle errors? Rewrite `product-of-list`$_3$ to incorpo-

```
(define match-sexp
  (lambda (pat sexp)
    (match-inner pat
                 sexp
                 (dict-empty)
                 (lambda (dict) dict)
                 (lambda () '*failed*))))

(define match-inner
  (lambda (pat sexp dict succeed fail)
    (cond ((null? pat)
           (if (null? sexp)
               (succeed dict)         ; Pat and sexp both ended
               (fail)))               ; Pat ended but sexp didn't
          ((null? sexp) (fail))       ; Sexp ended but pat didn't
          ((pattern-constant? pat)
           (if (sym=? pat sexp)
               (succeed dict)
               (fail)))
          ((pattern-variable? pat)
           (succeed
             (dict-bind (pattern-variable-name pat) sexp dict)))
          (else (match-inner (car pat)
                             (car sexp)
                             dict
                             (lambda (car-dict)
                               (match-inner (cdr pat)
                                            (cdr sexp)
                                            car-dict
                                            succeed
                                            fail))
                             fail)))))
```

Figure 9.10: A version of the pattern matcher that uses success and failure continuations.

rate your changes and show a sample call. ◁

9.2.3 Coroutines

Coroutining is a situation in which control jumps back and forth between conceptually independent processes. The most common version is producer/consumer coroutines, where a consumer process transfers control to a producer process when it wants the next value generated by the producer, and the producer returns control to the consumer along with the value. The standard example of this kind of coroutine is a compiler front end in which a parser requests tokens from the lexical scanner.

Here, we will show how simple producer/consumer coroutines can be implemented by using first-class procedures to represent control. The stream notion we will see in Chapter 10 (beginning on page 431) is an alternate technique for implementing such coroutines.

We represent a producer as a procedure that takes a consumer as its argument and hands that consumer the requested value along with the next producer. We represent a consumer as a procedure that takes a value and producer, and either returns or calls the producer on the next consumer.

For example, suppose `(count-from` n`)` makes a producer which generates the (conceptually infinite) increasing sequence of integers beginning with n, and `(add-first` m`)` makes a consumer that adds up the first m elements of the producer it's attached to. Then `((count-from 3) (add-first 5))` should return the sum of the integers from 3 to 7, inclusive. This example,coded in FL, is in Figure 9.11.

9.3 A Standard Semantics of FL!

To handle state in our semantics, we took the idiom of single-threading a store through a computation and made it part of the computational model. Similarly, we will handle control in our semantics by embedding in our computational model the idiom of passing continuations through a computation. The strategy of capturing common programming idioms in a semantic framework — or any language — is a powerful idea that lies at the foundation of programming language design. Indeed, languages can be considered expressive to the extent that they relieve the programmer of managing the details of common programming idioms.

Together, environments, stores, and continuations are sufficiently powerful to model most programming languages. As noted earlier, a semantics that uses

```
(define (count-from num)
  (letrec ((new-producer
            (lambda (n)
              (lambda (consumer)
                (consumer n (new-producer (+ n 1)))))))
    (new-producer num)))

(define (add-first count)
  (letrec ((new-consumer
            (lambda (c)
              (lambda (value next-producer)
                (if (= c 0)
                    0
                    (+ value (next-producer
                              (new-consumer (- c 1)))))))))
    (new-consumer count)))

;; Add up the 5 consecutive integers starting at 3
((count-from 3) (add-first 5)) ⟶FL 25
```

Figure 9.11: An example producer/consumer example.

only environments and stores is called a **direct semantics**. A semantics that adds continuations to a direct semantics is called a **standard semantics**, since most denotational definitions are written in this style. A standard semantics also implies particular conventions about the signatures of valuation functions. One advantage of standard semantics is that following a set of conventions simplifies the comparison of different programming languages defined by standard semantics. We already saw this kind of advantage when we studied parameter passing and scoping. Comparing different approaches was facilitated by the fact that the styles of the denotational definitions we were comparing were similar.

Now that we've built up some intuitions about continuations, it's time to model continuations explicitly in our denotational definitions. Figures 9.12–9.14 present the standard semantics for FLK!. *The definition given in the figures is just an alternate way to write the same semantics that we gave before.* In fact, there are mechanical transformations that could transform the denotational definition from the previous chapter into the definition in Figures 9.12–9.14.[2]

We introduce a standard semantics for FLK! because it is a much more powerful tool for studying control features than the direct semantics. In fact,

[2]Section 17.9 presents a mechanical transformation of FLAVAR! programs into **continuation passing style**.

$\gamma \in \mathit{Cmdcont} = \mathit{Store} \to \mathit{Answer}$
$k \in \mathit{Expcont} = \mathit{Value} \to \mathit{Cmdcont}$
$j \in \mathit{Explistcont} = \mathit{Value}^* \to \mathit{Cmdcont}$
$\mathit{Answer} =$ language dependent ; *Typically Expressible*
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Expcont} \to \mathit{Cmdcont}$
$x \in \mathit{Expressible} = (\mathit{Value} + \mathit{Error})_\perp$; *As before*
$v \in \mathit{Value} =$ language dependent
$y \in \mathit{Error} = \mathit{Sym}$; *Modified*

New auxiliaries:

top-level-cont : *Expcont*
$= \lambda v\,.\ \lambda s\,.\ (\mathit{Value} \mapsto \mathit{Expressible}\ v)$; *Assume Answer = Expressible*

error-cont : *Error* $\to$ *Cmdcont*
$= \lambda I\,.\ \lambda s\,.\ (\mathit{Error} \mapsto \mathit{Expressible}\ I)$; *Assume Answer = Expressible*

test-boolean : (*Bool* $\to$ *Cmdcont*) $\to$ *Expcont*
$= \lambda f\ .\ (\lambda v\,.$ **matching** v
$\triangleright\, (\mathit{Bool} \mapsto \mathit{Value}\ b) \llbracket\ (f\ \ b)$
$\triangleright$ **else** (*error-cont* `non-boolean`)
endmatching)

Similarly for:
test-procedure : (*Procedure* $\to$ *Cmdcont*) $\to$ *Expcont*
test-location : (*Location* $\to$ *Cmdcont*) $\to$ *Expcont*
etc.

ensure-bound : *Binding* $\to$ *Expcont* $\to$ *Cmdcont*
$= \lambda \beta k\,.$ **matching** β
$\triangleright\, (\mathit{Denotable} \mapsto \mathit{Binding}\ v) \llbracket\ (k\ \ v)$; *Assume CBV*
$\triangleright\, (\mathit{Unbound} \mapsto \mathit{Binding}\ \mathit{unbound}) \llbracket$ (*error-cont* `unbound-variable`)
endmatching

Similarly for:
ensure-assigned : *Assignment* $\to$ *Expcont* $\to$ *Cmdcont*
ensure-value : *Expressible* $\to$ *Expcont* $\to$ *Cmdcont*

Figure 9.12: Semantic algebras for standard semantics of FLK!.

$\mathcal{TL} : \text{Exp} \to \mathit{Answer}$; *Assume Answer = Expressible*
$\mathcal{E} : \text{Exp} \to \mathit{Environment} \to \mathit{Expcont} \to \mathit{Cmdcont}$
$\mathcal{E}^* : \text{Exp}^* \to \mathit{Environment} \to \mathit{Explistcont} \to \mathit{Cmdcont}$
$\mathcal{L} : \text{Lit} \to \mathit{Value}$; *Defined as usual*
$\mathcal{Y} : \text{Symlit} \to \mathit{Sym}$; $Y \in$Symlit are symbolic literals

$$\mathcal{TL}[\![E]\!] = (\mathcal{E}[\![E]\!]\ \mathit{empty\text{-}env}\ \mathit{top\text{-}level\text{-}cont}\ \mathit{empty\text{-}store})$$

$$\mathcal{E}[\![L]\!] = \lambda ek\,.\ (k\ \mathcal{L}[\![L]\!])$$

$$\mathcal{E}[\![I]\!] = \lambda ek\,.\ (\mathit{ensure\text{-}bound}\ (\mathit{lookup}\ e\ I)\ k)$$

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda ek_1\,.\ (k_1\ (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda \delta k_2\,.\ (\mathcal{E}[\![E]\!]\ [I:\delta]e\ k_2))))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\mathit{test\text{-}procedure}\ (\lambda p\,.\ (\mathcal{E}[\![E_2]\!]\ e\ (\lambda v\,.\ (p\ v\ k))))))$$

$$\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\mathit{test\text{-}boolean}\ (\lambda b\,.\ \textbf{if}\ b\ \textbf{then}\ (\mathcal{E}[\![E_2]\!]\ e\ k)\ \textbf{else}\ (\mathcal{E}[\![E_3]\!]\ e\ k)\ \textbf{fi})))$$

$$\mathcal{E}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\,.\ (\mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\,.\ (k\ (\mathit{Pair} \mapsto \mathit{Value}\ \langle v_1, v_2\rangle))))))$$

$$\mathcal{E}[\![\texttt{(cell}\ E\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E]\!]\ e\ (\lambda vs\,.\ (k\ (\mathit{Location} \mapsto \mathit{Value}\ (\mathit{fresh\text{-}loc}\ s))\ (\mathit{assign}\ (\mathit{fresh\text{-}loc}\ s)\ v\ s))))$$

$$\mathcal{E}[\![\texttt{(begin}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ e\ (\lambda v_{ignore}\,.\ (\mathcal{E}[\![E_2]\!]\ e\ k)))$$

$$\mathcal{E}[\![\texttt{(primop}\ O\ E^*\texttt{)}]\!] = \lambda ek\,.\ (\mathcal{E}^*[\![E^*]\!]\ e\ (\lambda v^*\,.\ (\mathcal{P}_{FLK!}[\![O]\!]\ v^*\ k)))$$

$$\mathcal{E}[\![\texttt{(error}\ I\texttt{)}]\!] = \lambda ek\,.\ (\mathit{error\text{-}cont}\ I)$$

$$\mathcal{E}^*[\![[\,]]\!] = \lambda ej\,.\ (j\ [\,]_{\mathit{Value}})$$

$$\mathcal{E}^*[\![E_{\mathit{first}}\ .\ E_{\mathit{rest}}{}^*]\!] = \lambda ej\,.\ (\mathcal{E}[\![E_{\mathit{first}}]\!]\ e\ (\lambda v\,.\ (\mathcal{E}^*[\![E_{\mathit{rest}}]\!]\ e\ (\lambda v^*\,.\ (j\ v\ .\ v^*)))))$$

Figure 9.13: Valuation clauses for standard semantics of FLK!, Part I.

$\mathcal{P}_{FLK!}$: Primop $\rightarrow$ *Value** $\rightarrow$ *Expcont* $\rightarrow$ *Cmdcont*
$\mathcal{P}_{FLK}$: Primop $\rightarrow$ *Value** $\rightarrow$ *Expressible* ; *Defined as usual*

$\mathcal{P}_{FLK!}[\![\texttt{cell-ref}]\!]$
$= \lambda[v]ks\,.\ ((\textit{test-location}\ (\lambda ls'\,.\ ((\textit{ensure-assigned}\ (\textit{fetch}\ l\ s')\ k)\ s')))$
$v\ s)$

$\mathcal{P}_{FLK!}[\![\texttt{cell-set!}]\!]$
$= \lambda[v_1, v_2]ks\,.\ ((\textit{test-location}\ (\lambda ls'\,.\ (k\ (\textit{Unit} \mapsto \textit{Value}\ \textit{unit})\ (\textit{assign}\ l\ v_2\ s'))))$
$v_1\ s)$

$\mathcal{P}_{FLK!}[\![O_{FLK}]\!] = \lambda v^* k\,.\ (\textit{ensure-value}\ (\mathcal{P}_{FLK}[\![O_{FLK}]\!]\ v^*)\ k)$
where $O_{FLK} \in$Primop $- \{\texttt{cell-ref}, \texttt{cell-set!}\}$

Figure 9.14: Valuation clauses for standard semantics of FLK!, Part II.

the area of control is the big payoff for our investment in denotational semantics; many control constructs that have succinct denotational descriptions are difficult to describe in an operational framework.

The standard semantics for FLK! differs from the direct semantics for FLK! in the following ways:

- The *Expressible* domain has been replaced by the *Answer* domain. In a standard semantics, the *Answer* domain is used to represent the "final" value of a program. Not all standard semantics actually return a value for an expression. For example, the initial continuation might be an interpreter's read-eval-print loop, which never returns. In this case, the behavior of the program could be modeled as a mapping from a sequence of input s-expressions to a sequence of output s-expressions. Nevertheless, in the particular case of FLK!, *Answer* is the same as *Expressible*.

- The standard semantics introduces two continuation domains, *Expcont* and *Cmdcont*:

  $k \in \textit{Expcont} = \textit{Expressible} \rightarrow \textit{Cmdcont}$
  $\gamma \in \textit{Cmdcont} = \textit{Store} \rightarrow \textit{Answer}$

  Expcont is the domain of expression continuations; *Cmdcont* is the domain of command continuations. These types of continuations reflect a common distinction in programming languages between **commands** and **expressions**. An expression yields a value in addition to any modifications it

might make to the store. The example languages in this book are *expression languages* because all program constructs are expressions that return a value. Many languages, e.g., PASCAL, have syntactically distinct expressions and commands as well as contexts that require one or the other.[3]

A command, on the other hand, is executed for its effect(s) and does not produce a meaningful value. Program output is the classic example of a command: `writeln` in PASCAL, for example, prints its arguments as a line of output on the standard output device. Variable assignment is also naturally thought of as a command. In FLAVAR!, `set!` expressions return the uninteresting value `#u` simply because they are required to return something, but the reason to execute an assignment is to modify the store. Sequencing using `begin` is a natural command context: it exists to enforce an order of state transformations.

Since expressions return a value and modify the store, the continuation for an expression expects both the value and store produced by that expression. A command continuation expects only a store. Note that because $Cmdcont = Store \rightarrow Answer$, we can also view *Expcont* as:

$k \in Expcont = Expressible \rightarrow Store \rightarrow Answer$

That is, we can think of an expression continuation as taking an expressible value and returning a command continuation; or we may think of it as taking an expressible value and a store and returning an answer. Which perspective is more fruitful depends on the situation.

- The signature of $\mathcal{E}$ has been modified:

  $\mathcal{E} : \text{Exp} \rightarrow Environment \rightarrow Expcont \rightarrow Cmdcont$

  Recall that since $Cmdcont = Store \rightarrow Answer$ we can also view $\mathcal{E}$ as:

  $\mathcal{E} : \text{Exp} \rightarrow Environment \rightarrow Expcont \rightarrow Store \rightarrow Answer$

  That is, $\mathcal{E}$ takes a syntactic expression and representations of the naming (*Environment*), control (*Expcont*), and state (*Store*) contexts, and finds the meaning of the expression (an answer) with respect to these contexts.

[3] It is possible to coerce an expression to a command by ignoring its return value. This is what FL does with all but the final subexpression in a `begin`.

An expression of the form

$$(\mathcal{E}[\![E]\!]\ e\ (\lambda v\ .\ \Box))$$

can be read as "Find the value of E in e and name the result v in $\Box$."

Since evaluating an expression requires a store in FLK!, why doesn't a store appear in the above expression? The reason is that the order of arguments to $\mathcal{E}$ has been chosen to take the store as its final argument, rather than the continuation. This argument order is one of the conventions of a standard semantics; it is used because it hides the store when it is threaded through an expression untouched. In essence, *Cmdcont* fulfills the role that the *Computation* domain did when we introduced state into FL. To specify that an expression takes in one store, say s_0, and returns another, s_1, we write:

$$(\mathcal{E}[\![E]\!]\ e\ (\lambda v s_1\ .\ \Box)\ s_0)$$

(Observe the explicit store parameters in the denotations for constructs involving cells.)

- The definition of the *Procedure* domain is changed to take an expression continuation:

  $p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Expcont} \to \mathit{Cmdcont}$

  Again, we can also view this definition as:

  $p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Expcont} \to \mathit{Store} \to \mathit{Answer}$

  The *Procedure* domain in the standard semantics differs from the *Procedure* domain in the direct semantics in that procedures take an additional argument from the *Expcont* domain. Intuitively, this argument is the "return address" that a procedure returns to when it returns a value.

- The new *test-xxx* auxiliary functions are used to convert continuations expecting arguments of type *xxx* into expression continuations. Like the *with-xxx* functions from previous semantics, the *test-xxx* functions hide details of error generation. However, unlike the *with-xxx* functions, the *test-xxx* functions do not propagate errors.

Even though FL! does not have any advanced control features (we'll add quite a few in the remainder of this chapter), the standard semantics still has an advantage over a direct semantics: the modelling of errors. A valuation clause in a standard semantics generates an error by ignoring the current continuation and directly returning an error. See, for example, the valuation clause for `error`. This captures the intuition that an error immediately aborts the computation.

The standard semantics valuation clauses in Figures 9.13 and 9.14 do not employ the computation abstraction that we have been using in our denotational definitions. We presented them in a concrete manner to help build intuitions about continuations. However, it is not difficult to recast standard semantics into the computation framework. Figures 9.15 and 9.16 show an implementation of the computation abstraction that defines *Computation* as *Expcont* $\rightarrow$ *Store* $\rightarrow$ *Expressible*. (We assume that *Answer* is *Expressible*, but we could readily redefine *Answer* to be another domain.) With this implementation of the computation abstraction, the FLK! valuation clauses from the previous chapter still hold, except for a minor tweak in the handling of CBV `rec`:

$\mathcal{E}[\![$`(rec` I E`)`$]\!]$
$= \lambda e\,.\ \mathbf{fix}_{Computation}\ (\lambda c\,.\ \lambda ks_0\,.\ (\mathcal{E}[\![E]\!]\ [I :: (\textit{extract-value}\ c\ s_0)]e\ k\ s_0))$

$\textit{extract-value} : \textit{Computation} \rightarrow \textit{Store} \rightarrow \textit{Binding}$
$= \lambda cs_0\,.\ \mathbf{matching}\,(c\ (\lambda vs\,.\,(\textit{Value} \mapsto \textit{Expressible}\ v))\ s_0)$
$\quad \triangleright\,(\textit{Value} \mapsto \textit{Expressible}\ v)\ [\!]\ (\textit{Denotable} \mapsto \textit{Binding}\ v)$
$\quad \triangleright\,(\textit{Error} \mapsto \textit{Expressible}\ y)\ [\!]\ \bot_{Binding}$
$\quad$ **endmatching**

There are two important changes in the valuation clause for `rec`:

- *extract-value* must account for the fact that the *Answer* domain is *Expressible* rather than *Expressible* $\times$ *Store*.
- The new $\mathcal{E}$ function requires a continuation argument, which we use to hijack the value used in the binding for I. Notice that this continuation is rather like the top level continuation in Figure 9.12.

Figure 9.17 introduces two continuation-specific auxiliary functions along with their associated reasoning laws. Since no FLK! construct does anything interesting with a continuation, these auxiliaries would not appear in valuation clauses for FLK!. However, they will be useful when we extend FLK! with advanced control features.

$\triangleright$ **Exercise 9.2** Imperative Languages Inc. was impressed with the simplicity and power of FL!. Noticing that it lacks a looping construct and not willing to support a product not in consonance with the company's programming language philosophy, the company calls Ben Bitdiddle to extend the language. Instead of a myriad of different constructs (e.g. `for`, `while`, `do-while`, etc.) Ben designs a single loop expression that embodies all forms of looping. The syntax of FLK! was extended as follows:

$E ::= \ldots$	[As before]
| `(loop` E`)`	[Evaluate E repeatedly]
| `(break` E`)`	[End lexically enclosing loop with value of E]
| `(continue)`	[Restart evaluation of enclosing loop expression]

$c \in \mathit{Computation} = \mathit{Expcont} \to \mathit{Store} \to \mathit{Expressible}$
$k \in \mathit{Expcont} = \mathit{Value} \to \mathit{Store} \to \mathit{Expressible}$
$x \in \mathit{Expressible} = (\mathit{Value} + \mathit{Error})_\perp$; *As before*
$v \in \mathit{Value} =$ *language dependent*
$y \in \mathit{Error} = \mathit{Sym}$; *Modified*

expr-to-comp : $\mathit{Expressible} \to \mathit{Computation}$
$= \lambda x\,.$ **matching** x
$\quad \triangleright\, (\mathit{Value} \mapsto \mathit{Expressible}\ v) \,[\!]\, (\textit{val-to-comp}\ v)$
$\quad \triangleright\, (\mathit{Error} \mapsto \mathit{Expressible}\ y) \,[\!]\, (\textit{err-to-comp}\ y)$
$\quad$ **endmatching**

val-to-comp : $\mathit{Value} \to \mathit{Computation}\ = \lambda v\,.\ \lambda k\,.\ (k\ v)$

err-to-comp : $\mathit{Error} \to \mathit{Computation}\ = \lambda I\,.\ \lambda ks\,.\ (\mathit{Error} \mapsto \mathit{Expressible}\ I)$

with-value : $\mathit{Computation} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda cf\ .\ \lambda k\,.\ (c\ (\lambda v\,.\,(f\ \ v\ k)))$
with-values, *with-boolean*, *with-procedure*, etc. can be written in terms of *with-value*.

check-location : $\mathit{Value} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda vf\ .$ **matching** v
$\quad \triangleright\, (\mathit{Location} \mapsto \mathit{Value}\ l) \,[\!]\, (f\ \ l)$
$\quad \triangleright$ **else** (*err-to-comp* `non-location`)
$\quad$ **endmatching**
check-boolean, *check-procedure*, etc. are similar.

Figure 9.15: Continuation-based computation abstraction, Part I.

State-based auxiliaries:

allocating : *Storable* → (*Location* → *Computation*) → *Computation*
$= \lambda\sigma f\ .\ \lambda ks\ .\ (f\ \ (\textit{fresh-loc}\ s)\ \ k\ \ (\textit{assign}\ \ (\textit{fresh-loc}\ s)\ \ \sigma\ \ s))$

fetching : *Location* → (*Storable* → *Computation*) → *Computation*
$= \lambda lf\ .\ \lambda ks\ .$ **matching** (*fetch* $l\ s$)
▷ (*Storable* ↦ *Assignment* σ) ‖ $(f\ \ \sigma\ \ k\ \ s)$
▷ **else** ((*err-to-comp* `unassigned-location`) $k\ s$)
endmatching

update : *Location* → *Storable* → *Computation*
$= \lambda l\sigma\ .\ \lambda ks\ .\ (k\ \ (\textit{Value} \mapsto \textit{Expressible}\ \ (\textit{Unit} \mapsto \textit{Value}\ \ \textit{unit}))\ \ (\textit{assign}\ \ l\ \ \sigma\ \ s))$

sequence : *Computation* → *Computation* → *Computation*
$= \lambda c_1\, c_2\ .\ (\textit{with-value}\ \ c_1\ \ (\lambda v\ .\ c_2))$; *Unchanged from before*

Figure 9.16: Continuation-based computation abstraction, Part II.

New auxiliary functions for control:

capturing-cont : (*Expcont* → *Computation*) → *Computation*
$= \lambda f\ .\ \lambda k\ .\ ((f\ \ k)\ \ k)$

install-cont : *Expcont* → *Computation* → *Computation*
$= \lambda k_{new}\, c\ .\ \lambda k_{old}\ .\ (c\ \ k_{new})$

New Reasoning Laws:

$(\textit{with-value}\ \ (\textit{install-cont}\ \ k\ \ c)\ \ f) = (\textit{install-cont}\ \ k\ \ c)$

$(\textit{with-value}\ \ (\textit{capturing-cont}\ \ f)\ \ g) = (\textit{capturing-cont}\ \ (\lambda k\ .\ (\textit{with-value}\ \ (f\ \ k)\ \ g)))$
where k is not free in f or g.

$(\textit{capturing-cont}\ \ (\lambda k\ .\ (\textit{install-cont}\ \ k\ \ c))) = c$
where k is not free in c.

Figure 9.17: Continuation-specific auxiliary functions on computations.

Here's the informal semantics of the new constructs:

a. `(loop` *E*`)`: Evaluates *E* (the "looping expression") repeatedly forever.

b. `(break` *E*`)`: Ends the nearest lexically enclosing `loop`, which then returns the value of *E*.

c. `(continue)`: Restarts the evaluation of the looping expression for the nearest lexically enclosing `loop`.

It is an error to evaluate either a `break` or a `continue` expression outside a lexically enclosing `loop` expression.

Here's an example of Ben's looping constructs in action:

```
; Ben's iterative factorial procedure
(lambda (n)
  (let ((fact 1))
    (loop
      (if (= n 0)
        (break fact)
        (begin
          (set! fact (* fact n))
          (set! n (- n 1)))))))
```

In addition to extending the language, Ben has been asked to extend its standard denotational semantics. It is here that Ben has subcontracted you.

a. Give the new signature of the meaning function $\mathcal{E}$.

b. Give the meaning function clauses for `(loop` *E*`)`, `(break` *E*`)`, and `(continue)`.

◁

▷ **Exercise 9.3** Ben Bitdiddle is very excited about the power of the standard semantics to describe complicated flows of control. Wanting to practice more with this wonderful tool, he started churning out a lot of FL! extensions (not all of them useful). Most recently, Ben added a construct `(self` *E*`)` in order to allow a procedure to call itself, without using `rec` or `letrec`. Ben modified the FLK! grammar as follows:

E ::= ... `existing` FLK! `constructs` ...
| `(self` *E*`)`

Informally, `(self` *E*`)` recursively calls the current procedure with an actual argument that is the result of evaluating *E*. Here is a small example:

```
(let ((fact (lambda (n) (if (= n 0) 1 (* n (self (- n 1)))))))
     (fact 4)) ⟶ 24
                eval
```

When `(self` *E*`)` is used outside a procedure, it causes the program to terminate immediately with a value that is the result of evaluating *E*.

Ben started describing the formal semantics of `(self` *E*`)` by modifying the signature of the meaning function $\mathcal{E}$ as follows:

$\mathcal{E}$: Exp → *Environment* → *SelfProc* → *Expcont* → *Cmdcont*
where *SelfProc* = *Procedure*

In spite of his enthusiasm, Ben is still inexperienced with standard semantics. It is your task to help him specify the formal semantics of the `self` construct.

a. Give the revised definition of the top level meaning function $\mathcal{TL}[\![E]\!]$.

b. Give the meaning function clause for `(call` E_1 E_2`)`, `(self` E`)` and `(proc` I E`)`.

c. Prove that `(self (self 1))` evaluates to (*Value* ↦ *Expressible* (*Int* ↦ *Value* 1)).

d. Prove that `(proc x (self 1))` evaluates to a procedure that, no matter what input it is called with, loops forever. ◁

▷ **Exercise 9.4** Ben Bitdiddle is now working in a major research university where he's investigating a new approach to programming based on coroutines. Of course, he bases his research on FLK! and its standard semantics. He adds the following expressions to the FLK! grammar:

E ::= `...`| `(coroutine (`I E_1`)` E_2`)`| `(yield` E`)`

The meaning of the expression `(coroutine (`I E_1`)` E_2`)` is simply E_2, unless E_2 performs a `(yield` E_3`)`. If E_2 performs a `(yield` E_3`)`, then the value of the coroutine expression is simply E_1, except that I is bound to E_3. However, if E_1 performs a `(yield` E_4`)`, then control transfers back to E_2, with the value of the original `(yield` E_3`)` — the point where control was originally transferred to E_1 — being replaced by E_4. Thus, `yield` transfers control back and forth between the two expressions, passing a value between them. A `(yield` E`)` in one expression transfers control to the other expression; that expression resumes at the point of its last `yield`, whose value is set to E.

The following example coroutine expressions may help to make things clear. The underline mark shows the active expression; the series of dots ····· shows a `yield` expression that has already yielded control to the other expression.

```
(coroutine (x 1) 2)
⇒ 2

(coroutine (x 1) (yield 2))
⇒ (coroutine (x 1) ·····)
⇒ 1

(coroutine (x x) (yield 2))
⇒ (coroutine (x x) ·····)
⇒ (coroutine (x 2) ·····)
⇒ 2
```

```
(coroutine (x (yield (+ 1 x))) (yield 2))
⇒ (coroutine (x (yield (+ 1 x))) ·····)
⇒ (coroutine (x (yield (+ 1 2))) ·····)
⇒ (coroutine (x (yield 3)) ·····)
⇒ (coroutine (x ·····) 3)
⇒ 3

(coroutine (x (yield (+ 1 x))) (+ 2 (yield 3)))
⇒ (coroutine (x (yield (+ 1 x))) (+ 2 ·····))
⇒ (coroutine (x (yield (+ 1 3))) (+ 2 ·····))
⇒ (coroutine (x (yield 4)) (+ 2 ·····))
⇒ (coroutine (x ·····) (+ 2 4))
⇒ 6
```

As one of Ben's students, your job is to write the denotational semantics for FLK! + coroutines. Ben has already revised the domain equations to include both normal and yield continuations, as follows:

$k \in \mathit{Normal\text{-}Cont} = \mathit{Expcont}$
$y \in \mathit{Yield\text{-}Cont} = \mathit{Expcont}$
$\mathit{Expcont} = \mathit{Value} \to \mathit{Cmdcont}$
$\mathit{Cmdcont} = \mathit{Yield\text{-}Cont} \to \mathit{Store} \to \mathit{Expressible}$
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Normal\text{-}Cont} \to \mathit{Cmdcont}$

He's also changed the signature of the $\mathcal{E}$ meaning function so that every expression is evaluated with both a normal and a yield continuation:

$$\begin{aligned}\mathcal{E} \;&:\; \mathrm{Exp} \to \mathit{Environment} \to \mathit{Normal\text{-}Cont} \to \mathit{Cmdcont}\\ &= \mathrm{Exp} \to \mathit{Environment} \to \mathit{Normal\text{-}Cont} \to \mathit{Yield\text{-}Cont} \to \mathit{Store} \to \mathit{Expressible}\end{aligned}$$

He didn't get that far when defining $\mathcal{E}$, but he did give you the meaning function clause for `(if` E_1 E_2 E_3`)` for reference.

$$\begin{aligned}&\mathcal{E}[\![\texttt{(if } E_1\ E_2\ E_3\texttt{)}]\!] =\\ &\quad \lambda e k y\,.\ (\mathcal{E}[\![E_1]\!]\ e\\ &\qquad\qquad (\mathit{test\text{-}boolean}\ \ (\lambda b\,.\,\mathbf{if}\ b\ \ \mathbf{then}\ (\mathcal{E}[\![E_2]\!]\ e\ k)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ \ (\mathcal{E}[\![E_3]\!]\ e\ k)))\\ &\qquad\qquad y)\end{aligned}$$

a. Give the meaning function clause for L, given the new domains.

b. Give the meaning function clause for `(yield` E`)`.

c. Give the meaning function clause for `(coroutine (`I E_1`)` E_2`)`.

d. Compute the meaning of `(yield (yield 3))` according to your semantics. ◁

▷ **Exercise 9.5** Alyssa P. Hacker thinks coroutines (see Section 9.2.3) make programs too hard to reason about. She suggests a simplified version of coroutines: the producer/consumer paradigm. Informally, a producer generates values one at a time, and

the values are used by a consumer in the order they are produced. Alyssa modifies the FLK! grammar as follows:

E ::= ... *normal* FLK! *constructs* ...
| `(producer` I_{yield} E_{body}`)`
| `(consume` $E_{producer}$ $I_{current}$ E_{body}`)`

The informal semantics of these two newly added constructs are:

- `(producer` I_{yield} E_{body}`)` creates a new kind of first-class object called a *producer*. When a producer is invoked (by `consume`) the identifier I_{yield} is bound to a yielding procedure. Calling I_{yield} in E_{body} with a value passes control and the yielded value to the consumer. When the consumer is done processing the value, the I_{yield} procedure returns `#u` to the producer. The value of E_{body} is the value returned by the producer when there are no more values to yield.
- `(consume` $E_{producer}$ $I_{current}$ E_{body}`)` invokes the producer that $E_{producer}$ evaluates to (it is an error if $E_{producer}$ doesn't evaluate to a producer). Whenever the producer yields a value, $I_{current}$ is bound to that value and E_{body} is evaluated. The result of evaluating E_{body} is then discarded, and control returns to the producer. The result returned by `consume` is the result returned by the producer.

For example, `up-to` is a procedure that takes an integer `n` as an argument, and returns a producer that yields the integers from `1` up to and including `n`.

```
(define up-to
  (lambda (n)
    (producer emit
      (letrec ((loop (lambda(i)
                       (if (> i n)
                           #f
                           (begin (emit i)
                                  (loop (+ i 1)))))))
        (loop 1)))))
```

The `sum` procedure adds all the numbers yielded by a given producer:

```
(define sum
  (lambda (prod)
    (let ((ans (cell 0)))
      (begin
        (consume prod n (cell-set! ans (+ n (cell-ref ans))))
        (cell-ref ans)))))
```

For example, `sum` can be used to add up the values produced by the producer `(up-to 5)`:

`(sum (up-to 5))` $\xrightarrow[eval]{}$ *33*

Note that when a producer does not yield additional values and returns a normal value v, execution of the invoking `consume` form is terminated and v is returned.

`(consume (up-to 5) i 7)` $\xrightarrow[eval]{}$ *false*

Assume in the following questions that FL! is desugared in the usual way to FLK!.

a. Alyssa wants to update the standard semantics of FLK! in order to specify the formal semantics of the newly introduced constructs. Alyssa starts by creating a new domain for producers, and adds it to the value domain:

$$v \in \mathit{Value} = \mathit{Unit} + \mathit{Bool} + \ldots + \mathit{Procedure} + \mathit{Producer}$$
$$q \in \mathit{Producer} = \mathit{Procedure} \rightarrow \mathit{Expcont} \rightarrow \mathit{Cmdcont}$$

Alyssa's valuation clause for the `consume` construct is as follows:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(consume } E_{producer}\ I_{current}\ E_{body}\texttt{)}]\!] = \\ \quad \lambda e k_{normal} \,.\, (\mathcal{E}[\![E_{producer}]\!]\ \ e \\ \qquad\qquad (\mathit{test\text{-}producer} \\ \qquad\qquad (\lambda q \,.\, (q\ \ (\lambda v_{yielded} k_{after-yield} \,. \\ \qquad\qquad\qquad\qquad (\mathcal{E}[\![E_{body}]\!]\ \ [I_{current} : v_{yielded}]e \\ \qquad\qquad\qquad\qquad\quad (\lambda v \,.\ (k_{after-yield}\ \ (\mathit{Unit} \mapsto \mathit{Value}\ \ \mathit{unit}))))) \\ \qquad\qquad\qquad\qquad k_{normal}))))\end{array}$$

$$\begin{array}{l}\mathit{test\text{-}producer} : (\mathit{Producer} \rightarrow \mathit{Cmdcont}) \rightarrow \mathit{Expcont} = \\ \quad \lambda f \,.\, (\lambda v \,.\, \textbf{matching}\ v \\ \qquad\qquad \triangleright\, (\mathit{Producer} \mapsto \mathit{Value}\ \ q)\ [\![\ \ (f\ \ q) \\ \qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}cont} \\ \qquad\qquad \textbf{endmatching}\ )\end{array}$$

Write the evaluation clause for the `producer` construct.

b. Alyssa also decides to specify the behavior of `producer` and `consume` in terms of their operational semantics. She starts with the SOS semantics for FLK! from Section 8.2.3.

State the modifications to the SOS semantics of FLK! that are necessary to handle `producer` and `consumer`, including any relevant rules.

c. Ben Bitdiddle discovers how to desugar Alyssa's constructs into normal FL! constructs. Ben's desugaring for `producer` is

$$\mathcal{D}\texttt{(producer } I_{yield}\ \ E_{body}\texttt{)} = \texttt{(lambda (}I_{yield}\texttt{)}\ \ E_{body}\texttt{)}$$

Write a corresponding desugaring for `consume`. ◁

▷ **Exercise 9.6** Sam Antics is aggressively using the standard semantics to define the meaning of some really non-standard FL! constructs. Most recently, he extended FL! with some special constructs for POP (i.e., "Politically Oriented Programming"). He extended the FLK! grammar as follows:

$$\begin{array}{rl}E ::= & \ldots \texttt{ existing FLK! constructs } \ldots \\ & \mid \texttt{(elect } E_{pres}\ \ E_{vp}\texttt{)} \\ & \mid \texttt{(reelect)} \\ & \mid \texttt{(impeach)}\end{array}$$

Here's the informal semantics of the newly introduced constructs:

- `(elect` E_{pres} E_{vp}`)` evaluates to the value of E_{pres} unless `(impeach)` is evaluated in E_{pres}, in which case it evaluates to the value of E_{vp}. It is possible to have nested `elect` constructs.
- If `(reelect)` is evaluated inside the E_{pres} part of a `(elect` E_{pres} E_{vp}`)` construct, it goes back to the beginning of the `elect` construct. Otherwise, it signals an error.
- If `(impeach)` is evaluated within the E_{pres} part of a `(elect` E_{pres} E_{vp}`)` construct, it causes the `elect` expression to evaluate to E_{vp}. Otherwise, it signals an error.

Here's a small example that Sam plans to use in his advertising campaign for the FL! *2000 Presidential Edition (TM)*:

```
(let ((scandals (cell 0)))
   (elect (if (< (cell-ref scandals) 5)
              (begin (cell-set! scandals
                                (+ (cell-ref scandals) 1))
                     (reelect))
              (impeach))
          (* (cell-ref scandals) 2)))
```

$\xrightarrow[eval]{}$ *10*

You are hired by Sam Antix to modify the standard denotational semantics of FLK! in order to define the formal semantics of the newly introduced constructs. Sam has already added the following semantic domains:

$r \in \mathit{Prescont} = \mathit{Cmdcont}$
$i \in \mathit{Vpcont} = \mathit{Cmdcont}$

He also changed the signature of the meaning function:

$\mathcal{E} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Prescont} \to \mathit{Vpcont} \to \mathit{Expcont} \to \mathit{Cmdcont}$

a. Give the definition of the top level meaning function $\mathcal{TL}[\![E]\!]$.

b. Give the meaning function clauses for $\mathcal{E}[\![$`(elect` E_{pres} E_{vp}`)`$]\!]$, `(reelect)`, and `(impeach)`.

c. Use the meaning functions you defined to compute $\mathcal{TL}[\![$`(elect (reelect 1))`$]\!]$.

◁

▷ **Exercise 9.7** This problem requires you to modify the standard denotational semantics for FLK!.

Sam Antics is working on a new language with hot new features that will appeal to government customers. He was going to base his language on Caffeine from Moon

Microsystems, but negotiations broke down. He has therefore decided to extend FLK! and has hired you, a top FLK! consultant, to assist with modifying the language to support these new features. The new language is called FLK#, part of Sam Antics' new .GOV platform. The big feature of FLK# is user tracking and quotas in the store. An important customer observed that government users tended to use the store carelessly, resulting in expensive memory upgrades. To improve the situation, the FLK# store will maintain a per-user quota. The Standard Semantics of FLK! are changed as follows:

$$\begin{aligned} w \in \mathit{UserID} &= \mathit{Int} \\ q \in \mathit{Quota} &= \mathit{UserID} \to \mathit{Int} \\ \mathit{gamma} \in \mathit{Cmdcont} &= \mathit{UserID} \to \mathit{Quota} \to \mathit{Store} \to \mathit{Answer} \end{aligned}$$

UserID is just an integer. 0 is reserved for the case when no one is logged in. *Quota* is a function that when given a *UserID* returns the number of cells remaining in the user's quota. The quota starts at 100 cells. *Cmdcont*, the command continuation, takes the currently logged in user ID, the current quota, and the current store to yield an answer. Plus, FLK# adds the following commands:

E ::= `...` [Classic FLK! expressions]
| `(login!` w`)` [Log in user w]
| `(logout!)` [Log out current user]

`(login!`w`)` — logs in the user associated with the identifier; returns the identifier (returns an error if a user is already logged in or if the UserID is 0)
`(logout!)` — logs the current user out; returns the last user's identifier (returns an error if there is no user logged in)
`(check-quota)` — returns the amount of quota remaining

The definition of $\mathcal{E}[\![$`(check-quota)`$]\!]$ is:

$$\begin{aligned} &\mathcal{E}[\![\texttt{(check-quota)}]\!] = \\ &\quad \lambda ekwq\,.\ \textbf{if}\ w = 0 \\ &\qquad\quad \textbf{then}\ (\mathit{error\text{-}cont}\ \texttt{no-user-logged-in}\ \ w\ \ q) \\ &\qquad\quad \textbf{else}\ (\mathrm{k}\ (\mathit{Int} \mapsto \mathit{Value}\ (q\ \ w))\ w\ \ q)\ \ \textbf{fi} \end{aligned}$$

a. Write the meaning function clause for $\mathcal{E}[\![$`(login!` E`)`$]\!]$.

b. Write the meaning function clause for $\mathcal{E}[\![$`(logout!)`$]\!]$.

c. Give the definition of $\mathcal{E}[\![$`(cell` E`)`$]\!]$. Remember you cannot create a cell unless you are logged in.

d. Naturally, Sam Antics wants to embed some "trap doors" into the .GOV platform to enable him to "learn more about his customers." One of these trap doors is the undocumented `(raise-quota! n)` command, which adds `n` cells to the quota of the current user and returns `0`. Give the definition of $\mathcal{E}[\![$`(raise-quota!` E`)`$]\!]$. ◁

9.4 Non-local Exits

A denotational semantics equipped with continuations is especially useful for modelling advanced control features of programming languages. One such feature is a **non-local exit**, a mechanism that aborts a pending computation by forcing control to jump to a specified place in the program.

To study non-local exits, we extend FLK! with two new constructs:

E ::= ... | `(label` I_{ctrl_pt} E_{body}`)` | `(jump` E_{ctrl_pt} E_{body}`)`

The informal semantics of these constructs is as follows:

- `(label` I_{ctrl_pt} E_{body}`)` evaluates E_{body} in an environment where the name I_{ctrl_pt} is bound to the **control point** that receives the value of the `label` expression.
- `(jump` E_{ctrl_pt} E_{body}`)` returns the value of E_{body} to the control point that is the value of E_{ctrl_pt}. If E_{ctrl_pt} does not evaluate to a control point, `jump` generates an error.

```
(+ 1 (label exit (* 2 (- 3 (/ 4 1))))) ⟶FL!⟶ -1

(+ 1 (label exit (* 2 (- 3 (/ 4 (jump exit 5)))))) ⟶FL!⟶ 6

(+ 1 (label exit
       (* 2 (- 3 (/ 4 (jump exit (+ 5 (jump exit 6)))))))) ⟶FL!⟶ 7

(+ 1 (label exit1
       (* 2 (label exit2 (- 3 (/ 4 (+ (jump exit2 5)
                                      (jump exit1 6)))))))) ⟶FL!⟶ 11
```

Figure 9.18: Some examples using `label` and `jump`.

Figure 9.18 shows some simple examples using `label` and `jump`. The first example illustrates that the value of `(label` I E`)` is the value of E if E performs no `jump`s. In the second example, `(jump exit 5)` aborts the pending `(* 2 (- 3 (/ 4 □)))` computation and returns `5` as the value of the `label` expression. The third example demonstrates that a pending `jump` can itself be aborted by a `jump` within one of its subexpressions. In the final example, the left-to-right evaluation of `call` subexpressions causes `5` to be returned as the value of `(label exit2 ...)`. If `call` subexpressions were evaluated in right-to-left order, the result of the final example would be `7`.

In practice, non-local exits are a convenient means of communicating information between two points of a program separated by pending operations without performing any of the pending operations. For instance, here is yet another version of a recursive procedure for computing the product of a list of numbers (see Section 9.2.2):

```
(define (prod-list a-list)
  (label return
    (letrec ((prod (lambda (lst)
                     (cond ((null? lst) 1)
                           ((= 0 (car lst)) (jump return 0))
                           (else (* (car lst)
                                    (prod (cdr lst))))))))
      (prod a-list))))
```

Upon encountering a `0` in the list, the internal `prod` procedure uses `jump` to immediately return `0` as the result of a call to `prod-list`. Any pending multiplications generated by recursive calls to `prod` are flushed.

The semantics for `label` and `jump` appear in Figure 9.19. Control points are modelled as expression continuations that are treated as first-class values. The valuation clauses for `label` and `jump` are presented in two styles: the traditional style of standard semantics, and a style based on the computation abstraction. `label` redefines its continuation k as a control point value and evaluates E_{body} in the environment e extended with a binding between I_{ctrl_pt} and the control point value. `jump` ignores its default continuation and instead evaluates E_{body} with the continuation determined by E_{ctrl_pt}.

Note that `label` refers to its continuation twice: it both names it and uses it as the continuation of E_{body}. (This is easier to see in the standard style than in the computation style.) This means that a value can be returned from a `label` expression in two ways: (1) by normal evaluation of E_{body} (without any `jump`s) and (2) by using `jump` with a control point that is extracted from the environment. In contrast, `jump` does not refer to its continuation at all. This means that a `jump` expression can never return! So it is meaningless to ask what the value of a `jump` expression is. Similarly, expressions containing `jump` expressions may also have no value. This is the first time we have seen expressions without values in a dialect of FL.

Like all other values in FL!, control point values are first-class: they can be named, passed as arguments, returned as results, and stored in data structures (pairs, cells). An interesting consequence of this fact is that it is possible to return to the same control point more than once. Consider the following FL! expression:

Abstract Syntax:

$$
\begin{array}{lll}
E ::= \ldots & & \text{[As before]} \\
\quad | \ \texttt{(label}\ I_{name}\ \ E_{body}\texttt{)} & & \text{[Label]} \\
\quad | \ \texttt{(jump}\ E_{control-point}\ \ E_{val}\texttt{)} & & \text{[Jump]}
\end{array}
$$

Semantic Domains:

$$
\begin{array}{l}
\quad \mathit{ControlPoint} = \mathit{Expcont} \\
v \in \mathit{Value} = \ldots + \mathit{ControlPoint}
\end{array}
$$

$$\mathit{test\text{-}control\text{-}point} : (\mathit{ControlPoint} \to \mathit{Cmdcont}) \to \mathit{Expcont}$$

Valuation functions (standard version):

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(label}\ I_{ctrl\_pt}\ \ E_{body}\texttt{)}]\!] \\
\quad = \lambda ek\ .\ (\mathcal{E}[\![E_{body}]\!]\ [I_{ctrl\_pt} : (\mathit{ControlPoint} \mapsto \mathit{Value}\ \ k)]e\ \ k)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(jump}\ E_{ctrl\_pt}\ \ E_{val}\texttt{)}]\!] \\
\quad = \lambda ek_{ignore}\ .\ (\mathcal{E}[\![E_{ctrl\_pt}]\!]\ \ e\ \ (\mathit{test\text{-}control\text{-}point}\ (\lambda k_{ctrl\_pt}\ .\ (\mathcal{E}[\![E_{val}]\!]\ \ e\ \ k_{ctrl\_pt}))))
\end{array}
$$

Valuation functions (computation version):

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(label}\ I_{ctrl\_pt}\ \ E_{body}\texttt{)}]\!] \\
\quad = \lambda e\ .\ (\mathit{capturing\text{-}cont}\ (\lambda k\ .\ (\mathcal{E}[\![E_{body}]\!]\ [I_{ctrl\_pt} : (\mathit{ControlPoint} \mapsto \mathit{Value}\ \ k)]e)))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(jump}\ E_{ctrl\_pt}\ \ E_{val}\texttt{)}]\!] \\
\quad = \lambda e\ .\ (\mathit{with\text{-}control\text{-}point} \\
\qquad\quad (\mathcal{E}[\![E_{ctrl\_pt}]\!]\ \ e)\ (\lambda k_{ctrl\_pt}\ .\ (\mathit{install\text{-}cont}\ \ k_{ctrl\_pt}\ \ (\mathcal{E}[\![E_{val}]\!]\ \ e))))
\end{array}
$$

Figure 9.19: The semantics of `label` and `jump` in FLK!.

```
(let ((c (cell 'later)))
  (let ((n (label bind-n
             (begin (cell-set! c bind-n)
                    1))))
    (if (> n 17)
        n
        (jump (cell-ref c) (* 2 n))))) ⟶_{FL!} 32
```

Here, `bind-n` names the control point that: (1) accepts a value, (2) binds the value to `n`, and (3) evaluates the `if` expression. This control point is stashed away in the cell `c` for later use, and then a 1 is returned to the normal flow of control. Since this value for `n` is less than 17, the `jump` is performed, which returns the value of 2 to the same `bind-n` control point. This causes `n` to be rebound to 2 and the `if` expression to be evaluated a second time. Continuing in this manner, the expression behaves like a loop that successively binds `n` to the values 1, 2, 4, 8, 16, and 32. The final result is 32 because that is the first power of two that is greater than 17.

A similar trick can be used to phrase an imperative version of an iterative factorial procedure in terms of `label` and `jump`:

```
(define factorial
  (lambda (n)
    (let ((loop (cell 'later))
          (num (cell n))
          (ans (cell 1)))
      (begin
        (label top (cell-set! loop (lambda ()
                                     (jump top 'ignore))))
        (if (= (cell-ref num) 0)
            (cell-ref ans)
            (begin
              (cell-set! ans (* (cell-ref num) (cell-ref ans)))
              (cell-set! num (- (cell-ref num) 1))
              ((cell-ref loop)))))))))
```

It turns out that mutation is not necessary for exhibiting this sort of looping behavior via `label` and `jump` (see Exercise 9.9).

The above examples of first-class continuations (i.e., control points) are rather contrived. However, in languages that support them (such as SCHEME and some dialects of ML), first-class continuations provide a powerful mechanism by which programmers can implement advanced control features. For instance, coroutines, backtracking, and multi-threading can all be implemented in terms of first-class continuations. But any control abstraction mechanism

this powerful can easily lead to programs that are virtually impossible to understand. After all, it turns the notion of `goto`-less programming on its head by making `goto` labels first-class values! Thus, great restraint should be exercised when using first-class continuations.

In SCHEME, first-class continuations are made accessible by the standard procedure `call-with-current-continuation`, which we will abbreviate as `cwcc` (another common abbreviation is `call/cc`.) This procedure can be written in terms of `label` and `jump` as follows:

```
(define (cwcc proc)
  (label here
    (proc (lambda (val) (jump here val)))))
```

The `proc` argument is a unary procedure that is applied to an **escape procedure** that, when called, will return a result from the call to `cwcc`. Here is a version of `prod-list` written in terms of `cwcc`.

```
(define (prod-list a-list)
  (cwcc
    (lambda (return)
      (letrec ((prod (lambda (lst)
                       (cond ((null? lst) 1)
                             ((= 0 (car lst)) (return 0))
                             (else (* (car lst)
                                      (prod (cdr lst))))))))
        (prod a-list)))))
```

The advantage of `cwcc` as an interface to capturing continuations is that it does not require extending a language with any new special forms. The binding performed by `label` is instead handled by the usual binding mechanism (`lambda`), and a `jump` is encoded as a procedure call.

Some languages put restrictions on capturable continuations that make them easier to reason about and to implement. For example, the DYLAN language provides a `(bind-exit (`*I*`)` *E*`)` form that is similar to `(cwcc (lambda (`*I*`)` *E*`))` except that the lifetime of the escape procedure is limited by the lifetime of the `bind-exit` form. The `catch` and `throw` constructs of COMMON LISP are similar to `label` and `jump` except that `throw` jumps to a named control point declared by a dynamically enclosing `catch`. Dynamically declared control points are a good mechanism for exception handling, which is our next topic of study.

▷ **Exercise 9.8** What are the values of the following expressions? (Assume `prod-list` is defined as above.)

a. `(prod-list '(2 3 4))`

b. `(prod-list '(2 0 yow!))`

c.
```
(prod-list '(yow! 0 2))
```

d.
```
(let ((twice (lambda (f x) (f (f x)))))
  (let ((f (label bind-f (lambda (new-f)
                           (jump bind-f new-f)))))
    ((f twice) (+ 1) 0)))
```

e.
```
(jump (label a a) (label b b))
```
◁

▷ **Exercise 9.9** It is possible to implement loops with `label` and `jump` without using mutation. As an example, here is a template for an iterative factorial procedure in FL + {`label`, `jump`} (recall that FL does not support mutation):

```
(define (factorial n)
  (let ((triple E_triple))
    (let ((loop (first triple))
          (num  (second triple))
          (acc  (third triple)))
      (if (= num 0)
          acc
          (loop (list loop (- num 1) (* acc num)))))))
```

(Assume that `first`, `second`, and `third` are the appropriate list accessing procedures.) Using `label` and `jump`, write an expression E_{triple} such that `factorial` behaves as advertised. ◁

▷ **Exercise 9.10** Chris Krenshall[4] is dissatisfied with FLK!+{`label`,`jump`}. He's never sure where his thread of control will end up! Therefore, Chris would like you to give him some control over his control points. Chris wants to have *applets* — syntactically distinguished regions of code across which control points cannot be used. Here are the proposed FLK! extensions:

E ::= ... | `(applet` I E`)` | `(label` I E`)` | `(jump` E_1 E_2`)`

Informally, the `label` and `jump` constructs work as described above: `label` establishes first class control points and `jump` transfers control to them. However, there is one important difference, related to the `applet` construct: it is only legal to jump to control points created by the current applet, which is determined by the identifier of the nearest *lexically enclosing* `applet`.

For example, the following program is legal and evaluates to 0.

[4]Recall the C. Krenshall Program for eliminating concurrency from government programming contracts.

```
(applet hot
  (let ((p (applet cool
             (proc x
               (label cool-return
                 (+ (if (= x 1)
                        (jump cool-return 0)
                        x)
                    7))))))
    (p 1))) ⟶eval 0
```

On the other hand, the following program should signal an error:

```
(applet hot
  (let ((p (proc x
             (label hot-return
               (applet cool
                 (+ (if (= x 1)
                        (jump hot-return 0)
                        x)
                    7))))))
    (p 1))) ⟶eval error
```

In this problem you will modify the standard semantics for FLK! to specify the semantics of the `applet`, `label`, and `jump` constructs.

a. Suppose we define an *ControlPoint* domain and modify the *Value* domain accordingly:

$$q \in \mathit{ControlPoint} = \mathit{Applet} \times \mathit{Expcont}$$
$$a \in \mathit{Applet} = \text{Identifier}$$
$$v \in \mathit{Value} = \mathit{ControlPoint} + \ldots$$

Modify the signature of $\mathcal{E}$ as necessary to support applets.

b. Give a new definition for *top-level*. In your semantics, use the special applet identifer *global-applet* $\in$ *Applet* no applet has been defined for a `label` or `jump`.

c. Give the meaning function clause for `(applet` I E`)`.

d. Give the meaning function clause for `(label` I E`)`.

e. Give the meaning function clause for `(jump` E_1 E_2`)`. ◁

▷ **Exercise 9.11** This exercise explores the semantics of `cwcc` in more detail:

a. We have shown that `cwcc` can be written in terms of `label` and `jump`. Show how `label` and `jump` can be desugared in a language that provides `cwcc`.

b. Write a standard style valuation clause for the `cwcc` primitive.

c. Write a computation style valuation clause for the `cwcc` primitive. ◁

.

9.5 Exception Handling

A common reason to alter the usual flow of control in a program is to respond to exceptional conditions. For example, upon encountering a divide-by-zero error, the caller of the division procedure may want the computation to proceed with a large number rather than terminate with an error. Dynamically responding to exceptional conditions is known as **exception handling**.

One strategy for exception handling is for every procedure to return values that are tagged with a **return code** that indicates whether the procedure is returning normally or in some exceptional way. The caller can then test for the return code and handle the situation accordingly. Although popular, the return code technique is unsatisfactory in many ways. For one, it effectively requires every call to a procedure to explicitly test for all return codes the procedure could potentially generate. By treating normal and exceptional returns in the same fashion, return codes fail to capture the notion that exceptions are generally perceived as rare events compared to normal returns. In addition, return codes provide a very limited way in which to respond to exceptional conditions. All responsibility for dealing with the condition resides in the caller; in particular, the point at which the condition was generated has been lost.

An alternate way to view exceptional conditions is that procedures can **raise** (or **signal**) an **exception** as an alternative to returning a value. The immediate caller may then **handle** the exception, or it might decline to handle the exception and instead allow other callers in the current call chain to handle the exception. There are two basic strategies for handling the exception:

1. In **termination semantics**, the handler receives control from the signaler of the exception and keeps it. This is the approach taken by ML's `raise` and `handle`, Common Lisp's `throw` and `catch`, and CLU's `signal` and `except when`.

2. In **resumption semantics**, the handler receives control from the signaler of the exception but later passes control back to the computation that raised the exception. Operating system traps usually follow this model.

Some languages (such as CLU) require the caller to explicitly resignal user exceptions to propagate them up the call chain. In other languages, unhandled exceptions propagate up the call chain until an appropriate handler is found. In these languages, programs are implicitly wrapped in a default handler that handles otherwise uncaught exceptions.

As a concrete example of exception handling, we extend FLK! to accommodate a rudimentary resumption-style exception facility:

E ::= ... | `(raise` I_{except} E_{val}`)` | `(trap` I_{except} $E_{handler}$ E_{body}`)`

The informal semantics of these constructs is as follows:

- `(raise` I_{except} E_{val}`)` raises an exception named I_{except} with argument E_{val}.

- `(trap` I_{except} $E_{handler}$ E_{body}`)` evaluates E_{body} in such a way that if a `raise` of I_{except} is encountered during the evaluation of E_{body}, the value of the `raise` form is the result of applying the **handler procedure** computed by $E_{handler}$ to the argument supplied by the `raise`. If there is more than one handler with the same name, the one associated with the nearest dynamically enclosing `trap` is used. The value of the `trap` form is the value returned by E_{body}. If $E_{handler}$ does not designate a procedure, `trap` generates an error.

As an example of exception handling, consider an FL! `add` procedure that normally returns the sum of its two arguments, but raises a `non-integer` exception if one of its arguments is not an integer:

```
(define add
  (lambda (x y)
    (let ((check-integer
           (lambda (a)
             (if (integer? a) a (raise non-integer a)))))
      (+ (check-integer x) (check-integer y)))))
```

(Even better, we could change the semantics of the `+` primitive to raise exceptions rather than generate errors.) Now suppose we use `add` within a procedure that sums the elements of a list:

```
(define sum-list
  (lambda (lst)
    (if (null? lst)
        0
        (add (car lst) (sum-list (cdr lst))))))

(sum-list '(1 2 3)) ⟶ 6
```

If we call `sum-list` on a list containing non-integer elements, we can use `trap` to specify how these elements should be handled. For example, here is a handler that treats false as 0, true as 1, and all symbols as 10; elements that are not booleans or symbols abort with an error:

```
(define simple-handler
  (lambda (x)
    (cond ((boolean? x) (if x 1 0))
          ((symbol? x) 10)
          (else (error not-boolean-or-symbol)))))

(trap non-integer simple-handler
  (sum-list '(5 yes #t)))
```
$\xrightarrow[eval]{}$ *16*

```
(trap non-integer simple-handler
  (sum-list '(5 (yes no) #t)))
```
$\xrightarrow[eval]{}$ $error : not - boolean - or - symbol$

We will assume that exceptions not handled by any dynamically enclosing `trap`s are converted into errors by a default top-level exception handler; e.g.:

```
(sum-list '(5 #t yes))
```
$\xrightarrow[eval]{}$ $error : non - integer$

While the informal semantics for `raise` and `trap` given above may seem like an adequate specification, it harbors some ambiguities. For example, what should be the result of the following program?

```
(trap a (lambda (x) (+ 4000 x))
  (trap b (lambda (x) (+ 300 (raise a (+ x 4))))
    (trap a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2)))))
```

The `raise` of `b` invokes a handler that raises the exception `a`. But which of the two `a` handlers should be used?

Once again, formal semantics comes to the rescue. In fact, because complex control constructs can easily befuddle our intuitions, we look more than ever to the guidance of formal semantics. Standard semantics is an excellent tool for precisely wiring down the meaning of complex control constructs like `raise` and `trap`.

Our approach is to treat `trap` as a binding construct that associates an exception name with an exception handler in a dynamic environment. An exception handler is just a procedure. `raise` looks up the handler associated with the given exception name in the current dynamic environment and applies the resulting procedure to the argument of `raise`.

To express these extensions formally, we will modify the standard semantics of FLK!. Figures 9.20 and 9.21 summarize the changes needed to accommodate `raise` and `trap`. Exception handlers are represented as procedure values that are named in a special environment, *Handler-Env*. Augmenting computations with this handler environment treats them as dynamic (as opposed to lexical) environments. That is, the domain *Procedure*, which is *Denotable* $\rightarrow$

$c \in \mathit{Computation} = \mathit{Handler\text{-}Env} \to \mathit{Expcont} \to \mathit{Cmdcont}$
$w \in \mathit{Handler\text{-}Env} = \text{Identifier} \to \mathit{Procedure}$
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Computation}$; *As usual*

New auxiliaries for handler environments:

$\mathit{extend\text{-}handlers} : \mathit{Handler} - \mathit{Env} \to \text{Identifier} \to \mathit{Procedure} \to \mathit{Handler} - \mathit{Env}$
$= \lambda w I_1\, p\,.\ \lambda I_2\,.\ \textbf{if}\ (\mathit{same\text{-}identifier?}\ I_1\ I_2)\ \textbf{then}\ p\ \textbf{else}\ (w\ I_2)\ \textbf{fi}$

$\mathit{get\text{-}handler} : \mathit{Handler} - \mathit{Env} \to \text{Identifier} \to \mathit{Procedure}\ = \lambda w I\,.\,(w\ I)$

$\mathit{default\text{-}handlers} : \mathit{Handler} - \mathit{Env}\ = \lambda I\,.\ \lambda p\,.\ (\mathit{err\text{-}to\text{-}comp}\ I)$

New computation auxiliaries:

$\mathit{extending\text{-}handlers} : \text{Identifier} \to \mathit{Procedure} \to \mathit{Computation} \to \mathit{Computation}$
$= \lambda I p c\,.\ \lambda w\,.\ (c\ \ (\mathit{extend\text{-}handlers}\ w\ I\ p))$

$\mathit{getting\text{-}handler} : \text{Identifier} \to (\mathit{Procedure} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda I f\ .\ \lambda w\,.\ (f\ \ (\mathit{get\text{-}handler}\ w\ I)\ \ w)$

Modifications to other computation auxiliaries:

$\mathit{val\text{-}to\text{-}comp} : \mathit{Value} \to \mathit{Computation}\ = \lambda v\,.\ \lambda w k\,.\ (k\ v)$

$\mathit{err\text{-}to\text{-}comp} : \mathit{Error} \to \mathit{Computation}\ = \lambda I\,.\ \lambda w k s\,.\ (\mathit{Error} \mapsto \mathit{Expressible}\ I)$

$\mathit{with\text{-}value} : \mathit{Computation} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda c f\ .\ \lambda w k\,.\ (c\ \ w\ \ (\lambda v\,.\ (f\ \ v\ \ w\ \ k)))$

Other computation auxiliaries similarly pass around handler environments.

Figure 9.20: Semantics of `raise` and `trap`, Part I.

Valuation functions (standard version):

$$\mathcal{E}[\![(\texttt{trap}\ I_{except}\ E_{handler}\ E_{body})]\!]$$
$$= \lambda ewk\,.\ (\mathcal{E}[\![E_{handler}]\!]$$
$$e\ \ w\ \ (\textit{test-procedure}$$
$$(\lambda p\,.\ (\mathcal{E}[\![E_{body}]\!]\ e\ (\textit{extend-handlers}\ w\ I_{except}\ p)\ k))))$$

$$\mathcal{E}[\![(\texttt{raise}\ I_{except}\ E_{body})]\!]$$
$$= \lambda ewk\,.\ (\mathcal{E}[\![E_{body}]\!]\ e\ w\ (\lambda v\,.\ ((\textit{get-handler}\ w\ I_{except})\ v\ w\ k)))$$

Valuation functions (computation version):

$$\mathcal{E}[\![(\texttt{trap}\ I_{except}\ E_{handler}\ E_{body})]\!]$$
$$= \lambda e\,.\ (\textit{with-procedure}\ \ (\mathcal{E}[\![E_{handler}]\!]\ e)$$
$$(\lambda p\,.\ (\textit{extending-handlers}\ I_{except}\ p\ \ (\mathcal{E}[\![E_{body}]\!]\ e))))$$

$$\mathcal{E}[\![(\texttt{raise}\ I_{except}\ E_{body})]\!]$$
$$= \lambda e\,.\ (\textit{with-value}\ \ (\mathcal{E}[\![E_{body}]\!]\ e)\ \ (\lambda v\,.\ (\textit{getting-handler}\ I_{except}\ (\lambda p\,.\ (p\ v)))))$$

Figure 9.21: Semantics of `raise` and `trap`, Part II.

Computation, is equivalent to the following:

$$\textit{Procedure} = \textit{Value} \to \textit{Handler} - \textit{Env} \to \textit{Expcont} \to \textit{Store} \to \textit{Expressible}$$

The auxiliaries *extend-handlers*, *get-handler*, and *default-handlers* are versions of *extend*, *lookup*, and *empty-env* for *Handler-Env*. The auxiliaries *extending-handlers* and *getting-handler* capture manipulations of the *Handler-Env* component of the computation in an abstract way. If the computation abstractions are used, it is not necessary to modify any of the valuation clauses from the semantics of FLK!. However, valuation clauses written in the standard style would have to be modified to pass along an extra w argument.

The valuation clauses for `trap` and `raise` are presented in both the standard style and the computation style. `trap` simply extends the dynamic handler environment with a new binding and evaluates E_{body} with respect to this new environment. `raise` invokes the dynamically bound handler on the value of E_{body}. Note that *default-handlers* initially binds every exception name to a handler that converts an exception into an error. A handler procedure is called with the dynamic handler environment in effect at the point of the `raise`. This gives rise to the following behavior:

```
(trap a (lambda (x) (+ 4000 x))
  (trap b (lambda (x) (+ 300 (raise a (+ x 4))))
    (trap a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2))))) ⟶FL! 327

(trap a (lambda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4))))) ⟶FL! 421
```

Exception handling is an excellent example of the utility of dynamic scoping. Suppose `trap` were to bind I_{except} in a lexical environment rather than a dynamic one. Then `raise` could only be handled by lexically apparent handlers. It would be impossible to specify handlers for a procedure on a per call basis.

Termination semantics for exception handlers can be simulated by using `label` and `jump` in conjunction with `raise` and `trap`. For example, suppose we want a call to `sum-list` to abort its computation and return 0 if the list contains a non-integer. This can be expressed as follows:

```
(label exit
  (trap non-integer (lambda (x) (jump exit 0))
    (sum-list '(5 yes #t)))) ⟶FL! 0
```

Here the handler procedure forces the computation to abort to the `exit` point when the symbol `yes` is encountered.

An alternative to using `label` and `jump` in situations like these is to develop a new kind of handler clause:

(`handle` I_{except} $E_{handler}$ E_{body})

Like `trap`, `handle` dynamically binds I_{except} to the handler computed by $E_{handler}$. But unlike `trap` handlers, when a `handle` handler is invoked by `raise`, it uses the dynamic environment and continuation of the `handle` expression rather than the `raise` expression. For example:

```
(handle a (lambda (x) (+ 4000 x))
  (handle b (lambda (x) (+ 300 (raise a (+ x 4))))
    (handle a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2))))) ⟶FL! 4006

(handle a (lambda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4))))) ⟶FL! 40
```

We leave the semantics of `handle` as an exercise (`raise` need not be changed).

▷ **Exercise 9.12** Sam Antix decides to add the new `handle` exception handling primitive to FL! + {`raise`, `trap`}. He adds alters the grammar of FL! + {`raise`, `trap`}:

(`handle` I_{except} $E_{handler}$ E_{body})

As we described above, Sam's new expression is similar to

$$\texttt{(trap } I_{except} \ E_{handler} \ E_{body}\texttt{)}.$$

Both expressions evaluate $E_{handler}$ to a handler procedure and dynamically install the procedure as a handler for exception I_{except}. Then the body expression E_{body} is evaluated. If E_{body} returns normally, then the installed handler is removed, and the value returned is the value of E_{body}.

However, if the evaluation of E_{body} reaches an expression

$$\texttt{(raise } I_{except} \ E\texttt{)},$$

then E is evaluated and the handler procedure is applied to the resulting value. With `trap`, this application is evaluated at the point of the `raise` expression. But with `handle`, the application is evaluated at the point of the `handle` expression. In particular, both the dynamic environment and continuation are inherited from the `handle` expression, *not* the `raise` expression.

Here is another example besides the one given above:

```
(handle a (lambda (x) (* x 10))
  (+ 1 (raise a (+ 2 (raise a 4))))) ⟶eval 40
```

a. Extend the denotational semantics of call-by-value FLK! + {`raise`, `trap`} with a meaning function clause for `handle` (the meaning function clause for `raise` doesn't need to be changed).

b. Give a desugaring of `handle` into FL! + {`raise`, `trap`, `label`, `jump`}. ◁

▷ **Exercise 9.13** Ben Bitdiddle, whose company is fighting for survival in the competitive FL! market, has an idea for getting ahead of the competition: adding recursive exception handlers! He wants to extend FLK! as follows:

E ::= ... `existing` FLK! `constructs` ...
| `(handle` I_{except} $E_{handler}$ E_{body}`)`
| `(rec-handle` I_{except} $E_{handler}$ E_{body}`)`
| `(raise` I_{except} E_{val}`)`

The `handle` and `raise` constructs are unchanged from Exercise 9.12. Informally, the `rec-handle` construct has the following semantics: first, $E_{handler}$ is evaluated to a procedure p (it is an error if it evaluates to something that is not a procedure). Next, E_{body} is evaluated; if it raises the exception I_{except}, the procedure p handles it. So far, `rec-handle` is identical to `handle`. However, if the execution of p raises the exception I_{except}, this exception is handled by p. The exceptions have termination semantics.

Here's a short example that Ben has prepared for you:

```
(rec-handle I
  (lambda (x)
    (if (= x 0) 1 (raise I (- x 1))))
  (raise I 5)) ⟶eval 1
```

Give the meaning function clause for `(rec-handle` I_{except} $E_{handler}$ E_{body}`)` (the semantic domains and the meaning function clauses for `handle` and `raise` remain unchanged). ◁

▷ **Exercise 9.14** Alyssa P. Hacker really likes the exception system of a certain Internet applet language. She has decided to add that exception system to her favorite language, FL!. In particular, she wants to modify the grammar of FLK! as follows:

E ::= ... `existing` FLK! `constructs (except proc)` ...
| `(handle` I_{except} $E_{handler}$ E_{body}`)`
| `(raise` I_{except} E_{val}`)`
| `(finally` E_{body} $E_{finally}$`)`
| `(proc` I_{except} I E`)`
| `(try` E_{body} `((`I_{except} I $E_{handler}$`)*)` $E_{finally}$`)`

Note: To improve the readability of the examples, all the exception identifiers used in this exercise start with the character `%`.

The `handle` and `raise` constructs are old friends, but the others are new. Here are their informal semantics:

- `(handle` I_{except} $E_{handler}$ E_{body}`)` establishes an exception handler for exception I_{except}. First $E_{handler}$ is evaluated to a handler procedure. Then E_{body} is evaluated. If E_{body} raises I_{except}, the exception handler is called with the value of the exception, and the value returned by the handler is returned by the `handle` expression. (That is, `handle` provides *termination* semantics.)

- `(raise` I_{except} E_{val}`)` passes the value of E_{val} to the exception handler for I_{except}. A `raise` expression never returns a value, because the `handle` expression provides termination semantics.

- `(finally` E_{body} $E_{finally}$`)` ensures that $E_{finally}$ is evaluated. Specifically, it evaluates E_{body} and either (1) returns its value or (2) propagates the exception it raises. Whether E_{body} returns normally or via an exception, $E_{finally}$ is evaluated before `finally` returns. The value of $E_{finally}$ is discarded.

  For example, the following expression always closes the input file:

```
(let ((port (open-input-file "foo.txt")))
  (handle %invalid (lambda (value)
                     (begin
                       (display "invalid: ")
                       (display value)
                       (newline)
                       'invalid))
    (finally (let ((value (read port)))
               (if (valid? value)
                   value
                   (raise %invalid value)))
      (close-input-file port))))
```

If the file's contents are valid, the expression returns the file's contents. Otherwise, the exception `%invalid` is raised, the handler prints a message and then returns the symbol `invalid` instead of the file's contents. No matter what happens, `close-input-file` is called to clean up.

- (`proc` I_{except} I E) returns a procedure of one argument. However, the procedure is guaranteed to raise only exception I_{except} or `%illegal-exception`. If its body raises any other exceptions, they are converted to `%illegal-exception`.

  For example, the following procedure raises the `%f` exception if its argument is `#f`. Otherwise it automatically raises the `%illegal-exception` exception:

  ```
  (proc %f b (if b (raise %t b) (raise %f b)))
  ```

- (`try` E_{body} ((I_{except} I E)*) $E_{finally}$) corresponds to the `try-catch-finally` construction in a certain unmentionable language. It works like this:

  - E_{body} is evaluated and, if it doesn't raise any exceptions, its value is returned as the value of the `try` expression. However, if an exception is raised by E_{body}, the (I_{except} I E) clauses are consulted to handle the exception.
  - If an exception I_{except} is raised by E_{body}, the corresponding variable I is bound to the value of the exception and the corresponding expression E is evaluated. The value of the `try` expression becomes the value of E in this case.
  - Regardless of whether or not an exception is raised by E_{body}, the expression $E_{finally}$ is evaluated immediately before control leaves the `try` expression. Its value is discarded.

As usual, Alyssa has vanished, probably to another Internet startup. She's left you with a helper function, *insert-procedure*. Informally, *insert-procedure* takes a handler environment, an identifier predicate, and a procedure. It returns a new handler environment that *conditionally* inserts the procedure at the beginning of the handler chain: the procedure is executed for any exception that satisfies the corresponding identifier predicate and was not caught yet by a handler. For example,

(*insert-procedure* w $(\lambda I_{except}\,.\,true)$ p)returns a handler environment that is like w except that p will be called first on *every* exception. Here is the signature and the definition of *insert-procedure*:

$$\begin{aligned}
&\textit{insert-procedure}: \textit{Handler-Env} \to (\text{Identifier} \to \textit{Bool}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \textit{Procedure} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \textit{Handler-Env} \\
&= \lambda w f\, p\,.\,(\lambda I_{except}\,.\,\textbf{if}\,(f\ \ I_{except}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \lambda\delta w' k\,.\ (p\ \ \delta\ \ w'\ \ (\lambda v\,.\,((w\ \ I_{except})\ \ \delta\ \ w'\ \ k))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (w\ \ I_{except}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}
\end{aligned}$$

You may find *insert-procedure* useful in solving the following problems:

a. Give the meaning function clause for `finally`.

b. Give the meaning function clause for `proc`.

c. Give a desugaring for

$$\texttt{(try}\ \ E_{body}\ \ \texttt{((}I_{except\_1}\ \ I_1\ \ E_1\texttt{)}\ \ \ldots\ \ \texttt{(}I_{except\_n}\ \ I_n\ \ E_n\texttt{))}\ \ E_{finally}\texttt{)}$$

into `handle`, `raise`, `proc`, and `finally`. Assume that the handler expression E_i is permitted to raise only the exception it handles, I_{except_i}. If it raises any other exceptions, they are automatically converted to `illegal-exception`. ◁

▷ **Exercise 9.15** Sam Antix thinks that exception handlers should be able to choose dynamically between termination or resumption semantics. Sam likes the termination semantics of `handle` (Exercise 9.12), but occasionally he would prefer resumption semantics. He decides to extend FL!+{`raise`, `handle`} with a new construct `(resume` E`)`.

$$\begin{aligned}
E ::= &\ \ldots && \text{[As before]} \\
&\mid \texttt{(resume}\ E\texttt{)} && \text{[Resume at point of most recent raise]}
\end{aligned}$$

Informally, `(resume` E`)` will cause a handler to resume at the point of the `raise` rather than terminating at the point of the `handle`. `resume` first evaluates E using the current dynamic handler environment and then returns control to the point of the most recent `raise` with the value of E. Further, any program that does not use `resume` should behave just as it would in FL!+{`raise`, `handle`}.

Sam came up with some short examples demonstrating his new `(resume` E`)` construct:

```
(handle exn
     (lambda (x) (+ x 2))
   (+ 20 (raise exn 1))) ⟶eval 3

(handle exn
     (lambda (x) (resume (+ x 2)))
   (+ 20 (raise exn 1))) ⟶eval 23

(resume 7) ⟶eval error : no − raise

(let ((f (lambda (x) (resume (+ x 4)))))
   (handle exn
        (lambda (x) (f (+ x 2)))
     (+ 20 (raise exn 1)))) ⟶eval 27
```

When `resume` is invoked, any pending computation in the handler is discarded, including any other `resume`s:

```
(handle exn
     (lambda (x) (resume (+ 300 (resume (+ x 2)))))
   (+ 20 (raise exn 1))) ⟶eval 23

(handle exn1
     (lambda (x) (+ 50000 (resume (+ x 4))))
   (+ 4000 (handle exn2
              (lambda (x) (+ 300 (raise exn1 (+ x 2))))
             (+ 20 (raise exn2 1))))) ⟶eval 4307
(handle exn1
     (lambda (x) (+ 50000 (resume (+ x 4))))
   (+ 4000 (handle exn2
              (lambda (x)
                 (resume (+ 300 (raise exn1 (+ x 2)))))
             (handle exn1
                  (lambda (x) (+ 600000 x))
               (+ 20 (raise exn2 1)))))) ⟶eval 4327
```

Now handlers can choose between termination and resumption semantics:

```
(define example-fctn
   (lambda (argument)
      (handle exn
           (lambda (x) (if (< x 3)
                           (+ x 300)
                           (+ 500000 (resume (+ x 4000)))))
         (+ 20 (raise exn argument)))))
```

`(example-fctn 2)` $\xrightarrow[eval]{}$ *302*

`(example-fctn 4)` $\xrightarrow[eval]{}$ *4024*

a. Unfortunately, Sam has fallen ill. You must flesh out his design. You should extend the standard semantics for FL!+{`raise`, `handle`} as follows.

   i. Give the signature of $\mathcal{E}$ and the definition of the semantic domain *Procedure*.

   ii. Give the meaning function clauses for `raise`, `handle`, and `resume`.

b. The `trap` construct presented above has resumption semantics. It is possible to translate FL!+{`trap`, `raise`} into FL!+{`handle`, `raise`, `resume`}. We emphasize that this is a *translation* between two different languages and *not* a desugaring from a language to itself.

   The translation of most expressions merely requires translating their subexpressions. For example,

   $\mathcal{T}[\![$`(call` E_1 E_2`)`$]\!] =$`(call` $\mathcal{T}[\![E_1]\!]$ $\mathcal{T}[\![E_2]\!]$`)`

   Give the translation for `trap` and `raise`. ◁

▷ **Exercise 9.16** Consider a lexically-scoped, call-by-value variant of FLK, with the following twist: it has a `switch` construct that allows a program to both generate and handle exceptions. Here is the complete grammar:

E ::= L | I | `(if` E_1 E_2 E_3`)`
| `(proc` I E_B`)` | `(call` E_0 E_1`)`
| `(switch` E`)`

The idea behind `switch` is that every expression is implicitly provided with two continuations called A and B. All expressions pass their return value to the A continuation. The top level meaning function $\mathcal{TL}$ is modified to call $\mathcal{E}$ by passing the identity function as the A continuation (in order to get back the normal result), and an error handler as the B continuation. Thus, the A continuation usually corresponds to a normal return, while the B continuation usually corresponds to an exceptional return. We will call values that are passed to A continuations "A values" and values that are passed to B continuations "B values."

The `switch` form is used to swap the A and B continuations during the evaluation of its component expression. `switch` can be used to both generate and handle exceptions. Here are two example uses of `switch` (assuming the usual desugarings):

```
(define (my-func i j)
  (if (< 10 (+ i j))            ; make sure that  i+j > 10
      (g i j)                   ; compute the result
      (switch "size")))         ; else "size" error

(switch
  (let ((bval (switch (my-func 3 4))))
    (switch
      (if (string-equal? bval "size")
          1          ; return 1 for size error
          0))))      ; return 0 for other errors
```

In the last example, the `switch` around the application of `my-func` will cause B values of `my-func` to be bound to the variable `bval`. If `my-func` has an A value (a normal value) then the `switch` around `my-func` will cause execution to bounce out to the outermost `switch`, where the A value returned by `my-func` will be the A value of the entire expression. If a B value is returned by `my-func`, the entire expression will have an A value of either `1` or `0`, depending on the B value returned by `my-func`.

a. Construct the standard semantics for the language described above: Give the signature of $\mathcal{E}$ and the definition of the *Procedure* domain. Also give the meaning function clauses for `switch` and `call`. Either write out the other meaning function clauses, or describe how the corresponding clauses from the semantics for FLK! would be modified.

b. Using your semantics, prove that `(switch (switch` E`))` has the same meaning as E.

c. Suppose we have a (strict) `pair` construct to make a pair of two values, and the operations `left` and `right` to select the first and second values of a pair respectively. Then we might define an FLK!-like `raise` construct by the following desugaring:

$$\mathcal{D}(raise I_{except} E_{val}) = \texttt{(switch (pair (symbol } I_{except}\texttt{)} \ \mathcal{D}E_{val}\texttt{))}$$

Give a corresponding desugaring for `(handle` I_{except} $E_{handler}$ E_{body}`)`. Does your solution implement termination or resumption semantics? Explain. ◁

Reading

The notion of **continuation** was developed in the early 1970's by Christopher Strachey, Christopher Wadsworth, F. Lockwood Morris, and others. See [SW74]

which was more recently reprinted in [SW00] (that issue of *Higher-Order and Symbolic Computation* was dedicated to Strachey's work). Subsequently, continuations played an important role in actor languages [Hew77] and in SCHEME [SS76, Ste78].

For more information on control, continuations, and denotational semantics, see John Reynolds's history of continuations [Rey93], David Schmidt's textbook on denotational semantics [Sch86b], as well as Joseph Stoy's coverage of continuations in denotational semantics [Sto77, esp. pp. 251ff]. Stoy makes the argument that it is better for expressing the meaning of programs to embed continuations in the semantics rather than syntactically transforming programs into continuation passing style and using direct semantics.

For transforming programs into continuation-passing style (which we will explore further in Chapter 17), see [SF93, FSDF93, SF92].

Continuations can be used to understand other control structures, such as `shift` and `reset` [DF92] and the `amb` (for "ambiguous") operator [McC67, Cli82].

For more information on coroutines, see Melvin Conway coroutines [Con63]. See also C. A. R. Hoare's classic text on Communicating Sequential Processes, [Hoa85], as well as the OCCAM reference manual [occ95] and the description of JCSP in [Lea99].

Chapter 10

Data

Conjunction Junction, what's their function?
I got "and"... and "or",
They'll get you pretty far.

"And": That's an additive, like "this and that". ...
And then there's "or":
O-R, when you have a choice like "This or that".
"And"... and "or",
Get you pretty far.

— *Conjunction Junction (Schoolhouse Rock), Bob Dorough*

Here's hoping we meet now and then
It was great fun
But it was just one of those things.

— *Jubilee, Cole Porter*

Well-designed data structures can make programs efficient, understandable, extensible, secure, and easy to debug. For this reason, programmers focus much of their energy on designing and using data structures. How successful they are depends in part on the tools provided by their programming language for declaring and manipulating data. This chapter explores some of the key data dimensions in programming languages.

10.1 Products

Products are compound values that result from gluing other values together. They are data structures that correspond to the product domains we have been using in our mathematical metalanguage (see Section A.3.2) to represent structured mathematical values with components. Standard examples of products are 2-dimensional points (consisting of x and y components), employee records (consisting of name, sex, age, identification number, hiring date, etc.), and the sequences of points in a polygon.

There are a wide variety of product data structures in programming languages that differ along a surprising number of dimensions:

- How are product values created and later decomposed into parts?
- Are the components of the product indexed by position or by name?
- When accessing a component, can its index be calculated or must an index be a manifest constant?
- Are the components values (as in call-by-value) or computations (as in call-by-name/call-by-need)?
- Are the components of the product immutable or mutable?
- Is the length of the product fixed or variable?
- Are all components of the product required to have the "same type," i.e., are products *homogeneous*?
- When products are nested, are the nested components all required to have the same size and/or "shape"?
- How are products passed as arguments, returned as results, and stored in assignments?
- Can the lifetime of a product exceed the lifetime of an invocation of a procedure in which it is created?

In this section, we will explore many of these dimensions, using our operational and denotational tools where appropriate to explain interesting points in the design space of products.

Products are known by a confusing variety of names — such as *array*, *vector*, *sequence*, *tuple*, *string*, *list*, *structure*, *record*, *environment*, *table*, *module*, and *association list* — that are used inconsistently between languages. We shall be

using some of these names in our study of products, but it is important to keep in mind that our use of a name may denote a different kind of product than what you might be familiar with from your programming experience.

10.1.1 Positional Products

10.1.1.1 Simple Positional Products

The simplest kind of product is a pair, which glues two values together. In Chapter 6, we studied the semantics of pairs in call-by-name and call-by-value versions of FL.

Pairs are an example of a **positional product**, in which component values are indexed by their position in the product value. We can extend pairs into more general positional products by adding the following two constructs to call-by-value FL![1]:

E ::= ...
| `(product` E^*`)` [Product Creation]
| `(proj` N E`)` [Product Projection]

The expression `(product` E_1 ... E_n`)` constructs an immutable positional product value whose n components are the values of the subexpressions E_1 through E_n. Such a value is traditionally known as a **tuple**. `(proj` N E_{prod}`)` extracts the component of the tuple denoted by E_{prod} that is at literal index N, where the components of an n-component product are indexed from 1 to n. An attempt to extract a component outside this index range is an error. The name `proj` is short for "project," the verb traditionally used to extract the component of a product.

An operational semantics of immutable positional products in call-by-value FL! is presented in Figure 10.1. A `product` expression with value expression components is considered a new kind of value expression. The [*product-progress*] rule evaluates the subexpressions of a `product` expression, so that the expression

```
(product (= 0 1) (* 2 3) (+ 3 4))
```

evaluates to the value expression

```
(product false 6 7).
```

The [*product projection*] rule extracts the value component at index N. If N is not in the valid range of indices, the `proj` expression is stuck, modeling an error. Using these rules, it is straightforward to show that the following FL expression evaluates to *9*:

[1] We study products in the context of a stateful language to facilitate coverage of product dimensions that involve state.

$$V \in \text{ValueExp} = \ldots \cup \{\texttt{(product } V_1 \ \ldots \ V_n\texttt{)}\}$$

$$\frac{\langle E_i, S\rangle \Rightarrow \langle E_i\,', S'\rangle}{\langle \texttt{(product } V_1 \ \ldots \ V_{i-1} \ E_i \ E_{i+1} \ \ldots \ E_n\texttt{)}, S\rangle \Rightarrow \langle \texttt{(product } V_1 \ \ldots \ V_{i-1} \ E_i\,' \ E_{i+1} \ \ldots \ E_n\texttt{)}, S'\rangle} \quad [\textit{product progress}]$$

$$\langle \texttt{(proj } N \texttt{ (product } V_1 \ \ldots \ V_n\texttt{))}, S\rangle \Rightarrow \langle V_N, S\rangle, \text{ where } 1 \leq N \leq \text{n} \quad [\textit{product projection}]$$

Figure 10.1: Operational semantics of immutable positional products in CBV FL!.

```
(let ((p (product (= 0 1) (* 2 3) (+ 4 5))))
  (if (proj 1 p) (proj 2 p) (proj 3 p)))
```

The corresponding denotational semantics of positional products in call-by-value FL! is presented in Figure 10.2. The *Value* domain is extended with a new summand, *Prod*, whose elements — sequences of values — represent product values. The evaluation of the subexpressions of a `product` expression is handled by *with-values*, and *nth* is used to extract the component at a given index in a `proj` expression. The validity of the index N is determined by the predicate

$$1 \leq (\mathcal{N}\ N) \textbf{ and } (\mathcal{N}\ N) \leq (\mathit{length}\ v^*),$$

which is known as a **bounds check**. If the bounds check fails, the `proj` expression denotes an error.

In many programming languages, the size of all products is known by the implementation, and a bounds check for every projection can be performed either at compile time or at run time. Important exceptions are C and C++, in which arrays carry no size information and bounds checks are not performed when array components are accessed. Programmers in these languages must pass array size information separately from the array itself and are expected to perform their own bounds checks. The lack of automatic bounds checks in C/C++ is the root cause of a high percentage of security flaws in modern software applications, many of which are due to so-called **buffer overrun** exploits that take advantage of C's permissiveness to fill memory with malicious code that can then be executed by a privileged process.

Product values with n components are often drawn as a box with n slots, sometimes with explicit indices. For example, the three-component product from above would be drawn as

false	*6*	*7*
1	2	3

$$
\begin{array}{rcl}
 & \mathit{Prod} & = \quad \mathit{Value}^* \\
v \in & \mathit{Value} & = \quad \ldots + \mathit{Prod}
\end{array}
$$

$$\mathcal{E}[\![(\texttt{product}\ E^*)]\!] = \lambda e\,.\ (\mathit{with\text{-}values}\ (\mathcal{E}^*[\![E^*]\!]\ e)\ \ \mathit{Prod} \mapsto \mathit{Value})$$

$$
\begin{array}{l}
\mathcal{E}[\![(\texttt{proj}\ N\ E_{prod})]\!] = \\
\quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{prod}]\!]\ e) \\
\qquad (\lambda v\,.\ \textbf{matching}\ v \\
\qquad\quad \triangleright\, (\mathit{Prod} \mapsto \mathit{Value}\ v^*)\ [\!]\ \textbf{if}\ 1 \leq (\mathcal{N}\ N)\ \textbf{and}\ (\mathcal{N}\ N) \leq (\mathit{length}\ v^*) \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ (\mathit{val\text{-}to\text{-}comp}\ \ (\mathit{nth}\ \ (\mathcal{N}\ N)\ \ v^*)) \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{error\text{-}comp} \\
\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
\qquad\quad \triangleright\, \textbf{else}\ \mathit{error\text{-}comp} \\
\qquad\quad \textbf{endmatching}\ )
\end{array}
$$

Figure 10.2: Denotational semantics of immutable positional products in CBV FL!.

Such a diagram suggests a low-level implementation in which the n components of a product are stored as the contents of n successive addresses in the memory of the computer, something we shall explore in more detail in Chapter 17.

We emphasize that tuples in FL! are immutable: there is no way to change the value stored in a slot. We will consider mutable products later. In the following subsections, we discuss many possible variants to the products presented above.

10.1.1.2 Sequences

In the projection expression considered above, the index is an integer literal, not an integer expression. This means that the projection index cannot be calculated. One benefit of this restriction is that the bounds check can always be performed at compile time and so need never be performed at run time. As we discuss later (page 423), literal indices also facilitate reasoning about programs written in statically typed languages.

The positional products studied above do not include any way to dynamically determine the size of the product, i.e., the number of components. It is assumed that the programmer knows the size of every tuple when writing the program. However, there are many situations where it is necessary or convenient to determine the size of a product and to extract a product component at an index calculated from an expression. For instance, given an arbitrary product containing numbers, determining the average of these numbers requires know-

ing the number of components and looping through all indices of the product to find the sum of the components. Such capabilities are normally associated with products called **arrays**. But since arrays usually imply mutable structures, we will instead use the name **sequence** for immutable products with calculated indices and dynamically determinable sizes. This terminology is consistent with our use of the term "sequence" in our mathematical metalanguage.

We can extend FL! with immutable sequences by adding the following constructs to the language:

E ::= ...
| `(sequence` E^*`)` [Sequence Creation]
| `(seq-proj` E_{index} E_{seq}`)` [Sequence Projection]
| `(seq-size` E_{seq}`)` [Sequence Size]

The expression `(sequence` E_1 ... E_n`)` creates and returns a size-n sequence whose components are the values of the expressions E_1 through E_n. The expression `(seq-proj` E_{index} E_{seq}`)` returns the ith component of the sequence denoted by E_{seq}, where i is the integer denoted by the arbitrary expression E_{index} (which must be checked against the bounds of the sequence). The expression `(seq-size` E_{seq}`)` returns the number of components in the sequence. The formal semantics of sequences is left as an exercise.

As an example of sequence manipulation, here is a procedure that finds the average of a sequence of numbers:

```
(define (average s)
  (letrec ((n (seq-size s))
           (sum-loop
             (lambda (i sum)
               (if (= i 0)
                   (/ sum n)
                   (sum-loop (- i 1)
                         (+ sum (seq-proj i s)))))))
    (sum-loop n 0)))
```

10.1.1.3 Product indexing

The positional products discussed above use **1-based indexing**, in which the components of an n-component tuple are accessed via the indices $1 \ldots n$. Many languages instead have **0-based indexing**, in which the slots are accessed via the indices $0 \ldots n-1$. For example:

false	*6*	*7*
0	1	2

Why use the index 0 to access the first slot of a product? One reason is that it can simplify some addressing calculations in the compiled code, which

results in the execution of fewer low-level instructions when projecting components from products. Another reason is that 0-based indexing simplifies certain addressing calculations for the programmer. For example, compare the following expressions for accessing the slot in row i and column j of a conceptually 2-dimensional matrix `m` with width w and height h that is actually represented as a one-dimensional sequence with $w \times h$ components stored in so-called row-major order[2]:

```
; 0-based indexing of matrices and sequences
(nth (+ (* w i) j) m)

; 1-based indexing of matrices and sequences
(nth (+ (* w (- i 1)) j) m)
```

The 0-based approach is simpler because it does not require the subtraction by 1 seen in the 1-based approach.

Using 0 or 1 as the index for the first component are not the only choices. Some languages allow using any integer range for product indices. Some, such as PASCAL, even allow using as index ranges any range of values that is isomorphic to an integer range. For instance, a PASCAL array can be indexed by the alphabetic characters from `'p'` to `'u'` or the days of the week from `monday` through `friday` (where an enumeration of days has been declared elsewhere).

10.1.1.4 Types

FL is a **dynamically typed language** in which each value is conceptually tagged with its type and type errors are not detected until the program is run. In contrast, many modern languages are **statically typed languages**, in which the type of every expression is known when the program is compiled. The goal of static typing affects the design of positional products in these languages. In particular, it must be possible for the compiler to determine the type of every value projected from a product value.

For instance, ML and HASKELL support so-called **heterogeneous tuples** in which each tuple component may have a different type. In order to determine the type of a projection, both tuple indices and sizes must be statically determinable. ML's vectors, HASKELL's arrays, and CLU's sequences are examples of updatable immutable products in typed languages. Since in general compilers cannot determine either the size of or the index of a projection from such products, these products are **homogeneous sequences** in which all components are

[2] **Row-major** order means the elements of each row are stored in consecutive locations in the sequence. This makes accessing elements along a row very inexpensive. Likewise, **Column-major order** means elements of each column are stored in consecutive sequence locations.

required to have the same type. Many languages treat homogeneous sequences of characters, known as **strings**, as a special kind of positional product. Immutable strings appear in languages such as JAVA, ML, HASKELL and CLU, while C and SCHEME provide mutable strings.

Product indices are usually restricted to the integer type, but, as mentioned above, some languages allow index types that are isomorphic to the integers or some finite range of the integers. For example, the HASKELL language allows arrays to be indexed by any type that provides the operations of an "indexable" type. In PASCAL, arrays can be indexed by any range type that is isomorphic to a finite integer range. Oddly, the index range (not merely the index type) is part of the array type in PASCAL, which means that the size of every PASCAL array is statically known, and it is not possible to write procedures that are parameterized over arrays of different lengths.

In later chapters, we will have much more to say about product types when we study types in more detail.

10.1.1.5 Specialized syntax

Many languages provide specialized syntax for product manipulation. For instance, ML tuples are constructed by comma separated expressions delimited by parentheses, and the ith tuple component is extracted via the syntax `#`i. Here is an ML version of our earlier s-expression example:

```
let val p = (0 = 1, 2 * 3, 4 + 5)
 in if #1(p) then #2(p) else #3(p)
end
```

For sequences (as well as for mutable arrays) a subscripting notation using square brackets is a standard way to project components, and `:=` might be used for an update operation. Below is a way to swap the components at indices `i` and `j` of an immutable vector `u` in a hypothetical version of ML extended with this specialized syntax:

```
let val v = u[i]
    val u2 = u[i] := u[j]
    val u3 = u2[j] := v
 in u3
end
```

▷ **Exercise 10.1** In the first three parts below, assume 1-based indexing.

a. Give an operational semantics for sequences in call-by-value FL.

b. Give a denotational semantics for sequences in call-by-value FL.

c. Explicitly enumerating the elements of a sequence in a `sequence` expression can be inconvenient. For example, a sequence of the squares of the integers from 1 to 5 would be written:

```
(sequence (* 1 1) (* 2 2) (* 3 3) (* 4 4) (* 5 5))
```

An alternative means of specifying such sequences is via a new construct

(`tabulate` E_{size} E_{proc})

where E_{size} denotes the size of the sequence and E_{proc} denotes a unary procedure f that maps the index i to the value (f i). For instance, using `tabulate`, the above 5-element sequence could be written

```
(tabulate 5 (lambda (i) (* i i)))
```

Given an operational and denotational semantics of `tabulate` in call-by-value FL.

d. What changes would need to be made to the above three parts to specify 0-based indexing rather than one-based indexing?

e. What changes would need to be made to the syntax for sequences and the first three parts to specify an indexing scheme that starts at an arbitrary dynamically determinable value *lo* rather than 0 or 1? ◁

▷ **Exercise 10.2** Even with immutable products, it is still useful to provide a facility for updating elements in, inserting elements into, and removing elements from a product. Since the product is immutable, none of these operations actually change a given product value, but they return a new product value that shares most of its components with the given product value. We shall call sequences that support one or more of these operations **updatable sequences**, though this is by no means a standard term.

Consider the following constructs for one form of updatable sequence:

E ::=	...	
	| (`usequence` E^*)	[Updatable Sequence Creation]
	| (`useq-proj` E_{index} E_{useq})	[Updatable Sequence Projection]
	| (`useq-size` E_{useq})	[Updatable Sequence Size]
	| (`useq-update` E_{index} E_{val} E_{useq})	[Updatable Sequence Update]
	| (`useq-insert` E_{index} E_{val} E_{useq})	[Updatable Sequence Insertion]
	| (`useq-delete` E_{index} E_{useq})	[Updatable Sequence Deletion]

The `usequence`, `useq-proj`, and `useq-size` are the updatable sequence versions of the corresponding (non-updatable) sequence constructs. For `useq-update`, `useq-insert`, and `useq-delete`, suppose that E_{index} denotes an integer i, E_{val} denotes a value v_{new}, E_{useq} denotes a size-n updatable sequence v_{useq}, and `u` denotes the sequence with integer values `[7,5,8]`. If v is an updateable sequence, let $\#v$ denote the size of v and $v \downarrow j$ denote the jth component value of v, where $1 \leq j \leq \#v$. Then:

- if $1 \leq i \leq n$, (`useq-update` E_{index} E_{val} E_{useq}) returns a size-n updatable sequence v_{useq2} such that

$v_{useq2} \downarrow i = v_{new}$; and
$v_{useq2} \downarrow j = v_{useq} \downarrow j$ for all $1 \leq j \leq n$ where $j \neq i$.

For example, `(useq-update 2 6 u)` returns the updatable sequence `[7,6,8]`.

- if $1 \leq i \leq n+1$, `(useq-insert` E_{index} E_{val} E_{useq}`)` returns a size-$n+1$ updatable sequence v_{useq2} such that

  $v_{useq2} \downarrow j = v_{useq} \downarrow j$ for all $1 \leq j < i$;
  $v_{useq2} \downarrow i = v_{new}$; and
  $v_{useq2} \downarrow k = v_{useq} \downarrow k - 1$ for all $i < k \leq n + 1$;

  For example, `(useq-insert 2 6 u)` returns the updatable sequence `[7,6,5,8]`.

- if $1 \leq i \leq n$, `(useq-delete` E_{index} E_{useq}`)` returns a size-$n-1$ updatable sequence v_{useq2} such that

  $v_{useq2} \downarrow j = v_{useq} \downarrow j$ for all $1 \leq j < i$; and
  $v_{useq2} \downarrow k = v_{useq} \downarrow k + 1$ for all $i \leq k \leq n - 1$;

  For example, `(useq-delete 2 u)` returns the updatable sequence `[7,8]`

a. Give an operational semantics for updatable sequences in call-by-value FL.

b. Give a denotational semantics for updatable sequences in call-by-value FL.

c. Show that `useq-update` is not strictly necessary in a language with updatable sequences because it can be desugared into other constructs.

d. Consider a language with updatable sequences that also has a `(useq-empty)` construct that returns an empty updatable sequence. Show that `usequence` is not strictly necessary in such a language because it can be desugared into other constructs. What are the benefits and drawbacks of such a desguaring? ◁

10.1.2 Named Products

In a **named product**, components are indexed by names rather than by positions. In Section 7.2, we introduced the **record**, a classic form of named product, and studied its semantics. We saw that records were effectively reified environments. Here we discuss some of the dimensions of named products.

The simplest form of named product is a named version of positional products with a product creator (`record`) and a product projector (`select`):

E ::= ...
| `(record (`I E`)*)` [Record Creation]
| `(select` I E`)` [Record Projection]

As above, we assume that such constructs are embedded in a call-by-value language and denote immutable products.

As a simple example of records, consider the following expression, which evaluates to *9*:

```
(let ((r (record (test (= 0 1)) (yes (* 2 3)) (no (+ 4 5)))))
  (if (select test r) (select yes r) (select no r)))
```

In named products, the order of bindings in the record constructor is irrelevant, so the value of the above expression would not be changed if the `record` subexpression were changed to be

```
(record ((no (+ 4 5)) (test (= 0 1)) (yes (* 2 3))))
```

Many languages with named products have special syntax for record creation and projection. For instance, here is our running example expressed in ML record syntax:

```
let val r = {test = (0=1), yes=2*3, no=4+5}
 in if #test(r) then #yes(r) else #no(r)
end
```

A more common syntax for record selection is the "dot notation" used with PASCAL records, C structures, and JAVA objects, as in:

```
if r.test then r.yes else r.no
```

In a language like ML that permits numeric record labels, positional products can be viewed as syntactic sugar for named products. E.g., the ML tuple `(true, 17)` is syntactic sugar for {`1=true, 2=17`}.

Simple records can be augmented with operations that parallel many of the extensions for positional products:

- (`record-size` E_{rcd}): Returns the number of components in a record.

- (`record-insert` I E_{val} E_{rcd}): Let v_{rcd} be the record denoted by E_{rcd}. Then the `record-insert` expression returns a new record that has a binding of I to the value of E_{val} in addition to all the bindings of v_{rcd}. If v_{rcd} already has a binding for I, the new binding overrides it. With named products, `record-insert` corresponds to both `seq-insert` and `seq-update` for positional products.

- (`record-delete` I E_{rcd}): Let v_{rcd} be the record denoted by E_{rcd}. Then the `record-delete` expression returns a new record that has all the bindings of v_{rcd} except for any with the name I.

The `override` construct from Section 7.2 is a generalization of `record-insert` that combines two environments, while the `conceal` construct presented there is a generalization of `record-delete`. Other forms of record combination and name manipulation are also possible. For instance, it is possible to take the "intersection" or "difference" of two environments, or to specify the names that should be kept in a record rather than those that should be concealed.

It is even possible, but rare, to have a named index that can be calculated. In FL, such a construct might have the form `(select-sym` E_{sym} E_{rcd}`)`, where E_{sym} is an expression denoting a symbol value v_{sym} and `select-sym` selects from the record denoted by E_{rcd} the value associated with the label that is the underlying identifier of I_{sym}. It would be hard to imagine such a construct in a statically typed language. However, this idiom is often used in dynamically typed languages (such as LISP dialects) in the form of **association lists**, which are list of bindings between explicit symbols and values.

10.1.3 Non-strict Products

Our discussion so far has focused on **strict products**, in which the expressions specifying the product components are fully evaluated into values that are stored within the resulting product value. Another option is to have **non-strict products**, in which the component computations themselves are stored within the product value and are only run when their values are "demanded." Such products are the default in non-strict languages like HASKELL, but we will see that there are considerable benefits to integrating non-strict products into a call-by-value language, which is the focus of this section.

A simple approach to non-strict products is to adapt the call-by-name parameter passing mechanism to product formation. We will call the resulting data **call-by-name (CBN) products** in contrast to the **call-by-value (CBV) products** we have studied so far. An operational and denotational semantics for immutable positional CBN products in a call-by-value version of FL! is presented in Figure 10.3. We use the names `nproduct`/ `nproj` instead of `product`/`proj` to syntactically distinguish CBN products from CBV products. In the operational semantics, the delayed computation of product components is modeled by *not* having any progress rules for evaluating the component expressions of an `nproduct` expression. In the denotational semantics, a product value is represented as a sequence of computations rather than as a sequence of values. Intuitively, these computations are only "forced" into values upon projection from the CBN product by the occurrences of *with-value* that are sprinkled throughout the rest of the denotational semantics for CBV FL!.

As a simple example of how CBN products differ from CBV products, con-

Operational semantics for CBN products

$V \in \text{ValueExp} = \ldots \cup \{\texttt{(nproduct } E_1 \ \ldots \ E_n\texttt{)}\}$

$\langle \texttt{(nproj } N \texttt{ (nproduct } E_1 \ \ldots \ E_n\texttt{))}, S\rangle \Rightarrow \langle E_N, S\rangle,$ where $1 \leq N \leq \text{n}$ [*nproduct projection*]

Denotational semantics for CBN products

$$\begin{array}{rcl} \mathit{NProd} & = & \mathit{Computation}^* \\ v \in \mathit{Value} & = & \ldots + \mathit{NProd} \end{array}$$

$$\mathcal{E}[\![\texttt{(nproduct } E^*\texttt{)}]\!] = \lambda e\,.\,(\mathit{NProd} \mapsto \mathit{Value}\ (\mathcal{E}^*[\![E^*]\!]\ e))$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(nproj } N\ E_{prod}\texttt{)}]\!] = \\ \quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{prod}]\!]\ e) \\ \qquad (\lambda v\,.\ \textbf{matching}\ v \\ \qquad\quad \triangleright\,(\mathit{NProd} \mapsto \mathit{Value}\ c^*)\ [\!]\ \textbf{if}\ 1 \leq (\mathcal{N}\ N)\ \textbf{and}\ (\mathcal{N}\ N) \leq (\mathit{length}\ c^*) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (\mathit{nth}\ \ (\mathcal{N}\ N)\ \ c^*) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{error\text{-}comp} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\ \qquad\quad \triangleright\,\textbf{else}\ \mathit{error\text{-}comp} \\ \qquad\quad \textbf{endmatching}\,) \end{array}$$

Figure 10.3: Operational and denotational semantics for CBN positional products in call-by-name FL!.

sider the following expression:

```
(let ((c (cell 5)))
  (let ((p (nproduct (begin (:= c (+ (^ c) 1)) (^ c))
                     (begin (:= c (* (^ c) 2)) (^ c)))))
    (list (nproj 2 p) (nproj 1 p) (nproj 1 p) (nproj 2 p))))
```

The value of this expression is $[10, 11, 12, 24]$, indicating that the increments and doublings of the argument expressions are performed at every projection rather than when the CBN product is formed. If we had instead used CBV products, the above expression would yield $[6, 12, 12, 6]$, indicating that the side effects of the argument expressions are performed exactly once when the product is created.

In CBN products, the component computation is re-evaluated at every projection. Another option, inspired by the call-by-need parameter passing mechanism, is to evaluate the component computation at the very first projection and memoize the resulting value for later projections. We shall call this form of non-strict product a **lazy (CBL) product**. Using a lazy product in the above example would yield the list $[10, 11, 11, 10]$, which indicates that the side effects are performed on the first projections but not on subsequent projections.

The operational semantics of lazy products is presented in Figure 10.4. We use the names `lproduct` and `lproj` to distinguish lazy products from CBV and CBN products. A lazy product value is a sequence of locations in the store that may contain non-value expressions. At the first projection of a lazy product component, the [*lproj progress*] rule forces the evaluation of an unevaluated component expression to a value that is returned by the [*lproduct projection*] rule. Because the resulting value is "remembered" in the component location, subsequent projections of the component will return the value directly.

In the denotational semantics for lazy products (Figure 10.5), this memoizing behavior is modeled by extending *Storable* to be *Memo*,[3] which includes both values and computations. For a CBV language, we modify the *allocating* function to inject the initial value for a location in *Memo*, and introduce *allocatingComp* and *allocatingComps* for storing computations in freshly allocated locations. We modify *fetching* so that whenever the contents of a location is fetched, any computation stored at that location is evaluated to a value that is memoized at that location. A lazy product itself is modeled as a sequence of locations holding elements of *Memo*.

[3]This domain implies that computations could be stored at any location (such as cell locations), but in fact they can only be stored in lazy product locations. A practical implementation of lazy products would localize the overhead of memoization to lazy product component locations so that the efficiency of manipulating cell locations was not affected.

$$
\begin{array}{lcll}
S & \in & \text{Store} & = & \text{Assignment*} \\
Z & \in & \text{Assignment} & = & \text{Location} \times \text{Exp} \\
V & \in & \text{ValueExp} & = & \ldots \cup \{\texttt{(lproduct } L_1 \ \ldots \ L_n\texttt{)}\}
\end{array}
$$

$get : \text{Location} \to \text{Store} \rightharpoonup \text{Exp}$

$$
\begin{array}{l}
\langle \texttt{(lproduct } E_1 \ \ldots \ E_n\texttt{)}, S\rangle \\
\Rightarrow \langle \texttt{(lproduct } L_1 \ \ldots \ L_n\texttt{)}, [\langle L_1, E_1\rangle, \ldots, \langle L_n, E_n\rangle] \ @ \ S\rangle, \qquad [\textit{lproduct creation}] \\
\quad \text{where } L_1 \ \ldots \ L_n \text{ are fresh locations not appearing in } S.
\end{array}
$$

$$
\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\begin{array}{l}\langle \texttt{(lproj } N \texttt{ (lproduct } L_1 \ \ldots \ L_n\texttt{))}, S\rangle \\ \Rightarrow \langle \texttt{(lproj } N \texttt{ (lproduct } L_1 \ \ldots \ L_n\texttt{))}, (\langle L_N, E'\rangle \ . \ S')\rangle, \\ \quad \text{where } 1 \le N \le \text{n and } (get \ L_N \ S) = E\end{array}} \qquad [\textit{lproj progress}]
$$

$$
\begin{array}{l}
\langle \texttt{(lproj } N \texttt{ (lproduct } L_1 \ \ldots \ L_n\texttt{))}, S\rangle \Rightarrow \langle V, S\rangle, \qquad [\textit{lproduct projection}] \\
\quad \text{where } 1 \le N \le \text{n and } (get \ L_N \ S) = V
\end{array}
$$

Figure 10.4: Operational semantics for CBL products.

Non-strict products may be added to stateless languages like FL. We have chosen to focus on the stateful language FL! for two reasons:

1. It is easier to demonstrate the differences between the three forms of products in a language with state. In FL, only termination and errors could be used to distinguish strict and non-strict products, and CBN and CBL products are observationally indistinguishable.

2. Explaining the memoization of CBL products requires some form of state, so for presentational purposes it is easier to add these to a language like FL! that already has state.

The main benefit of non-strict products is that they enable the creation of conceptually infinite data structures that improve program modularity. For instance, we can introduce infinite lists, sometimes called **streams**, into a CBV language with the following sugar for `scons` (stream `cons`):

$$\mathcal{D}_{\text{exp}}[\![\texttt{(scons } E_1 \ E_2\texttt{)}]\!] = \texttt{(lproduct } E_1 \ E_2\texttt{)}$$

along with the following procedures:

```
(define (scar x) (lproj 1 x))
(define (scdr x) (lproj 2 x))
```

$$
\begin{array}{rcll}
mm & \in & Memo & = & Computation + Value \\
\sigma & \in & Storable & = & Memo \\
 & & LProd & = & Location^* \\
v & \in & Value & = & \ldots + LProd
\end{array}
$$

$$
\begin{array}{l}
allocating : Value \to (Location \to Computation) \to Computation \\
= \lambda vf\ .\ \lambda s\ .\ (f\ \ (\textit{fresh-loc}\ s)\ \ (\textit{assign}\ \ (\textit{fresh-loc}\ s)\ \ (Value \mapsto Memo\ v)\ \ s))
\end{array}
$$

$$
\begin{array}{l}
allocatingComp : Computation \to (Location \to Computation) \to Computation \\
= \lambda cf\ .\ \lambda s\ .\ (f\ \ (\textit{fresh-loc}\ s)\ \ (\textit{assign}\ \ (\textit{fresh-loc}\ s)\ \ (Computation \mapsto Memo\ c)\ \ s))
\end{array}
$$

$$
\begin{array}{l}
allocatingComps : Computation^* \to (Location^* \to Computation) \to Computation \\
= \lambda c^* f\ .\ (\textbf{matching}\ c^* \\
\qquad \triangleright [\,]_{Computation} \parallel (f\ \ [\,]_{Location}) \\
\qquad \triangleright (c\ .\ c^*) \parallel allocatingComp\ c\ \ (\lambda l\,.\,allocatingComps\ c^*\ \ (\lambda l^*\,.\,f\ \ (l\ .\ l^*))) \\
\qquad \textbf{endmatching}\ )
\end{array}
$$

$$
\begin{array}{l}
fetching : Location \to (Value \to Computation) \to Computation \\
= \lambda lf\ .\ \lambda s\ .\ \textbf{matching}\ (\textit{fetch}\ l\ s) \\
\qquad \triangleright (Storable \mapsto Assignment\ mm) \parallel \\
\qquad\ \textbf{matching}\ mm \\
\qquad\ \triangleright (Value \mapsto Memo\ v) \parallel f\ \ v\ \ s \\
\qquad\ \triangleright (Computation \mapsto Memo\ c) \parallel \\
\qquad\qquad \textit{with-value}\ \ c\ \ (\lambda vs'\ .\ \ f\ \ v\ \ (\textit{assign}\ \ l\ \ (Value \mapsto Memo\ v)\ \ s')) \\
\qquad\ \textbf{endmatching} \\
\qquad \triangleright \textbf{else}\ (\textit{error-comp}\ \ s) \\
\qquad \textbf{endmatching}
\end{array}
$$

$$
\mathcal{E}[\![\texttt{(lproduct}\ E^*\texttt{)}]\!] = \lambda e\ .\ (allocatingComps\ \ (\mathcal{E}^*[\![E^*]\!]\ \ e)\ \ (\lambda l^*\,.\,(LProd \mapsto Value\ \ l^*)))
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lproj}\ \ N\ \ E_{prod}\texttt{)}]\!] = \\
\quad \lambda e\ .\ (\textit{with-value}\ \ (\mathcal{E}[\![E_{prod}]\!]\ \ e) \\
\qquad\quad (\lambda v\ .\ (\textbf{matching}\ v \\
\qquad\qquad \triangleright (LProd \mapsto Value\ \ l^*) \parallel \textbf{if}\ 1 \le (\mathcal{N}\ \ N)\ \textbf{and}\ (\mathcal{N}\ \ N) \le (\textit{length}\ \ l^*) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (\textit{fetching}\ \ (\textit{nth}\ \ (\mathcal{N}\ \ N)\ \ l^*)\ \ \textit{val-to-comp}) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \textit{error-comp} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
\qquad\qquad \triangleright \textbf{else}\ \textit{error-comp} \\
\qquad\qquad \textbf{endmatching}\ )))
\end{array}
$$

Figure 10.5: Denotational semantics for CBL products in FL!.

The stream of all natural numbers can be created via `(ints-from 0)`, where the `ints-from` procedure is defined as

```
(define (ints-from n) (scons n (ints-from (+ n 1))))
```

The fact that the evaluation of the component expression `(ints-from (+ n 1))` is delayed until it is accessed prevents what would otherwise be an infinite recursion if `cons` were used instead of `scons`.

To view a prefix of a stream as a regular list, we will use the following procedure:

```
(define (prefix n str)
  (if (= n 0)
      (list)
      (cons (scar str) (prefix (- n 1) (scdr str)))))
```

For example:

`(prefix 5 (ints-from 3))` $\xrightarrow[FL]{}$ $[3,\ 4,\ 5,\ 6,\ 7]$

The stream mapping and filtering procedures in Figure 10.6 are handy for creating streams, such as the examples in Figure 10.7. Note how laziness enables the streams `nats`, `twos`, and `fibs` to all be defined directly in terms of themselves, without the need for an explicit recursive generating function like `ints-from`. The stream of prime numbers, `primes`, is calculated using the sieve of Eratosthenes method, which begins at 2 and keeps as primes only those following integers that are not multiples of previous primes. It is worth emphasizing that all of these examples could be implemented using regular lists (manipulated via `cons`, `car`, and `cdr`) in a call-by-name language or a call-by-need language; special lazy products are only necessary in a call-by-value language.

As an example of the modularity benefits of the conceptually infinite data structures enabled by non-strict products, consider the `first-bigger-than` procedure, which returns the first value in a numeric stream that is strictly bigger than a given threshhold `n`.

```
(define (first-bigger-than n str)
  (if (> (scar str) n)
      (scar str)
      (first-bigger-than n (scdr str))))

(first-bigger-than 1000 nats) ⟶FL! 1001
(first-bigger-than 1000 evens) ⟶FL! 1002
(first-bigger-than 1000 twos) ⟶FL! 1024
(first-bigger-than 1000 fibs) ⟶FL! 1597
(first-bigger-than 1000 primes) ⟶FL! 1009
```

```
; Applies a unary function F elementwise to stream STR.
(define (smap f str)
  (scons (f (scar str))
         (smap f (scdr str))))

; Applies a binary function G elementwise to corresponding
; elements of STR1 and STR2.
(define (smap2 g str1 str2)
  (scons (g (scar str) (scar str2))
         (smap2 g (scdr str1) (scdr str2))))

; Returns a stream with only those elements of STR
; satisfying the predicate PRED.
(define (sfilter pred str)
  (if (pred (scar str))
      (scons (scar str) (sfilter pred (scdr str)))
      (sfilter pred (scdr str))
```

Figure 10.6: Mapping and filtering procedures for streams.

Infinite lists allow a list processing termination condition to be specified in the consumer of a list rather than in the producer of a list. With strict lists, all lists must be finite, so the termination condition must be specified when the list is produced. To get the behavior of `first-bigger-than` with strict lists, it would be necessary to intertwine the details of generating the next element with checking it against the threshhold – a strategy that would compromise the modularity of having a separate `first-bigger-than` procedure.

▷ **Exercise 10.3** The **Hamming numbers** are all positive integers whose non-trivial factors are 2, 3, and 5 exclusively. Define a stream of the Hamming numbers. What is the first Hamming number strictly larger than 1000? ◁

▷ **Exercise 10.4** Many SCHEME implementations support a form of stream created out of pairs where the second component is lazy but the first is not:

$\mathcal{D}_{\text{exp}}[\![$`(cons-stream` E_1 E_2`)`$]\!] =$ `(cons` E_1 `(delay` E_2`))`

```
(define (head str) (car str))
(define (tail str) (force (cdr str)))
```

Here, `delay` and `force` implement a memoized delayed value, like `lazy` and `touch` did in Exercise 7.1.

a. Show that it is possible to define all lazy lists illustrated in this section as SCHEME streams.

```
; All natural numbers
(define nats (scons 0 (smap (+ 1) nats))

(prefix 5 nats) ⟶FL [0, 1, 2, 3, 4]

; All even natural numbers
(define evens (sfilter (lambda (x) (= (rem x 2) 0)) nats))

(prefix 5 evens) ⟶FL [0, 2, 4, 6, 8]

; All powers of two
(define twos (scons 1 (smap (* 2) twos)))

(prefix 5 twos) ⟶FL [1, 2, 4, 8, 16]

; All Fibonacci numbers
(define fibs (scons 0 (scons 1 (smap2 + fibs (scdr fibs)))))

(prefix 10 fibs) ⟶FL [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

; All prime numbers
(define primes
  (letrec
    ((sieve
       (lambda (str)
         (scons (scar str)
                (sieve (sfilter (lambda (x)
                                  (not (= (rem x (scar str)) 0)))
                                (scdr str)))))))
    (sieve (ints-from 2))))

(prefix 10 primes) ⟶FL [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Figure 10.7: Some sample streams of numbers.

b. Design a stream in which laziness in the first component is essential – that is, which can be defined via `scons/scar/scdr` but not via `cons-stream/head/tail`. ◁

▷ **Exercise 10.5**

a. Use `lproduct/lproj` to define constructors and selectors for infinite binary trees in which each node holds a value in addition to its left and right subtrees.

b. Use your constructs to define an infinite binary tree whose left-to-right inorder traversal yields the positive integers in order of magnitude.

c. Define an `inorder-stream` procedure that returns a stream of the elements of an infinite binary tree as they would be encountered in a left-to-right inorder traversal. ◁

10.1.4 Mutable Products

Thus far we have discussed only immutable products — those whose components do not change over time. But in popular imperative languages, the vast majority of built-in data structures are mutable products. Here we explore some design dimensions of mutable products and some examples of mutable products in real languages.

All of the dimensions we explored above for immutable products are relevant to mutable products. For example, mutable product components are either named or positional. Examples of mutable products with named components include C's structures and PASCAL's records. A canonical example of a fixed-size mutable product with positional components is SCHEME's pairs, whose two components may be altered via `set-car!` and `set-cdr!`. Mutable sequences are typically called arrays (as in C/C++, JAVA, PASCAL, FORTRAN, and CLU) or vectors (as in SCHEME and JAVA). All of these support the ability to update the component at any index, often via a special subscripting notation, such as `a[i] = 2*a[i];` in C/C++/JAVA. Only some of these — CLU's arrays and JAVA's vectors (but not JAVA's arrays) — support the ability to expand or contract the size of the mutable sequence by inserting or removing elements. All of these examples of mutable products have 0-based indexing except for FORTRAN (which has 1-based arrays), CLU (whose arrays can have any lower bound but are 1-based by default), and PASCAL (whose arrays support arbitrary enumerations as indices). In all of these examples, all components are required to be of the same type, except for SCHEME's vectors (where any slot may contain any value) and JAVA's vectors (where any slot may contain any object).

Although the mutable products mentioned above seem similar on the surface, their semantics differ in fundamental ways. Below we explore some of the dimensions along which mutable products can differ. For simplicity, we consider only mutable fixed-length positional products of heterogeneous values, which we shall call **mutable tuples**. It is easy to generalize these to other kinds of mutable products. We will study the addition of mutable tuples to FL!. We assume a CBV parameter passing mechanism unless otherwise stated. Here are the constructs we will consider:

E ::= ...
| `(mprod` E^*`)` [Mutable Tuple Creation]
| `(mget` N_{index} E_{mt}`)` [Mutable Tuple Projection]
| `(mset!` N_{index} E_{mt} E_{new}`)` [Mutable Tuple Assignment]

Informally, these constructs have the following semantics:

- `(mprod` E_1 ... E_n`)` creates a new mutable tuple with n mutable slots indexed from 1 to n where slot i is initially filled with the value of E_i.
- In `(mget` N_{index} E_{mt}`)`, assume that E_{mt} evaluates to a mutable tuple mt with n slots, where $1 \leq N_{index} \leq n$. Then `mget` returns the value in the ith slot of mt. Otherwise, `mget` signals an error.
- In `(mset!` N_{index} E_{mt} E_{new}`)`, assume that E_{mt} evaluates to a mutable tuple mt with n slots, where $1 \leq N \leq n$, and E_{new} evaluates to v. Then `mset!` changes the value in the ith slot of mt to be v. Otherwise, `mset!` signals an error.

For example, here is an expression involving a mutable tuple:

```
(let ((m (mprod 3 4)))
  (begin
    (mset! 1 m (+ (mget 1 m) (mget 2 m))) ; 1st slot is now 7.
    (mset! 2 m (+ (mget 1 m) (mget 2 m))) ; 2nd slot is now 11.
    (* (mget 1 m) (mget 2 m)))) ⟶FL! 77
```

A very simple way to include mutable products in a language is to have a single kind of mutable entity — such as a mutable cell — and allow this entity to be a component of otherwise immutable structures. This is the approach taken in ML, where immutable tuples, vectors, and user-defined datatypes may have mutable cells as components. We can model this approach in FL! via the following desugarings for `mprod`, `mget`, and `mset!`:

$\mathcal{D}[\![$`(mprod` E_l ... E_n`)`$]\!]$ = `(product (cell` $\mathcal{D}[\![E_l]\!]$`) ... (cell` $\mathcal{D}[\![E_n]\!]$`))`
$\mathcal{D}[\![$`(mget` N E_{mp}`)`$]\!]$ = `(cell-ref (proj` N $\mathcal{D}[\![E_{mp}]\!]$`))`
$\mathcal{D}[\![$`(mset!` N E_{mp} E_{new}`)`$]\!]$ = `(cell-set! (proj` N $\mathcal{D}[\![E_{mp}]\!]$`)` $\mathcal{D}[\![E_{new}]\!]$`)`

In typical imperative languages, a more common design is to directly support various kinds of mutable products, perhaps along with some immutable ones. The CLU language, for example, supports a variety of different built-in datatypes, each of which comes in both mutable and immutable flavors.

In most imperative languages, mutable products would be modeled as a sequence of locations, as shown in the denotational semantics presented in Figure 10.8, which is a straightforward generalization to the semantics of mutable cells. Mutable tuple values are represented as sequences of locations. This is similar to the representation of lazy products, except that in `mprod`, the computed values of the subexpressions (rather than the computatations for these subexpressions) are stored in the locations.

A key issue in the semantics of mutable products is how they are passed as parameters. When mutable products are added as values to the CBV version of FL! we have studied, we shall say that the they are passed via a **call-by-value-sharing (CBVS)** mechanism because both the caller and the callee share access to the same locations in the mutable product. For example, in the following expression, references to `t` and `m` in the body of the procedure `f` refer to the same mutable product, so that changes to the components of one are visible in the other:

```
(let ((t (mprod 5 6)))
  (let ((f (lambda (m)
             (begin
               (mset! 1 t (* 10 (mget 1 t)))
               (mset! 2 m (* 100 (mget 2 m)))
               (mget 1 m)))))
    (+ (f t) (mget 2 t))))
```
$\xrightarrow[CBVS\ \ FL!]{}$ *650*

This is the behavior expected for mutable products in languages such as JAVA, SCHEME and CLU. Conceptually, when a mutable product is assigned to a variable, passed as a parameter, returned as a result, or stored in a data structure, no new product locations are created; the existing product locations are simply shared in all parts of the program to which the given product value has "flowed."

An alternative strategy for passing mutable products in a CBV language is to create a new product with new locations whenever a product is passed from one part of a program to another. This approach, which we shall term **call-by-value-copy (CBVC)** is explained by the denotational semantics for a variant of FL! with mutable products (Figure 10.9). Whenever a value is passed, a copy of the value is made. Primitive values, procedures, and locations (i.e., cells) are not copied, but a mutable tuple with n slots is copied by allocating n new locations and filling these with copies of the contents of the existing locations.

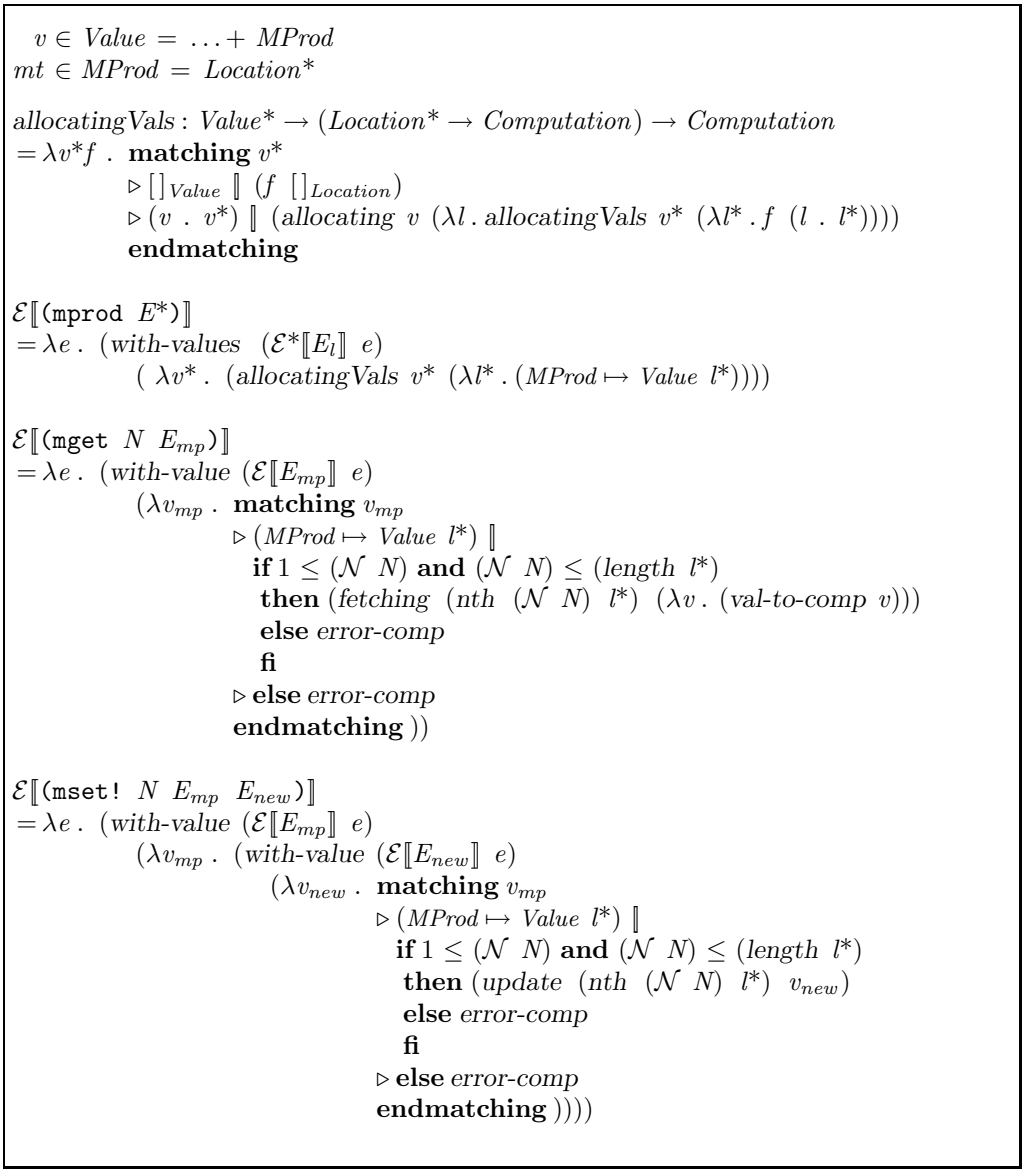


Figure 10.8: Denotational semantics of mutable tuples with CBVS parameter passing.

In a CBVC interpretation of the example expression considered for CBVS, the names `t` and `m` refer to two distinct mutable tuples, so that changes to one pair are not visible in the other:

```
(let ((t (mprod 5 6)))
  (let ((f (lambda (m)
             (begin
               (mset! 1 t (* 10 (mget 1 t)))
               (mset! 2 m (* 100 (mget 2 m)))
               (mget 1 m)))))
    (+ (f t) (mget 2 t))))  ⟶ 11
```

$v \in \mathit{Value} = \mathit{Unit} + \mathit{Bool} + \mathit{Int} + \mathit{Sym} + \mathit{Procedure} + \mathit{Location} + \mathit{MProd}$

$$
\begin{array}{l}
\mathit{allocatingCopies} : \mathit{Location}^* \to (\mathit{Value} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}) \\
\qquad\qquad \to (\mathit{Location}^* \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda l^* g f \;.\; \textbf{matching}\ l^* \\
\qquad \triangleright [\,]_{\mathit{Location}} \;[\![\; f \;\; [\,]_{\mathit{Location}} \\
\qquad \triangleright (l_{old} \;.\; l_{old}{}^*) \;[\![\; \mathit{fetching}\ l_{old} \\
\qquad\qquad (\lambda v \;.\; g \;\; v \;\; (\lambda v' \;.\; \mathit{allocating}\ v' \\
\qquad\qquad\qquad (\lambda l_{new} \;.\; \mathit{allocatingCopies}\ l_{old}{}^* \\
\qquad\qquad\qquad\qquad (\lambda l_{new}{}^* \;.\; f \;\; (l_{new} \;.\; l_{new}{}^*)))))
\end{array}
$$

$$
\begin{array}{l}
\mathit{deepCopying} : \mathit{Value} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda v f \;.\; \textbf{matching}\ v \\
\qquad \triangleright (\mathit{MProd} \mapsto \mathit{Value}\ l_{old}{}^*) \\
\qquad\quad [\![\; (\mathit{allocatingCopies}\ l_{old}{}^*\ \mathit{deepCopying}\ (\lambda l_{new}{}^* \,.\, f \;\; (\mathit{MProd} \mapsto \mathit{Value}\ l_{new}{}^*))) \\
\qquad \triangleright \textbf{else}\ f \;\; v \\
\qquad \textbf{endmatching}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda e \;.\; \mathit{with\text{-}procedure\text{-}comp}\ (\mathcal{E}[\![E_1]\!]\ e) \\
\qquad\qquad (\lambda p \;.\; \mathit{with\text{-}value}\ (\mathcal{E}[\![E_2]\!]\ e) \\
\qquad\qquad\qquad (\lambda v \;.\; (\mathit{deepCopying}\ v\ \mathit{val\text{-}to\text{-}comp})))
\end{array}
$$

Figure 10.9: Call-by-value-copy (CBVC) semantics for passing mutable tuples.

The CBVC strategy for passing mutable products is used for passing arrays and records by value in PASCAL and for passing structures by value in C. On the other hand, arrays in C are passed via CBVS. Passing arrays in C via CBVC can be achieved by wrapping an array in a one-component struct! The inconsistency between the mechanisms for passing named vs. positional products in C is perplexing from the viewpoint of semantics but was apparently motivated by pragmatic issues.

The kind of data copying performed in Figure 10.9 is known as a **deep copy** because the copying process is recursively applied at all levels of the data. An alternative strategy, known as a **shallow copy**, is to copy only the first level of a data structure and share the contents of the other levels. Although it would be possible to use shallow copying in the call-by-copy strategy, we do not know of a real programming language that uses this strategy.

In languages supporting the call-by-reference (CBR) mechanism presented in Section 8.3.3.4, mutable products introduce new ways to alias locations between the caller and callee. When an `mget` construct is used in a parameter position, its L-value (the location of the product slot, as determined by $\mathcal{LV}$ in Figure 10.10) is passed rather than its R-value (the contents of the L-value). In the following CBR example, the L-values of `(mget 2 u)` and `r` denote the same location:

```
(let ((u (mprod 7 8)))
  (let ((g (lambda (p r)
             (begin
               (set! r (+ 20 r))
               (mset! 2 p (+ 100 (mget 2 p)))))))
    (begin (g u (mget 2 u))
           (mget 2 u))))
```

$\xrightarrow[CBR\ \ FLAVAR!]{}$ *128*

In contrast, under a CBV interpretation, changes to `r` would not affect `u` and `p`. The above expression would evaluate to *108* under CBVS and *8* under CBVC.

$$
\begin{array}{l}
\mathcal{LV}[\![\texttt{(mget}\ N\ E_{mp}\texttt{)}]\!] \\
= \lambda e\ .\ (\textit{with-value}\ (\mathcal{E}[\![E_{mp}]\!]\ e) \\
\qquad (\lambda v_{mp}\ .\ \textbf{matching}\ v_{mp} \\
\qquad\quad \triangleright\ (\textit{MProd} \mapsto \textit{Value}\ l^*)\ [\![\\
\qquad\quad\ \ \textbf{if}\ 1 \leq (\mathcal{N}\ N)\ \textbf{and}\ (\mathcal{N}\ N) \leq (\textit{length}\ l^*) \\
\qquad\quad\ \ \textbf{then}\ (\textit{val-to-comp}\ (\textit{Location} \mapsto \textit{Value}\ (\mathcal{N}\ N))l^*) \\
\qquad\quad\ \ \textbf{else}\ \textit{error-comp} \\
\qquad\quad\ \ \textbf{fi} \\
\qquad\quad \triangleright\ \textbf{else}\ \textit{error-comp} \\
\qquad\quad \textbf{endmatching}\ ))
\end{array}
$$

Figure 10.10: Extension to the CBR FLAVAR! semantics to handle mutable tuples.

▷ **Exercise 10.6** Write a single expression that returns the symbol `sharing` under CBVS, `deep` under CBVC with deep copying, and `shallow` under CBVC with shallow copying. Your expression should only use symbols, mutable tuples, and procedures. ◁

▷ **Exercise 10.7**

a. Modify the CBVC denotational semantics in Figure 10.9 to use shallow rather than deep copying.

b. Write three different versions of an operational semantics for FL! with mutable tuples that differ in their parameter passing mechanism: (1) CBVS (2) CBVC with deep copying (3) CBVC with shallow copying. ◁

10.2 Sums

Sums are entities that can be one of several different kinds of values. They are data structures that correspond to the sum domains that we have been using in our mathematical metalanguage (see Section A.3.3) to represent mathematical values that can come from several different component domains. Intuitively, a sum value augments an underlying component value with a **tag** that indicates which kind of value it is. Whenever a sum value is processed, this tag is dynamically examined to determine how to handle the underlying value. Sums are used in situations where programmers use the terms "either" or "one of" to informally describe a data structure. For example:

- A linked list is either a list node (with head and tail components) or the empty list.
- A graphics system might support shapes that are either circles, rectangles, or triangles.
- In a banking system, transactions might be one of deposit, withdrawal, transfer, or balance query.

Sums are known by such names as *tagged sums*, *unions*, *tagged unions*, *discriminated unions*, *oneofs*, and *variants*.

Just as sum domains are duals of product domains, sum data structures are dual to product data structures: for any given product structure, there is a dual sum data structure. Sums therefore vary along the same dimensions as products: positional vs. named, immutable vs. mutable, and dynamically typed vs. statically typed. Our discussion will focus on the first of these dimensions: **positional sums**, in which the different cases are distinguished only by their position in the sum specification (i.e., their tags are natural numbers) vs. **named sums**, in which the different cases are distinguished by specified names.

10.2.1 Positional Sums

Positional product data structures use integer indices to distinguish product components. Similarly, positional sums use integer tags to distinguish summands. To add positional sums to FL, we extend the syntax of expressions as follows:

E ::= ...
| `(inj` N E`)` [Sum Introduction]
| `(sumcase` E_{disc} I_{val} E_{body}`*)` [Sum Elimination]

`(inj` N E`)` creates a sum value whose tag is the integer N, where N is a manifest constant and not a computed value. Sum values are taken apart with `(sumcase` E_{disc} I_{val} E_{body}`*)`, which evaluates the **discriminant** E_{disc} to what should be a sum value, examines the numeric tag N of this sum value, and evaluates the Nth body expression with I_{val} bound to the untagged sum component.

Figure 10.11 shows a simple bank transaction system implemented with positional sums. An account is a pair of a savings balance and a checking balance. There are four kinds of transactions distinguished by an integer tag:

1. Deposit of an integer amount to savings.
2. Withdrawal of an integer amount from checking.
3. Transfer of an integer amount from savings to checking.
4. Transfer of an integer amount from checking to savings.

Given a transaction and account, the `process` procedure returns an updated account that reflects the actions of the translation.

The operational semantics for call-by-value sums is presented in Figure 10.12. Stuck states arise when any of the subexpressions get stuck, when the discriminant does not evaluate to a sum value, or when the integer tag does not correspond to an appropriate body (e.g., the integer is negative, zero, or larger than the number of bodies supplied). Call-by-name sums are similar (Figure 10.12), except that no attempt is made to evaluate the expression being injected.

The denotational semantics for call-by-value sums is presented in Figure 10.14. It might seem odd that the domain *Sum* of sum values is modeled via a product that pairs an integer tag and the injected value. But such a product is isomorphic to an infinite sum of injected values, so it does indeed represent a sum. The clause for `sumcase` calculates the denotations of all body expressions and chooses one based on the integer tag of the sum value. However, as with the denotational semantics of `if` expressions, only the chosen body is "evaluated." The denotational semantics of call-by-name sums is left as an exercise.

Positional sums are awkward to use in practice for several reasons:

```
(define (make-account checking savings) (pair checking savings))
(define (checking account) (left account))
(define (savings account) (right account))

(define (process transaction account)
  (sumcase transaction amount
    ; Deposit to savings
    (make-account (checking account)
                  (+ (savings account) amount))
    ; Withdrawal from checking
    (if (<= amount (checking account))
        (make-account (- (checking account) amount)
                      (savings account))
        (error 'insufficient-checking))
    ; Transfer from savings to checking
    (if (<= amount (savings account))
        (make-account (+ (checking account) amount)
                      (- (savings account) amount))
        (error 'insufficient-savings))
    ; Transfer from checking to savings
    (if (<= amount (checking account))
        (make-account (- (checking account) amount)
                      (+ (savings account) amount))
        (error 'insufficient-checking))
    ))

(process (inj 1 10) (make-account 25 40)) ⟶_FL ⟨25, 50⟩
(process (inj 2 10) (make-account 25 40)) ⟶_FL ⟨15, 40⟩
(process (inj 3 10) (make-account 25 40)) ⟶_FL ⟨35, 30⟩
(process (inj 4 10) (make-account 25 40)) ⟶_FL ⟨15, 50⟩
```

Figure 10.11: Bank transactions with positional sums.

$$V \in \text{ValueExp} = \ldots \cup \{\texttt{(inj } N\ V\texttt{)}\}$$

$$\frac{E \Rightarrow E'}{\texttt{(inj } N\ E\texttt{)} \Rightarrow \texttt{(inj } N\ E'\texttt{)}} \quad [\textit{inj-progress}]$$

$$\frac{E \Rightarrow E'}{\texttt{(sumcase } E\ I\ E_1\ \ldots\texttt{)} \Rightarrow \texttt{(sumcase } E'\ I\ E_1\ \ldots\texttt{)}} \quad [\textit{sumcase-progress}]$$

$$\texttt{(sumcase (inj } N\ V\texttt{)}\ I\ E_1\ \ldots\ E_m\texttt{)} \Rightarrow [V/I]E_N, \text{ where } 1 \leq N \leq m \quad [\textit{sumcase}]$$

Figure 10.12: CBV operational semantics for positional sums

$$V \in \text{ValueExp} = \ldots \cup \{\texttt{(inj } N\ E\texttt{)}\}$$

$$\frac{E \Rightarrow E'}{\texttt{(sumcase } E\ I\ E_1\ \ldots\texttt{)} \Rightarrow \texttt{(sumcase } E'\ I\ E_1\ \ldots\texttt{)}} \quad [\textit{sumcase-progress}]$$

$$\texttt{(sumcase (inj } N\ E_{val}\texttt{)}\ I\ E_1\ \ldots\ E_m\texttt{)} \Rightarrow [E_{val}/I]E_N, \text{ where } 1 \leq N \leq m \quad [\textit{sumcase}]$$

Figure 10.13: CBN operational semantics for positional sums

$$\begin{array}{rcl} su & \in & Sum = Int \times Value \\ v & \in & Value = \ldots + Sum \end{array}$$

$$\mathcal{E}[\![\texttt{(inj } N\ E\texttt{)}]\!] = \lambda e.\ \textit{with-value}\ (\mathcal{E}[\![E]\!]\ e)\ (\lambda v.\ (\textit{val-to-comp}\ (Sum \mapsto Value\ \langle(\mathcal{N}\ N),\ v\rangle)))$$

$$\begin{array}{l} \mathcal{E}[\![\texttt{(sumcase } E_{disc}\ I_{val}\ E^*\texttt{)}]\!] = \\ \quad \lambda e.\ \textit{with-value}\ (\mathcal{E}[\![E_{disc}]\!]\ e) \\ \qquad (\lambda v_{disc}.\ \textbf{matching}\ v_{disc} \\ \qquad\quad \triangleright\ (Sum \mapsto Value\ \langle i,\ v\rangle)\ [\!]\ (nth_{Computation}\ i\ (\mathcal{E}^*[\![E^*]\!]\ [I_{val} : v]e)) \\ \qquad\quad \triangleright\ \textbf{else}\ \textit{error-comp} \\ \qquad\quad \textbf{endmatching}\ ) \end{array}$$

Figure 10.14: CBV denotational semantics for sums

1. The programmer must remember the arbitrary association between each integer tag and its intended meaning;

2. In a `sumcase` expression, the body expressions must be carefully ordered to have the correct (implicit) index;

3. Since all sum values use integer tags, a sum value intended for one purpose may accidentally be used for another without any error being reported.

▷ **Exercise 10.8** In call-by-name positional sums, the expression `(inj` N E`)` does not tag the *value* denoted by E but rather tags the *computation* denoted by E. Modify the denotational semantics for positional sums in Figure 10.14 to be call-by-name rather than call-by-value. ◁

▷ **Exercise 10.9** The simplest kind of positional product is a pair, which glues together two component values. Dually, the simplest kind of positional sum chooses between two component values. Such a sum value is called an an **either**. It has two two possible tags: *left* or *right*.

Here we consider an extension to FL that supports eithers rather than general positional sums. Suppose we extend the syntax of FL as follows:

E ::= ...
| `(inleft` E`)` [Either Left Injection]
| `(inright` E`)` [Either Right Injection]
| `(ecase` E_{disc} I_{val} E_{left} E_{right}`)` [Either Case Analysis]

`(inleft` E`)` creates an either whose tag is *left* and whose value is the value of E.

`(inright` E`)` creates an either with whose tag is *right* and whose value is the value of E.

`(ecase` E_{disc} I_{val} E_{left} E_{right}`)` examines the discriminant value represented by E_{disc}, binds the untagged value to the identifier I_{val}, and then evaluates E_{left}, if the tag is *left*, or E_{right}, if the tag is *right*. It is an error if E_{disc} is not an either.

For example, we can use eithers in an extended version of FL to encode whether a geometric shape is a square (in which case the value of the either is the length of a side) or a circle (in which case the value of the either is the radius). We can then write a procedure for computing the area of a shape:

```
(define (square side) (inleft side))
(define (circle radius) (inright radius))
(define pi 3.14159)
(define (area shape)
  (ecase shape v
    (f* v v)  ; square case (f* multiplies floating point numbers)
    (f* pi (f* v v)) ; circle case
    ))

(area (square 10.0)) ⟶FL 100.0
(area (circle 10.0)) ⟶FL 314.159
```

a. Write an operational semantics for CBV eithers. What causes stuck states in your semantics?

b. Write a denotational semantics for CBV eithers. You may find it convenient to have a new domain for eithers as well as new *Left* and *Right* domains.

c. Write an operational semantics for CBN eithers. What causes stuck states in your semantics?

d. Write a denotational semantics for CBN eithers. You may find it convenient to have a new domain for eithers as well as new *Left* and *Right* domains. ◁

10.2.2 Named Sums

Named sums address the problems of positional sums by using programmer-supplied names to distinguish the various cases in a sum value. Named sums involve two new constructs:

E ::= ...
| `(one` I_{tag} E`)` [Oneof Intro]
| `(tagcase` E_{disc} I_{val} `(`I_{tag} E_{body}`)*` [`(else` E_{else}`)`]`)` [Oneof Elim]

A named sum value, which we shall call a **oneof**, is created by the evaluation of the expression `(one` I_{tag} E`)`, which conceptually pairs the tag I_{tag} with the component value given by E. Oneofs are decomposed via the expression `(tagcase` E_{disc} I_{val} `(`I_{tag} E_{body}`)*)`, which dispatches to a clause based on the tag of the oneof value of the discriminant expression E_{disc}. The value of the `tagcase` is the result of evaluating the body of the clause with the matching tag in a scope where I_{val} is bound to the untagged oneof component. A `tagcase` expression may have an optional `else` clause whose body E_{else} is evaluated and returned when no clause tag matches the discriminant tag. It is an error if E_{disc} does not evaluate to a oneof or if there is no clause in an `else`-less `tagcase` whose tag matches the discriminant tag.

Figure 10.15 shows how the bank transaction example can be expressed with named sums. Using symbolic tags instead of integers makes such programs easier to read and write; the tags serve as comments and allow the `tagcase` clauses to be written in any order. Although it is still possible for the same symbolic tag to be used for conceptually different oneofs, the likelihood that a oneof will be used in a incorrect context without generating a dynamic error is greatly reduced.

```
(define (process transaction account)
  (tagcase transaction amount
    (savings-deposit
      (make-account (checking account)
                    (+ (savings account) amount)))
    (checking-withdrawal
      (if (<= amount (checking account))
          (make-account (- (checking account) amount)
                        (savings account))
         (error 'insufficient-checking)))
    (savings->checking
      (if (<= amount (savings account))
          (make-account (+ (checking account) amount)
                        (- (savings account) amount))
          (error 'insufficient-savings)))
    (checking->savings
      (if (<= amount (checking account))
          (make-account (- (checking account) amount)
                        (+ (savings account) amount))
          (error 'insufficient-checking)))
    ))

(process (one savings-deposit 10) (make-account 25 40)) ⟶_FL ⟨25, 50⟩
(process (one checking-withdrawal 10) (make-account 25 40)) ⟶_FL ⟨15, 40⟩
(process (one savings->checking 10) (make-account 25 40)) ⟶_FL ⟨35, 30⟩
(process (one checking->savings 10) (make-account 25 40)) ⟶_FL ⟨15, 50⟩
```

Figure 10.15: Bank transactions with named sums.

Oneofs have semantics similar to that for positional sums, except that identifiers are used as tags rather than integers. Figure 10.16 gives the call-by-value operational semantics for oneofs. As before, stuck states arise when a value that is not a oneof appears as the discriminant of a `tagcase` or when a `tagcase` does not specify a clause appropriate for the dynamic tag of the oneof value.

The denotational semantics for call-by-value oneofs (Figures 10.17–10.18) shows their duality with records quite clearly. Records use an environment to

$$V \in \text{ValueExp} = \ldots \cup \{\texttt{(one } I\ V\texttt{)}\}$$

$$\frac{E \Rightarrow E'}{\texttt{(one } I\ E\texttt{)} \Rightarrow \texttt{(one } I\ E'\texttt{)}} \quad [\textit{one-progress}]$$

$$\frac{E \Rightarrow E'}{\texttt{(tagcase } E\ I\ \ldots\texttt{)} \Rightarrow \texttt{(tagcase } E'\ I\ \ldots\texttt{)}} \quad [\textit{tagcase-progress}]$$

$$\texttt{(tagcase (one } I_i\ V\texttt{) } I\ \texttt{(}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{))} \Rightarrow [V/I]E_i \quad [\textit{tagcase}]$$

$$\texttt{(tagcase (one } I_{tag}\ V\texttt{) } I\ \texttt{(}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{) (else } E_{else}\texttt{))} \Rightarrow [V/I]E_{else}, \quad [\textit{tagcase-else}]$$
$$\text{where } I_{tag} \notin \{I_1, \ldots I_n\}$$

Figure 10.16: CBV operational semantics for named sums

glue together named values, one of which is later chosen at each `select` site. Dually, `one` creates a sum that is later processed in the context of a `tagcase` that uses an environment to glue together named clause bodies. In a continuation-based semantics, the environment associated with the `tagcase` would map names to continuations, suggesting a duality between values and continuations.

▷ **Exercise 10.10**

a. Modify the operational semantics for named sums in Figure 10.16 to be call-by-name rather than call-by-value.

b. Modify the denotational semantics for named sums in Figure 10.18 to be call-by-name rather than call-by-value.

c. Write a denotational semantics for call-by-name and call-by-value named sums in a continuation-based semantics. ◁

10.3 Sum-of-Products

In practice, sum and product data are often used together in idiomatic ways. Many common data structures can be viewed as a tree constructed from different kinds of nodes, each of which has multiple components. Here are some examples:

- A shape in a simple geometry system is either:

$$
\begin{array}{l}
t \in \textit{Tag-environment} = \text{Identifier} \to \textit{Denotable} \to \textit{Computation} \\
\textit{empty-tenv} : \textit{Tag} - \textit{environment} = \lambda I \delta \,.\; \textit{error-comp} \\
\\
\textit{extend-tenv} : \textit{Environment} \to \text{Identifier} \to \text{Identifier} \to \text{Exp} \to \textit{Tag} - \textit{environment} \\
\qquad \to \textit{Tag} - \textit{environment} \\
= \lambda e\ I_{val}\ I\ E\ t\,.\,\lambda I'\ \delta\,.\,\textbf{if}\ (\textit{same-identifier?}\ I'\ I)\ \textbf{then}\ (\mathcal{E}[\![E]\!]\ [I_{val} : \delta]e)\ \textbf{else}\ (t\ I') \\
\\
\textit{extend-tenv*} : \textit{Environment} \to \text{Identifier} \to \text{Identifier*} \to \text{Exp*} \to \textit{Tag} - \textit{environment} \\
\qquad \to \textit{Tag} - \textit{environment} \\
= \lambda e\ I_{val}\ I^*\ E^*\ t\,.\ \textbf{matching}\ \langle I^*, E^* \rangle \\
\qquad \triangleright \langle [\,]_{\text{Identifier}}, [\,]_{\text{Exp}} \rangle\ [\![\ t \\
\qquad \triangleright \langle I\ .\ {I_{rest}}^*, E\ .\ {E_{rest}}^* \rangle \\
\qquad\quad [\![\ (\textit{extend-tenv*}\ e\ \ I_{val}\ \ I_{rest}\ \ {E_{rest}}^* \\
\qquad\qquad\qquad\qquad (\textit{extend-tenv}\ e\ I_{val}\ I_{rest}\ E_{rest}\ t)) \\
\qquad \triangleright\ \textbf{else}\ \textit{empty-tenv} \\
\qquad \textbf{endmatching}
\end{array}
$$

Figure 10.17: Auxiliary domains and functions for denotational semantics of named sums (oneofs)

  - a circle with a radius;
  - a rectangle with a width and a height;
  - a triangle with three side lengths.

- A list of integers is either:
  - an empty list;
  - a list node with an integer head and an integer list tail.
- An ELM expression is either:
  - an integer literal;
  - an argument expression with an index;
  - an arithmetic operation with an operator symbol, a left operand expression, and a right operand expression.

In each of the above examples, the variety of possible nodes for a data structure can be modeled as a sum, and each individual kind of node can be modeled as a product. For this reason, such data structures are known as **sum-of-product** structures.

As a simple example, consider the following list of geometric shapes:

$$
\begin{array}{rcll}
su & \in & Sum & = \text{Identifier} \times Value \\
v & \in & Value & = \ldots + Sum
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(one } I\ E\texttt{)}]\!] = \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E]\!]\ e) \\
\qquad\qquad (\lambda v\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Sum} \mapsto \mathit{Value}\ \langle I,\ v\rangle))))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(tagcase } E_{disc}\ I_{val}\ \texttt{(}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}]\!] = \\
\quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{disc}]\!]\ e) \\
\qquad (\lambda v_{disc}\,.\ \textbf{matching}\ v_{disc} \\
\qquad\quad \triangleright\, (\mathit{Sum} \mapsto \mathit{Value}\ \langle I_{tag},\ v\rangle) \\
\qquad\qquad [\![\ ((\mathit{extend\text{-}tenv}^*\ e\ I_{val}\ [I_1 \ldots I_n]\ [E_1 \ldots E_n]\ \mathit{empty\text{-}tenv})\ I_{tag}\ v) \\
\qquad\quad \triangleright\, \textbf{else}\ \mathit{error\text{-}comp} \\
\qquad\quad \textbf{endmatching}\ )
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(tagcase } E_{disc}\ I_{val}\ \texttt{(}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{) (else } E_{else}\texttt{))}]\!] = \\
\quad \lambda e\,.\ \mathit{with\text{-}value}\ (\mathcal{E}[\![E_{disc}]\!]\ e) \\
\qquad \textbf{let}\ \mathit{elsetenv}\ \textbf{be}\ (\lambda I\delta\,.\ (\mathcal{E}[\![E_{else}]\!]\ [I_{val} : \delta]e))\ \textbf{in} \\
\qquad\quad (\lambda v_{disc}\,.\ \textbf{matching}\ v_{disc} \\
\qquad\qquad \triangleright\, (\mathit{Sum} \mapsto \mathit{Value}\ \langle I_{tag},\ v\rangle) \\
\qquad\qquad\quad [\![\ ((\mathit{extend\text{-}tenv}^*\ e\ I_{val}\ [I_1 \ldots I_n]\ [E_1 \ldots E_n]\ \mathit{elsetenv})\ I_{tag}\ v) \\
\qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}comp} \\
\qquad\qquad \textbf{endmatching}\ ) \\
\qquad \textbf{letend}
\end{array}
$$

Figure 10.18: CBV denotational semantics for oneofs

```
(list (one rectangle (record (width 3) (height 4)))
      (one triangle (record (side1 5) (side2 6) (side3 7)))
      (one square (record (side 2))))
```

In this encoding, oneof tags are used to distinguish squares, rectangles, and triangles. The two sides of a rectangle (`width` and `height`) and three sides of a triangle (`side1`, `side2`, and `side3`) are named as fields in a record. Even though a square has only a single side length (`side`), it too is encapsulated in a record for uniformity. Of course, we could have used positional rather than named products, in which case the meaning of each position would need to be specified.

Manipulating a sum-of-product datum typically involves performing a case analysis on its tag and extracting the components of the associated record. For example, here is a procedure that calculates the perimeter of a shape:

```
(define (perim shape)
  (tagcase shape r
    (square (* 4 (select side r)))
    (rectangle (* 2 (+ (select width r) (select height r))))
    (triangle (+ (select side1 r)
                 (+ (select side2 r) (select side3 r))))))
```

As another example, consider the sum-of-product encoding of the ELM temperature conversion expression `(/ (* 5 (- (arg 1) 32)) 9)` shown in Figure 10.19. In this encoding, oneof tags distinguish arithmetic operations (`arithop`), integer literals (`lit`), and argument references (`arg`). The three components of an arithmetic operation — the operation (`op`) (a symbol) and two operands (`rand1` and `rand2`) are represented as a record. As with square shapes, the single number component of a literal expression and index component of an argument expression are boxed up into records for uniformity.

To handle this representation for ELM expressions, the `elm-eval` procedure from Chapter 6 would be rewritten:

```
(define (elm-eval exp args)
  (tagcase exp r
    (lit rcd (select num r))
    (arg rcd (arg-get (select index r) args))
    (arithop rcd ((primop->proc (select op r))
                  (elm-eval (select rand1 r) args)
                  (elm-eval (select rand2 r) args)))))
```

The rigidity of the above sum-of-product encodings is sometimes relaxed in practice. For instance, the case where a product has a single component can be optimized by replacing the product by the component value. If a product

```
(one arithop
  (record
    (op '/)
    (rand1 (one arithop
             (record
               (op '*)
               (rand1 (one lit (record (num 5))))
               (rand2 (one arithop
                        (record
                          (op '-)
                          (rand1 (one arg (record (index 1))))
                          (rand2 (one lit (record (num 32))))))))))
    (rand2 (one lit (record (num 9))))))
```

Figure 10.19: An ELM expression for converting temperatures from degrees Fahrenheit to degrees Celsius.

has zero components, it can be replaced by the unit value. In several popular data structures (including linked lists and binary trees), there are only two summands, one of which has no components. This situation is often handled by representing the non-trivial summand (e.g., list or tree node) directly as a product and representing the nullary summand (e.g., empty list or tree leaf) as a distinguished **null pointer** value. Conceptually, there is still a sum in this case: a value is *either* a null pointer or a node. But in terms of pragmatics, it is not necessary to associate a tag with a node because it is assumed that there is a cheap test that determines whether or not a node is the null pointer. For example, some runtime systems represent a null pointer with a value that contains all zeros to take advantage of efficient machine instructions for testing for zero.

Programming languages differ widely in terms of their support for sum-of-product data. For example:

- The ML and HASKELL programming languages have powerful facilities for declaring and manipulating sum-of-product data. We shall see similar facilities in the following sections.

- In object-oriented languages, such as JAVA, SMALLTALK, and C++, the dynamic dispatch performed when invoking a method on an object effectively performs a case analysis on the class (think tag) of the object, whose instance variables can be viewed as a record.

- In LISP dialects, it is common to represent a sum-of-product datum as a list

s-expression whose first element is a symbolic tag indicating the summand and whose remaining elements are the components of the product. For instance, the Fahrenheit-to-Centigrade conversion expression given above can be represented as the following Lisp s-expression:

```
(arithop /
         (arithop *
                  (lit 5)
                  (arithop - (arg 1) (lit 32)))
         (lit 9))
```

This, in turn, can be optimized without ambiguity into an s-expression identical to the ELM concrete s-expression syntax:

```
(/ (* 5
      (- (arg 1)
         32))
   9)
```

Indeed, syntax trees are without a doubt the most important sum-of-product data structure used in the study of programming languages. The ease with which they can be represented as s-expressions is the reason we have adopted s-expression grammars for the toy languages in this book.

- In document description languages like HTML and XML, summand tags appear in begin/end markups and product components are encoded both in the association lists of markups as well as in components nested within the begin/end markups. For instance, Figure 10.20 shows how the Fahrenheit-to-Centigrade expression might be encoded in XML. The reader is left to ponder why XML, which at one level is a verbose encoding of s-expressions, is a far more popular standard for expressing structured data than s-expressions. In fact, the WATER language [Plu02] goes the distance, using XML as a representation for s-expressions in a language with SCHEME-like semantics.

- In the C programming language, programmers must "roll their own" sum-of-product data structures using `union` and `struct`. For instance, Figure 10.21 shows how the geometric shape example from above can be expressed in C. In C, `union` is used to declare storage that can contain one of several different kinds of values. However, there is no built-in support for tagging such values. Instead, an explicit `struct` is typically used to associate a tag (`shapetag` in the example) with the value (`sum` in the example). Values with multiple components (e.g., `rect` and `tri`) are

```
<arithop>
  <op name="/"/>
  <rand1>
    <arithop>
      <op name="*"/>
      <rand1>
        <lit num=5/>
      </rand1>
      <rand2>
        <arithop>
          <op name="-"/>
          <rand1>
            <arg index=1/>
          </rand1>
          <rand2>
            <lit num=32/>
          </rand2>
        </arithop>
      </rand2>
    </arithop>
  </rand1>
  <rand2>
    <lit num=9/>
  </rand2>
</arithop>
```

Figure 10.20: The ELM Fahrenheit-to-Centigrade expression in XML notation.

themselves encoded via additional `struct` declarations.

As is apparent from the example in Figure 10.21, encoding sum-of-product data in C is awkward. Nesting `struct` declarations to provide explicit tags is cumbersome and leads to unwieldy name paths like `s.sum.rect.width`. But much worse is the fact that the language enforces no connection between the tag and the sum. For instance, consider the following sequence of C statements:

```
shape s4;
s4.tag = square;
s4.sum.rect.width = 8;
s4.sum.rect.height = 9;
printf("The perimeter of s4 is \%d\n", perim(s4));
```

Although conceptually it makes no sense to manipulate a rectangle's components in a square, in many C implementations, the above code compiles and runs without error, yielding 32 as the perimeter of `s4`. Why? Because the storage set aside for a `union` type is that required for the largest summand (in this case, the three integers of a triangle) and `s4.sum.side`, `s4.sum.rect.width`, and `s4.sum.tri.side1` are all just synonyms that reference the first slot of this storage.

This is a classic example of a **type loophole** in C. PASCAL's variant records, which encode sum-of-product datatypes in a way reminiscent of C, exhibit a similar type loophole. The same sort of undesirable behavior can be exhibited with the LISP s-expression `(square 8 9)`, for which a perimeter procedure would return 32 if the means of extracting the side of a square was returning the second element of an s-expression list. But the difference between LISP and C/PASCAL on this score is that C and PASCAL, unlike LISP, sport a static type system that might be expected to catch such type-related bugs at compile time. We will have much more to say about static typing in Chapter **??**.

10.4 Data Declarations

Programming with "raw" sums and products is cumbersome and error-prone. Here we study a high-level data declaration facility that simplifies the creation and manipulation of sum-of-product data. We extend our FL family of languages with a `define-data` declaration that specifies a new kind of sum-of-product data. We introduce this construct via a declaration for geometric shapes:

```
(define-data shape
  (square side)
  (rectangle width height)
  (triangle side1 side2 side3))
```

```
typedef enum {square, rectangle, triangle} shapetag;

typedef struct {
  shapetag tag;
  union {
    int side;
    struct {int width; int height;} rect;
    struct {int side1; int side2; int side3;} tri;
  } sum;
} shape;

int perim (shape s) {
  switch (s.tag) {
  case square:
    return 4*(s.sum.side);
  case rectangle:
    return 2*(s.sum.rect.width + s.sum.rect.height);
  case triangle:
    return (s.sum.tri.side1 + s.sum.tri.side2 + s.sum.tri.side3);
  }
}

int main () {
  shape s1, s2, s3;
  s1.tag = square;
  s1.sum.side = 2;
  s2.tag = rectangle;
  s2.sum.rect.width = 3;
  s2.sum.rect.height = 4;
  s3.tag = triangle;
  s3.sum.tri.side1 = 5;
  s3.sum.tri.side2 = 6;
  s3.sum.tri.side3 = 7;
  printf("The perimeter of s1 is \bs\%d\bs{}n", perim(s1));
  printf("The perimeter of s2 is \bs\%d\bs{}n", perim(s2));
  printf("The perimeter of s3 is \bs\%d\bs{}n", perim(s3));
}
```

Figure 10.21: The shape example encoded using `struct` and `union` in C.

This declaration specifies that a shape is either a square with one component, a rectangle with two components, or a triangle with three components. Each of the names `square`, `rectangle`, and `triangle` is a **value constructor procedure** (or just **constructor** for short) that takes the specified number of components and returns a sum-of-product datum with those components. For example, the list of shapes

```
(list (square 2) (rectangle 3 4) (triangle 7 8 9))
```

is equivalent to the list

```
(list (one square (product 2)))
      (one rectangle (product 3 4))
      (one triangle (product 5 6 7))
```

In contrast with the previous section, the sum-of-product data created by `define-data` constructors uses positional rather than named products.

In the example, the data name `shape` and the component names `side`, `width`, `height`, etc. are just comments. Only the *number* of components specified for a constructor is relevant. For instance, we could emphasize that all components are integers by writing

```
(define-data shape
  (square int)
  (rectangle int int)
  (triangle int int int)),
```

or we could use nonsense words to specify an equivalent declaration, as in

```
(define-data frob
  (square foo)
  (rectangle bar baz)
  (triangle quux quuux quuuux)).
```

The reason for requiring such comments is that the comment positions will assume a non-trivial meaning when we study a typed version of `define-data` in Chapter 15.

For every constructor procedure C that takes n arguments, `define-data` also declares an associated **deconstructor procedure** that takes three arguments:

1. the value v to be deconstructed;
2. a **success continuation**, an n-argument procedure that is applied to the n components of v in the case where v is constructed by C;
3. a **failure continuation**, a nullary procedure that is invoked in the case where v is not constructed by C.

We assume a convention in which the deconstructor has a name that is the name of the constructor followed by the tilde character, `~`, which is pronounced "twiddle." For instance, the `square~`, `rectangle~`, and `triangle~` deconstructors introduced by the `shape` declaration can be used to calculate the perimeter of a shape:

```
(define (perim s)
  (square~ s (lambda (s) (* 4 s))
    (lambda ()
      (rectangle~ s (lambda (w h) (* 2 (+ w h)))
        (lambda ()
          (triangle~ s (lambda (s1 s2 s3) (+ s1 s2 s3))
            (lambda ()
              (error not-a-shape)))))))
```

Deconstructors are somewhat awkward to use directly. In the next section we will study a pattern-matching facility based on deconstructors that significantly simplifies the deconstruction of sum-of-product data.

As another example of constructors and deconstructors, consider the `elm-exp` declaration in Figure 10.22. The `lit`, `arg`, and `arithop` constructors introduced by this declaration are illustrated in the Fahrenheit-to-Centigrade expression `f2c`, and the deconstructors `lit~`, `arg~`, and `arithop~` are used to define `elm-eval`.

We can even use `define-data` to define list constructors and deconstructors (Figure 10.23), replacing the desugaring given in Chapter 6.

A formal definition of `define-data` is presented in Figure 10.24. The syntax of FL programs is extended to include `define-data` clauses along with the usual definitions. The meaning of a `define-data` declaration can be explained by desugaring the declaration into a sequence of procedure definitions via $\mathcal{D}_{\text{def}}$, which has signature $D \rightarrow D^*$. The resulting sequence of definitions is spliced into the `program` construct, and all program definitions are further desugared as shown in Chapter 6. Each summand clause $(I_{tag}\ I_1\ \ldots\ I_n)$ desugars into 2 definitions:

- An n-argument constructor procedure named I_{tag} that constructs a oneof with tag I_{tag} of a product whose components are $I_1 \ldots I_n$.

- A three-argument deconstructor procedure that applies the second argument (an n-argument success continuation) to the n components of the product if the oneof has the right tag and otherwise invokes the third argument (a nullary failure continuation). The name of this deconstructor is created from the name I_{tag} by adding `~` as a suffix. We shall use the no-

```
(define-data elm-exp
  (lit num)
  (arg index)
  (arithop op rand1 rand2))

(define f2c (arithop '/
              (arithop '*
                       (lit 5)
                       (arithop '-
                                (arg 1)
                                (lit 32)))
              (lit 9)))

(define (elm-eval exp args)
  (lit~ exp (lambda (n) n)
    (lambda ()
      (arg~ exp (lambda (i) (get-arg i args))
        (lambda ()
          (arithop~ exp
            (lambda (op r1 r2)
              ((primop->proc op) (elm-eval r1 args) (elm-eval r2 args)))
            (lambda () (error not-an-elm-exp)))))))))
```

Figure 10.22: ELM examples.

```
(define-data list
  (null)
  (cons head tail))

(define (null? xs)
  (null~ xs (lambda () true) (lambda () false)))

(define (car xs)
  (cons~ xs (lambda (hd tl) hd)
    (lambda () (error car-of-nonlist-or-empty-list))))

(define (cdr xs)
  (cons~ xs (lambda (hd tl) tl)
    (lambda () (error cdr-of-nonlist-or-empty-list))))
```

Figure 10.23: Defining lists via `define-data`.

tation $I_1 \bowtie I_2$ to concatenate identifiers. For example, `square⋈~` denotes the identifier `square~`.

For example, Figure 10.25 shows the constructors and deconstructors introduced by the `shape` declaration.

Syntax

P ::= `(program` $D_{definitions}$`*` E_{body}`)` [Program]

D ::= `(define` I_{name} E_{value}`)` [Definition]
| `(define-data` I_{data} `(`I_{tag} I`*)*)`

Sugar

If D = `(define-data` I_{data} `(`I_{tag_1} $I_{1,1}$ ... I_{1,k_1}`)` ... `(`I_{tag_n} $I_{n,1}$ ... I_{n,k_n}`))`,
$\mathcal{D}_{\text{def}}[\![D]\!]$ = $\mathcal{D}_{\text{cl}}[\![$`(`$I_{tag_1}$ $I_{1,1}$ ... I_{1,k_1}`)`$]\!]$ @ ⋯ @ $\mathcal{D}_{\text{cl}}[\![$`(`$I_{tag_n}$ $I_{n,1}$ ... I_{1,k_n}`)`$]\!]$
and $\mathcal{D}_{\text{cl}}[\![$`(`$I_{tag_i}$ $I_{i,1}$ ... I_{i,k_i}`)`$]\!]$ =

Constructor	*Deconstructor*

[`(define (`I_{tag_i} `x1...x⋈k`$_i$`)`
`(one` I_{tag_i}
`(product x1...x⋈k`$_i$`)))`] ,
	`(define (`I_{tag_i}`⋈~ val succ fail)`
	`(tagcase val x`
	`(`I_{tag_i} `(succ (proj 1 x)`
	⋮
	`(proj k`$_i$ `x)))`
	`(else (fail))))`]

Figure 10.24: Syntax and desugaring of `define-data`.

▷ **Exercise 10.11** Extend the declaration of `elm-exp` and the definition of `elm-eval` to handle the full EL language. ◁

▷ **Exercise 10.12** It is possible to tweak the desugaring of `define-data` to use more efficient representations than those given in Figure 10.24.

a. Modify the `define-data` desugaring to avoid creating products for constructors that take zero or one argument.

b. Modify the `define-data` desugaring to represent a sum-of-products datum with tag I_{tag} and components $v_1 \ldots v_n$ as the heterogeneous sequence

`(sequence (symbol` I_{tag}`)` v_1 ... v_n`)`

(This desugaring makes sense for a dynamically typed language but not a statically typed one.) ◁

```
(define square
  (lambda (x1)
    (one square (product x1))))

(define square~
  (lambda (val succ fail)
    (tagcase val x
      (square (succ (proj 1 x)))
      (else (fail)))))

(define rectangle
  (lambda (x1 x2)
    (one rectangle (product x1 x2))))

(define rectangle~
  (lambda (val succ fail)
    (tagcase val x
      (rectangle (succ (proj 1 x) (proj 2 x)))
      (else (fail)))))

(define triangle
  (lambda (x1 x2 x3)
    (one triangle (product x1 x2 x3))))

(define triangle~
  (lambda (val succ fail)
    (tagcase val x
      (triangle (succ (proj 1 x) (proj 2 x) (proj 3 x)))
      (else (fail)))))
```

Figure 10.25: Value constructors and deconstructors introduced by the `shape` declaration.

▷ **Exercise 10.13** SML and HASKELL support user-defined datatype declarations. Below are the geometric shape declarations expressed in SML and HASKELL:

SML	HASKELL

`datatype Shape =`
`  Square of int`
`| Rectangle of int * int`
`| Triangle of int * int * int`
	`data Shape =`
	`  Square Int`
	`| Rectangle Int Int`
	`| Triangle Int Int Int`

In SML, passing multiple arguments to a data constructor is modeled by collecting the arguments into a tuple, as in `Triangle(5,6,7)`, where the tuple `(5,6,7)` has type `int * int * int`. It is a type error to supply the constructor with the wrong number of arguments, as in `Triangle(5,6)`.

In contrast, HASKELL data declarations allow curried constructors that can take multiple arguments one at a time. For instance, the invocation `Triangle 5 6` denotes a unary function that "expects" the third side of the triangle.

Is FL extended with `define-data` more like ML or HASKELL in this respect? For example, does `(triangle 5 6)` denote an error or a unary function? How would you change the desugaring of `define-data` to model the other language? ◁

▷ **Exercise 10.14** The desugaring for `define-data` in Figure 10.24 introduces two procedures (a constructor I_{tag} and a deconstructor $I_{tag}\bowtie\tilde{}$) for each summand clause $(I_{tag}\ \ I_1\ \ \ldots\ \ I_n)$. An alternative approach is to introduce $n+2$ procedures:

- An n-argument constructor procedure named I_{tag}.
- A unary predicate named $I_{tag}\bowtie\tilde{}$ that returns true for a oneof value with tag I_{tag} and false for any other oneof value. It is an error to apply this predicate to a value that is not a oneof value.
- n unary selector procedures named $I_1 \ldots I_n$, where I_i extracts the ith component of a product tagged with I_{tag}. It is an error to apply a selector procedure to a value that is not a oneof value or a oneof value with a tag that is not I_{tag}.

In this approach, the component names matter, since they are names of selectors, not just comments. For example, here is the `perim` procedure in this approach:

```
(define (perim s)
 (cond
    ((square? s) (* 4 (side s)))
    ((rectangle? s) (* 2 (+ (width s) (height s))))
    ((triangle? s) (+ (side1 s) (+ (side2 s) (side3 s))))
    ))
```

a. Give a desugaring for `define-data` that implements the new approach.

b. In your new desugaring, compare the evaluation of the conditional clause

```
((triangle? s) (+ (side1 s) (+ (side2 s) (side3 s))))
```

with the following deconstructor application in the original desugaring

```
(triangle~ s (lambda (s1 s2 s3) (+ s1 s2 s3))
  (lambda ()
    (error not-a-shape)))
```

Which evaluation is more efficient?

c. One drawback of having `define-data` desugar into so many procedures is that it increases the possibility of name conflicts. For instance, the `shape` declaration introduces procedures with names like `square`, `rectangle?`, and `width` that very well might be useful in other contexts. One way to address this problem is for programmers use more specific names within data declarations, as in:

```
(define-data shape
  (shape-square shape-side)
  (shape-rectangle shape-width shape-height)
  (shape-triangle shape-side1 shape-side2 shape-side3))
```

Another approach is to modify the desugaring for `define-data` to automatically concatenate the data type name with the name of every constructor, predicate, and selector procedure. For instance, something like this is done in COMMON LISP's `defstruct` facility. Discuss the benefits and drawbacks of these two ways to address potential name conflicts in a program with data declarations.

d. Yet another way to address name conflicts is to treat constructor, predicate, and selector applications as special forms that refer to a different namespace than the usual value namespace. Design an extension to FL that handles datat declarations based on this idea. Do you think it is a good way to handle name conflicts? ◁

10.5 Pattern Matching

10.5.1 Introduction to Pattern Matching

Deconstructors are a sufficient mechanism for dispatching on and extracting the components of sum-of-product data, but they are awkward to use in practice. It is more convenient to manipulate sum-of-product data using a **pattern matching** facility that simultaneously tests for a summand and names the components of the associated product when the test succeeds. We have made extensive use of a form of pattern matching (via the **matching** construct) in the mathematical metalanguage of this book. Pattern matching is also an important feature of some real-world programming languages, such as PROLOG, ML and HASKELL.

We will study pattern matching in the context of an extension to FL that includes `define-data` from the previous section along with a new `match` construct. First, we will give an informal introduction to `match` via a series of examples. Then we will describe the semantics of `match` in detail by desugaring it into deconstructor applications.

The `match` construct has the form `(match` E_{disc} `(`P_{pat} E_{body}`)*)`, where E_{disc} is the **discriminant** and each **match clause** of the form `(`P_{pat} E_{body}`)` has a **pattern** P_{pat} and a **body** E_{body}. A pattern P consists of either an FL literal value, an identifier, a wild card ("`_`"), or a tagged list of patterns:

P ::= L [Literal]
| I [Pattern Variable]
| `_` [Wild Card]
| `(`I P*`)` [Tagged List]

Informally, a `match` expression is evaluated by first evaluating E_{disc} into a value v_{disc}, then finding the first clause whose pattern P_i matches v_{disc}, and finally evaluating the associated body E_i of this clause relative to any bindings introduced by the successful match of v_{disc} to P_i. If no clause has a pattern matching v_{disc}, the `match` expression signals an error.

We begin with a few examples of `match` involving patterns that are just literals, identifiers, or wild cards. Here is a procedure that converts a boolean to an integer (and signals an error for a non-boolean input).

```
(define (bool->int b)
  (match b
    (#f 0)
    (#t 1)))
```

The `negate` procedure below returns a symbol that negates the sense of a `yes` or `no` input but returns `unknown` for any other input. The underscore pattern is a wildcard pattern that matches any discriminant.

```
(define (negate s)
  (match s
    ('yes 'no)
    ('no 'yes)
    (_ 'unknown)))
```

The following procedure returns one more than the square of a given number, except at the inputs -1 and 1, where it returns 0:

```
(define (squarish n)
  (match (* n n)
    (1 0)
    (x (+ 1 x))))
```

A pattern variable like `x` always successfully matches any discriminant value, and the name may be used to denote this value in the associated body expression.

To introduce tagged list patterns, we consider pattern matching involving lists of integers. Consider the following two procedures:

```
(define (match-ints-1 ints)
  (match ints
    ((cons x (null)) (* x x))
    (_ 17)
    ))

(define (match-ints-2 ints)
  (match ints
    ((cons x (null)) (* x x))
    ((cons 3 (cons y ns)) (+ y (length ns)))
    (_ 17)
    ))
```

- The pattern `(cons x (null))` matches a list that contains exactly one element, and names that element `x` in the scope of the body. So both procedures return the square of the first (only) element of the list when given a singleton list.

- The pattern `(cons 3 (cons y ns))` matches a list that has at least two elements, the first of which is the integer `3`. In the case of a match, the body is evaluated in a scope where the second element is named `y` and the list of all but the first two elements is named `ns`. So when this pattern matches, the second procedure returns the sum of the second element and the length of the rest of the list.

- The final wild card pattern in both procedures matches any value not matched by the first two patterns, in which case a `17` is returned.

The following table shows the results returned by these two procedures when supplied with various integer lists as an argument:

	`(list)`	`(list 3)`	`(list 3 4)`	`(list 6 8)`	`(list 3 6 8)`
`match-ints-1`	17	9	17	17	17
`match-ints-2`	17	9	4	17	7

The most important use of `match` is to perform pattern matching on user-defined sum-of-product data. For instance, here is a succinct version of the perimeter procedure based on pattern matching:

```
(define (perim shape)
  (match shape
    ((square s) (* 4 s))
    ((rectangle w h) (* 2 (+ w h)))
    ((triangle s1 s2 s3) (+ s1 (+ s2 s3)))))
```

```
(define (elm-eval exp args)
  (match exp
    ((lit n) n)
    ((arg i) (get-arg i args))
    ((arithop op r1 r2)
     ((primop->proc op) (elm-eval r1 args) (elm-eval r2 args)))))

(define (get-arg index nums)
  (match (list index nums)
    ((list 1 (cons n _)) n)
    ((list i (cons _ ns)) (get-arg i ns))))

(define (primop->proc sym)
  (match sym ('+ +) ('- -) ('* *) ('/ /)))
```

Figure 10.26: A complete ELM evaluator based on pattern matching.

The pattern `(square s)` matches a sum-of-product value constructed by the constructor application `(square` v_{side}`)`, in which case `s` names v_{side} in the body of the match clause. Similarly, the pattern `(rectangle w h)` matches a value constructed by `(rectangle` v_{width} v_{height}`)`, where `w` names v_{width} and `h` names v_{height}. The `triangle` pattern is handled similarly.

Some other nice illustrations of the conciseness of pattern matching involve the ELM language. Figure 10.26 presents a complete ELM evaluator based on pattern matching. The twelve lines of code are easy to understand and analyze. A compelling use of nested patterns is in the crude algebraic simplifier for ELM expressions in Figure 10.27. The second `match` clause in the `simp` procedure expresses that literals and argument references are self-evaluating (i.e., they simplify to themselves). The first clause simplifies an `arithop` by simplifying the arguments and then attempting to further simplify the resulting `arithop`. `simp-arithop` handles six special cases. The first four clauses express that zero is an identity for addition and one is an identity for multiplication. The next two clauses capture that multiplication by zero yields zero.[4] In order to appreciate the succinctness of pattern matching, the reader is encouraged to re-express the `simp` procedure in a version of FL that does not support pattern matching.

All the examples seen so far are "well-typed" in the sense that the discriminant of the `match` is "expected" to be a particular type (e.g., a list of integers, a shape, an ELM expression) and the results of all the clause bodies in a given `match` have the same type. But in a dynamically typed language, `match` is not

[4]This is not a safe transformation when the other subexpression contains a dynamic error!

```
(define (simp exp)
  (match exp
    ((arithop p r1 r2) (simp-arithop (arithop p (simp r1) (simp r2))))
    (x x)))

(define (simp-arithop exp)
  (match exp
    ((arithop '+ (lit 0) x) x)
    ((arithop '+ x (lit 0)) x)
    ((arithop '* (lit 1) x) x)
    ((arithop '* x (lit 1)) x)
    ((arithop '* (lit 0) _) (lit 0))
    ((arithop '* _ (lit 0)) (lit 0))
    (_ exp)))
```

Figure 10.27: An algebraic simplifier for ELM expressions.

required to have this behavior, as indicated by the following example:

```
(define (dynamic x)
  (match x
    ((0 #f)
     (#t 'zero)
     ('one 17))))
```

In Chapter **??**, we will study a statically typed version of FL in which `dynamic` will not be a legal procedure. However, all the other `match` examples above will still be legal.

10.5.2 A Desugaring-based Semantics of `match`

In order to motivate the structure of the desugaring of `match`, which is rather complex, we will incrementally develop the desugaring in the context of some concrete `match` examples rather than simply presenting the final desugaring. We begin with the `bool->int` procedure from the previous subsection:

```
(define (bool->int b)
  (match b
    (#f 0)
    (#t 1)))
```

It would be natural to desugar the `match` in `bool->int` into a series of `if` expressions:

```
(define (bool->int b)
  (if (equal? b #f)
      0
      (if (equal? b #t)
          1
          (error no-match))))
```

The case where `b` is not a boolean is handled by an explicit `error` expression indicating that the value of the discriminant did not match the pattern of any `match` clause.

In general, the discriminant of a `match` will be an arbitrary expression whose value should be calculated only once. To avoid recalculation of the discriminant, our `match` desugaring first names the discriminant (using `let`) and then performs a case analysis on the name. As shown in Exercise 10.19, this name can be eliminated when it is not necessary. For example, in `bool->int`, the discriminant is already bound to the variable `b`. Here is a revised desugaring for `bool->int` that names the discriminant:[5]

```
(define (bool->int b)
  (let ((disc b))
    (if (equal? disc #f)
        0
        (if (equal? disc #t)
            1
            (error no-match)))))
```

Whenever a mismatch between a pattern and a value is discovered, the matching process should stop processing the pattern in the current `match` clause and begin processing the pattern in the next `match` clause. When we study the desugaring of tagged patterns later, we will see that such a mismatch may be discovered at many different points in the processing of a given pattern. To avoid replicating the code that begins processing the pattern in the next `match` clause, our desugaring will wrap this code into a **failure thunk** that may potentially be invoked from several different points in the desugared code. Here is a version of the desugaring for `bool->int` that includes failure thunks named `fail1` and `fail2`:

[5] In the examples, all new identifiers introduced by the desugaring are assumed to be fresh so they do not clash with any program variables.

```
(define (bool->int b)
  (let ((disc b))
    (let ((fail1 (lambda ()
                   (let ((fail2 (lambda () (error no-match))))
                     (if (equal? disc #t)
                         1
                         (fail2))))))
      (if (equal? disc #f)
          0
          (fail1)))))
```

In the simple `match` within `bool->int`, each failure thunk is invoked exactly once. But soon we will see examples in which the failure thunk is invoked multiple times. In the case where the failure thunk is invoked zero or one times, it is possible for the desugarer to avoid introducing a named failure thunk. We leave this as an exercise.

The discussion so far leads to a first cut for the `match` desugaring shown in Figure 10.28. The desugaring of `match` is performed by $\mathcal{D}_{\text{match}}$. For simplicity, we assume that all `match` constructs are first eliminated by $\mathcal{D}_{\text{match}}$ in a separate pass over the program before other FL desugarings are performed. It is possible to merge all desugarings into a single pass, but that would make the description of the `match` desugaring more complex.

$\mathcal{D}_{\text{match}}[\![$`(match` E_{disc} `(`P_1 E_1`) ... (`P_n E_n`))`$]\!] =$
  `(let ((`I_{disc} E_{disc}`)) ;` I_{disc} `fresh`
    $(\mathcal{D}_{\text{clauses}}\ [P_1, \ldots, P_n]\ [E_1, \ldots, E_n]\ I_{disc})$

$\mathcal{D}_{\text{clauses}}\ [\,]\ [\,]\ I_{disc} =$`(error no-match)`
$\mathcal{D}_{\text{clauses}}\ (P_1\ .\ P_{rest}{}^*)\ (E_1\ .\ E_{rest}{}^*)\ I_{disc} =$
  `(let ((`I_{fail} `(lambda () ;` Failure thunk: if P_1 doesn't match, try the other clauses
                $(\mathcal{D}_{\text{clauses}}\ P_{rest}{}^*\quad E_{rest}{}^*\quad I_{disc})$`))))`
    $(\mathcal{D}_{\text{pat}}[\![P_1]\!]\ I_{disc}\ E_1\ I_{fail})$

$\mathcal{D}_{\text{pat}}[\![L]\!]\ I_{disc}\ E_{succ}\ I_{fail} =$ `(if (equal?`$_L$ I_{disc} L`)` E_{succ} `(`I_{fail}`))`
$\mathcal{D}_{\text{pat}}[\![\_]\!]\ I_{disc}\ E_{succ}\ I_{fail} =$ To be added
$\mathcal{D}_{\text{pat}}[\![I]\!]\ I_{disc}\ E_{succ}\ I_{fail} =$ To be added
$\mathcal{D}_{\text{pat}}[\![$`(`$I$ P_1 ... P_n`)`$]\!]\ I_{disc}\ E_{succ}\ I_{fail} =$ To be added

Figure 10.28: A first cut of the `match` desugaring.

$\mathcal{D}_{\text{match}}$ first introduces the fresh name I_{disc} for the value of the discriminant expression E_{disc} and then processes the `match` clauses via $\mathcal{D}_{\text{clauses}}$. The

$\mathcal{D}_{\text{clauses}}$ function takes three arguments: (1) a list of clause patterns, (2) a list of clause body expressions, and (3) the identifier naming the discriminant. The third argument allows the desugarer to refer to the discriminant by its identifier when processing the clauses. The $\mathcal{D}_{\text{clauses}}$ function uses $\mathcal{D}_{\text{pat}}$ to process the first pattern and body expression in a context where the fresh identifier I_{fail} names the failure thunk that processes the rest of the clauses. When no clauses remain, the desugarer yields an `error` expression that will be reached only when the desugared code for processing the clauses finds no pattern that matches the discriminant.

The core of the `match` desugaring is the $\mathcal{D}_{\text{pat}}$ function. This takes four arguments: (1) the pattern being matched, (2) the identifier naming the discriminant, (3) the **success expression** that is evaluated when the pattern matches the discriminant, and (4) the name of the failure thunk that is invoked when the pattern does not match the discriminant. A literal pattern is an easy case. The desugared code first compares the literal and discriminant via the equality operator $\texttt{equal}_L$. In a dynamically typed language, $\texttt{equal}_L$ is just the generic equality-testing procedure `equal?`, but when desugaring `match` in a statically typed language (as in Section 15.5), the equality operator $\texttt{equal}_L$ depends on the domain of the literal L. If the literal and discriminant are the same, the success expression is evaluated; otherwise, the failure thunk is invoked, which will either process the next `match` clause (if there is one) or signal a `no-match` error (if there is no next clause).

The literal case is the only $\mathcal{D}_{\text{pat}}$ case that is needed to explain the `bool->int` desugaring. The desugarings for the other three types of patterns (wildcards, identifiers, and tagged lists) are not shown in Figure 10.28 but will be fleshed out in the following discussion.

We first consider the wildcard pattern, as used in the `negate` procedure:

```
(define (negate s)
  (match s
    ('yes 'no)
    ('no 'yes)
    (_ 'unknown)))
```

The wildcard pattern always matches the discriminant, so the desugarer can simply emit the success expression for this case:

$$\mathcal{D}_{\text{pat}}[\![\_]\!]\ I_{disc}\ E_{succ}\ I_{fail} = E_{succ}$$

The result of desugaring the `match` expression within the `negate` procedure is:

```
(define (negate s)
  (let ((disc s))
    (let ((fail1
           (lambda ()
             (let ((fail2
                    (lambda ()
                      (let ((fail3 (lambda ()
                                     (error no-match))))
                        'unknown))))
               (if (equal? disc 'no) 'yes (fail2))))))
      (if (equal? disc 'yes) 'no (fail1)))))
```

It turns out that `fail3` can never be referenced, so the subexpression:

```
(let ((fail3 (lambda () (error no-match))))
  'unknown)
```

could simply be replaced by `'unknown`. This optimization could be performed by the desugarer itself or by a post-desugaring optimization pass (see Exercise 10.19.)

The case of patterns that are identifiers is similar to the wildcard case, except that the success expression must be evaluated in an environment where the identifier name is bound to the value of the discriminant:

$$\mathcal{D}_{\text{pat}}[\![I]\!]\ I_{disc}\ E_{succ}\ I_{fail} = \texttt{(let ((}I\ I_{disc}\texttt{))}\ E_{succ}\texttt{)}$$

As an example, consider the `squarish` procedure introduced above:

```
(define (squarish n)
  (match (* n n)
    (1 0)
    (x (+ 1 x))))
```

After desugaring the `match` expression within `squarish`, the procedure becomes:

```
(define (squarish n)
  (let ((disc (* n n)))
    (let ((fail1
           (lambda ()
             (let ((fail2 (lambda () (error no-match))))
               (let ((x disc))
                 (+ x 1))))))
      (if (equal? disc 1)
          0
          (fail1)))))
```

As in the `negate` example, the creation of the innermost failure thunk can be eliminated by an optimization (see Exercise 10.19). Note that the binding of

the discriminant to an identifier is significant here: `(* n n)` would otherwise be evaluated twice. If the discriminant expression performed any side effects, this would be a semantic issue as well as an efficiency concern.

The last case for $\mathcal{D}_{\text{pat}}$ is a tagged list pattern of the form `(`$I\ P_1\ \ldots\ P_n$`)`. Recall that I in this case is some sort of constructor procedure, such as `cons` or `triangle` in the pattern matching examples. Handling this case is tricky because it requires decomposing a constructed value into parts and recursively matching the subpatterns $P_1\ \ldots\ P_n$ against these parts. It turns out that deconstructor procedures are an excellent way to deal with tagged list patterns:

$\mathcal{D}_{\text{pat}}[\![(I\ P_1\ \ldots\ P_n)]\!]\ I_{disc}\ E_{succ}\ I_{fail}\ =$
  `(`$I\tilde{\bowtie}\ I_{disc}$
    `(lambda (`$I_1\ \ldots I_n$`)` ; Fresh identifiers for components.
      ; Match the component parts of the constructed value.
      $(\mathcal{D}_{\text{pats}}\ [P_1, \ldots, P_n]\ [I_1, \ldots, I_n]\ E_{succ}\ I_{fail})$`)`
    I_{fail}`)`

$\mathcal{D}_{\text{pats}}\ [\,]\ [\,]\ E_{succ}\ I_{fail}\ =\ E_{succ}$
$\mathcal{D}_{\text{pats}}\ (P_1\ .\ P_{rest}{}^*)\ (I_1\ .\ I_{rest}{}^*)\ E_{succ}\ I_{fail}\ =$
  $\mathcal{D}_{\text{pat}}[\![P_1]\!]\ I_1\ (\mathcal{D}_{\text{pats}}\ P_{rest}{}^*\ \ I_{rest}{}^*\ \ E_{succ}\ I_{fail})\ I_{fail}$

The $\mathcal{D}_{\text{pat}}$ function processes a tagged list pattern `(`$I\ P_1\ \ldots\ P_n$`)` by emitting code that invokes the deconstructor associated with I on the discriminant value denoted by I_{disc}, a success expression (call it E_{pats}) constructed by $\mathcal{D}_{\text{pats}}$, and the current failure thunk, denoted by I_{fail}. The E_{pats} expression is constructed by recursively matching the patterns $P_1 \ldots P_n$ against the components denoted by $I_1 \ldots I_n$ relative to the initial success expression E_{succ} and the failure thunk I_{fail}. Observe that I_{fail} is the same for all invocations of $\mathcal{D}_{\text{pat}}$ and $\mathcal{D}_{\text{pats}}$ in the processing of a single `match` clause, and that this I_{fail} denotes the failure thunk that processes the rest of the `match` clauses. This means that should there be any mismatch between the patterns and component values when E_{pats} is evaluated at run-time, I_{fail} will be invoked, terminating the attempt to match the current `match` clause against the discriminant, and starting to match the next `match` clause against the discriminant. On the other hand, if no mismatch is found when E_{pats} is evaluated, then the initial success expression E_{succ} will be evaluated in a context where all pattern variables are bound to the appropriate component values.

As concrete examples of desugaring tagged list patterns, we will study the desugarings of `match` within the `match-ints-1`, and `match-ints-2` procedures presented earlier. Recall that `match-ints-1` was defined as follows:

```
(define (match-ints-1 ints)
  (match ints
    ((cons x (null)) (* x x))
    (_ 17)
    ))
```

Here is a version of `match-ints-1` in which the `match` expression has been desugared:

```
(define (match-ints-1 ints)
  (let ((disc ints))
    (let ((fail1 (lambda ()
                   (let ((fail2 (lambda ()
                                  (error no-match))))
                     17))))
      (cons~ disc
        (lambda (v1 v2)
          (let ((x v1))
            (null~ v2
              (lambda () (* x x))
              fail1)))
        fail1))))
```

If the value denoted by `ints` and `disc` is a singleton list, then the `cons~` and `null~` deconstructors will both succeed, and `(* x x)` will be evaluated in an environment where `x` is bound to the single element (denoted by `x` and `v1`). If the discriminant is not a singleton list, then one of `cons~` or `null~` will invoke the failure continuation `fail1`, which returns the 17 specified in the second clause.

The code generated by the desugarer for `match-ints-1` is inefficient in many respects. By making the desugarer cleverer and/or transforming the result of the desugarer by a simple optimizer, it is possible to generate the following more compact and efficient code:

```
(define (match-ints-1 ints)
  (let ((fail1 (lambda () 17)))
    (cons~ ints
      (lambda (v1 v2)
        (null~ v2
          (lambda () (* v1 v1))
          fail1))
      fail1)))
```

As a second example of desugaring tagged list patterns, reconsider `match-ints-2`:

```
(define (match-ints-2 ints)
  (match ints
    ((cons x (null)) (* x x))
    ((cons 3 (cons y ns)) (+ y (length ns)))
    (_ 17)
    ))
```

The `match` desugaring functions yield the desugared definition in Figure 10.29. Everything is the same as the desugaring for `match-ints-1` except that the failure thunk `fail1` now corresponds to matching the second and third clauses of the `match` within `match-ints-2` and the failure thunk `fail2` now corresponds to matching the third clause. Note how the desugaring guarantees that the expression `(+ y (length ns))` is evaluated in an environment that contains correct bindings for the two names `y` and `ns`. Also observe that the second clause pattern `(cons 3 (cons y ns))` can fail to match the discriminant for three distinct reasons, all of which cause the invocation of the failure thunk `fail2`:

1. the discriminant `disc` is not a pair;
2. the discriminant `disc` is a pair whose first element `v3` is not 3;
3. the discriminant `disc` is a pair whose first element `v3` is 3 but whose second element `v4` is not a pair.

In general, a failure thunk is only invoked in two situations: (1) a literal is not equal to the value it is matched against or (2) a deconstructor invokes the failure thunk as its failure continuation when the discriminant does not match the associated constructor.

With the handling of tagged lists, we have completed the presentation of the desugaring of `match`. Whew! The complete desugaring rules for `match` are presented in Figure 10.30. Recall that we assume the usual FL desugaring is performed on the expression resulting from the `match` desugaring.

We have presented an approach to pattern matching based on desugaring and deconstructors. But this is by no means the only way to specify or implement pattern matching. For instance, the dynamic semantics for the core language of SML [MTHM97] treats pattern matching as a fundamental language feature that is explained via operational semantics rules. Whereas the deconstructor-based desugaring requires linearly testing the `match` clauses one-by-one in order, the SML definition does not imply a particular implementation. Indeed, there are clever implementations of ML pattern matching that can greatly reduce the number of tests that need to be performed [JM88].

```
(define (match-ints-2 ints)
  (let ((disc ints))
    (let ((fail1
           (lambda ()
             (let ((fail2
                    (lambda ()
                      (let ((fail3 (lambda ()
                                     (error no-match))))
                        17))))
               (cons~ disc
                 (lambda (v3 v4)
                   (if (equal? v3 3)
                       (cons~ v4
                         (lambda (v5 v6)
                           (let ((y v5))
                             (let ((ns v6))
                               (+ y (length ns)))))
                         fail2)
                       (fail2))))
                 fail2))))
      (cons~ disc
        (lambda (v1 v2)
          (let ((x v1))
            (null~ v2
              (lambda () (* x x))
              fail1)))
        fail1))))
```

Figure 10.29: The result of desugaring `match` in `match-ints-2`.

```
D_match⟦(match E_disc (P_1 E_1) ... (P_n E_n))⟧ =
  (let ((I_disc E_disc)) ; I_disc fresh
    (D_clauses [P_1,...,P_n] [E_1,...,E_n] I_disc))

D_clauses [] [] I_disc =(error no-match)
D_clauses (P_1 . P_rest*) (E_1 . E_rest*) I_disc =
  (let ((I_fail ; I_fail fresh
          (lambda () ; Failure thunk: if P_1 doesn't match, try the other clauses
            (D_clauses P_rest* E_rest* I_disc))))
    (D_pat⟦P_1⟧ I_disc E_1 I_fail))

D_pat⟦L⟧ I_disc E_succ I_fail = (if (equal? I_disc L) E_succ (I_fail))
D_pat⟦_⟧ I_disc E_succ I_fail = E_succ
D_pat⟦I⟧ I_disc E_succ I_fail = (let ((I I_disc)) E_succ)
D_pat⟦(I P_1 ... P_n)⟧ I_disc E_succ I_fail =
  (I⋈~ I_disc
    (lambda (I_1 ...I_n) ; Fresh identifiers for components.
      ; Match the component parts of the constructed value.
      (D_pats [P_1,...,P_n] [I_1,...,I_n] E_succ I_fail))
    I_fail)

D_pats [] [] E_succ I_fail = E_succ
D_pats (P_1 . P_rest*) (I_1 . I_rest*) E_succ I_fail =
  D_pat⟦P_1⟧ I_1 (D_pats P_rest* I_rest* E_succ I_fail) I_fail
```

Figure 10.30: The final version of the `match` desugaring

▷ **Exercise 10.15** Define the free identifiers of a `match` expression directly (i.e., without desugaring it). ◁

▷ **Exercise 10.16** Extend the `match` desugaring to directly handle record and oneof patterns. As an example of such patterns, consider the following alternative definition of the perimeter procedure.

```
(define (perim shape)
  (match perim
    ((one square (record (side s)))
     (* 4 s))
    ((one rectangle (record (width w) (height h)))
     (* 2 (+ w h)))
    ((one triangle (record (side1 s1) (side2 s2) (side3 s3)))
     (+ s1 (+ s2 s3)))
    ))
```

◁

▷ **Exercise 10.17** Extend the `match` desugaring to handle `list` patterns like those in the following procedure:

```
(define (match-list ints)
  (match ints
    ((list x) (+ x 1))
    ((list _ y) (* 2 y)
    ((list x y 3) (* x y))
    (_ 0)
    ))
```

For example:

```
(match-list (list)) ⟶FL 0
(match-list (list 4)) ⟶FL 5
(match-list (list 7 8)) ⟶FL 16
(match-list (list 5 4 3)) ⟶FL 20
(match-list (list 3 4 5)) ⟶FL 0
(match-list (list 1 2 3 4)) ⟶FL 0
```

◁

▷ **Exercise 10.18** Consider the following procedure for removing duplicates from a sorted list of integers:

```
(define (remove-dups sorted-list)
  (match sorted-list
    ((cons x (cons y zs))
     (if (= x y)
         (remove-dups (cons y zs))
         (cons x (remove-dups (cons y zs)))))
    (_ sorted-list)
    ))
```

Matching with nested tagged list patterns helps to extract the first two elements (`x` and `y`) of a list with at least two elements. But it is inelegant to name the remainder of such a list (`zs`) and to rebuild the tail of `sorted-list` via `(cons y zs)`.

One way to avoid these problems is to use nested `match` constructs:

```
(define (remove-dups-2 sorted-list)
  (match sorted-list
    ((cons x ys)
     (match ys
       ((cons y _)
        (if (= x y)
            (remove-dups ys)
            (cons x (remove-dups ys))))
       (_ sorted-list)))
    (_ sorted-list)
    ))
```

But this is verbose and requires duplication of the last `match` clause.

A more elegant approach is to introduce **named patterns** of the form `(<-> I P)`. When such a pattern is matched against a value v:

- if P matches v, then `(<-> I P)` also matches v, and the environment is extended with a binding between I and v as well as with any bindings implied by the match of P against v;
- if P does not match v, then `(<-> I P)` does not match v.

For example, with named patterns, `remove-dups` can be elegantly expressed as:

```
(define (remove-dups-3 sorted-list)
  (match sorted-list
    ((cons x (<-> ys (cons y _)))
     (if (= x y)
         (remove-dups ys)
         (cons x (remove-dups ys))))
    (_ sorted-list)
    ))
```

Extend the `match` desugaring to handle named patterns and show the result of your extended `match` desugaring for `remove-dups-3`. ◁

▷ **Exercise 10.19** Modify the `match` desugaring functions and/or define a post-desugaring optimizer to make the desugared code more compact and efficient. You should handle at least the following optimizations:

- Optimize unnecessary renamings of the form `(let ((`I_1 I_2`)) ...)`. E.g., the expression `(let ((x v1)) (* x x))` should be replaced by `(* v1 v1)`.
- Eliminate the creation of failure thunks that are never used. E.g., the expression `(let ((fail (lambda () `E_1`))) `E_2`)` should be replaced by E_2 if `fail` is not free within E_2.
- Eliminate the naming of failure thunks that are referenced only once. The single reference should be replaced by the `lambda` expression itself. E.g., the expression

  `(let ((fail `E_1`)) (cons~ `E_2 E_3 `fail))`

  should be replaced by `(cons~ `E_2 E_3 E_1`)`.
- Optimize the invocation of a manifest thunk. E.g., `((lambda () `E`))` should be replaced by E. ◁

10.5.3 Views

While the deconstructor-based desugaring of pattern matching may be inherently inefficient compared to other approaches, it provides an important advantage in expressiveness for the programmer. In languages like ML and HASKELL, sum-of-product datatypes can only be deconstructed by referencing the constructor in a pattern context. But using `match`, programmers can define arbitrary deconstructors from scratch and use them in patterns.

As an example, consider the `snoc`[6] procedure, which postpends an element to the back of a list:

```
(define (snoc xs x)
  (if (null? xs)
      (list x)
      (cons (car xs) (snoc (cdr xs) x))))
```

It is often handy to have a deconstructor corresponding to `snoc` that decomposes a non-empty list L into two values: the list of all elements in L excluding the last, and the last element L. This can be expressed with the following deconstructor[7]:

[6] So-called because it is a "backwards `cons`."

[7] An alternative approach to defining `snoc~` would be to express it in terms of two auxiliary procedures, one of which returns all but the last element of a non-empty list and the other of which returns the last element of a non-empty list. In such a definition, `snoc~` would walk over the given list twice. The definition given above effectively uses the success continuation to return multiple values and only walks over the given list once.

```
(define (snoc~ xs succ fail)
  (if (null? xs)
      (fail)
      (if (null? (cdr xs))
          (succ nil (car xs))
          (snoc~ (cdr xs)
                 (lambda (but-last last)
                   (succ (cons (car xs) but-last) last))
                 (lambda () (error cant-fail))))))
```

For example:

```
(snoc~ (list 1 2 3)
  (lambda (ns n) (cons n ns))
  (lambda () nil)) ⟶_FL [3, 1, 2]
```

Because of the way the `match` desugaring is defined, it is possible to invoke `snoc~` by referencing `snoc` in a pattern context. For example, here is a compact definition of a quadratic time list reversal procedure using `snoc~` via pattern matching:

```
(define (reverse xs)
  (match xs
    ((null) xs)
    ((cons _ (null)) xs)
    ((snoc ys y) (cons y (reverse ys)))))
```

The ability to choose from multiple deconstructors when decomposing a data structure characterizes what is known as a **views facility**, so-called because it allows a compound data value to be viewed from different perspectives depending on the context [Wad87]. For example, among the many possible views of a non-empty length-n list are

- the `cons` view: the list is the first element prepended onto a list containing elements 2 through n.
- the `snoc` view: the list is the nth element postpended onto the sublist containing the elements 1 through $(n-1)$.
- the `split` view: a list is the result of appending a left sublist (elements 1 through $\lceil n/2 \rceil$) and a right sublist (elements $\lceil n/2 \rceil + 1$ through n).
- the `interleave` view: a list is the result of interleaving a list containing all the odd-indexed elements with a list containing all the even-indexed elements.

- the `join` view: the list is the result of sandwiching element $\lceil n/2 \rceil$ between a left sublist (elements 1 through $\lceil n/2 \rceil - 1$) and a right sublist (elements $\lceil n/2 \rceil + 1$ through n).

These views show up in many standard list algorithms. For instance, the `interleave` (or `split`) view is at the heart of a mergesort algorithm for sorting lists:

```
(define (mergesort nums)
  (match nums
    ((null) nums)
    ((cons _ (null)) nums)
    ((interleave ms ns) ; Could decompose with split as well
     (merge (mergesort ms) ; merge left as an exercise
            (mergesort ns)))))
```

In addition to allowing compound data to be decomposed via pattern matching in different ways in different contexts, the views facility provided by user-defined deconstructor procedures helps to overcome a key drawback of ML and HASKELL style pattern matching: the lack of abstraction in patterns. While such patterns are wonderful for concisely specifying algorithms that manipulate sum-of-product data, the fact that they expose concrete implementation details hinders program development by making it difficult to change the implementation of data abstractions.

As an example of the sort of flexibility lost with ML-style patterns, consider a simple implementation of binary trees with integers stored in the nodes:

```
(define (node num left right) (product num left right))
(define (leaf) unit)
(define (leaf? t) (unit? t))
(define (val t) (proj 1 t))
(define (left t) (proj 2 t))
(define (right t) (proj 3 t))
```

Given these basic tree manipulation primitives, we can define many other tree procedures. For example:

```
(define (sum t)
  (if (leaf? t)
      0
      (+ (val t)
         (+ (sum (left t))
            (sum (right t))))))

(define (height t)
  (if (leaf? t)
      0
      (+ 1 (max (height (left t))
                (height (right t))))))
```

Suppose we wish to modify this implementation so that each node additionally keeps track of its height. This can be accomplished with only minor changes:

```
(define (node num left right)
  (product num left right
           (+ 1 (max (height left)
                     (height right)))))

(define (height t) (proj 4 t))
```

No other changes need to be made. In particular, procedures like `sum` that do not use the height remain unchanged.

Now instead suppose that we used sum-of-products data and pattern matching to implement the initial version of trees, where nodes did not maintain their height (Figure 10.31).

Let's now modify the nodes so that they maintain a height component. If we want `node` to remain a three-argument procedure, in an ML-style system, we must give a different name (say, `hnode`) to the constructor that takes a fourth argument, the height. In every pattern that uses `node`, we must change the constructor name to `hnode` and add an extra pattern to account for the height component (Figure 10.32).

It might seem easy to make these changes. But suppose we have hundreds of tree procedures in our program that needed to be changed in this manner. It would be tedious and error-prone to make the change everywhere — so much so that we might avoid making such representation changes. The concrete nature of ML-style patterns thus stands in the way of a software engineering principle that dictates that programming languages should be designed in such a way to facilitate changing representations.

A view mechanism like explicit deconstructors addresses this issue. When

```
(define-data int-tree
  (leaf)
  (node val left right))

(define (sum t)
  (match t
    ((leaf) 0)
    ((node v l r) (+ v (sum l) (sum r)))))

(define (height t)
  (match t
    ((leaf) 0)
    ((node v l r)
     (+ 1 (max (height l) (height r))))))
```

Figure 10.31: Integer binary trees expressed via `define-data` and `match`.

we introduce `hnode`, in addition to defining a new `node` procedure that has the same meaning as the old `node` constructor, we can also define a new `node~` deconstructor:

```
(define (node~ val succ fail)
  (match val
    ((leaf) (fail))
    ((hnode v l r h) (succ v l r))))
```

With this deconstructor, the original definition of `sum` that used `node` in its `match` clause need not be modified even though the representation of nodes has changed. In this way, user-defined deconstructors (and view facilities in general) facilitate representation changes to programs.

▷ **Exercise 10.20** Define the list deconstructors `split~`, `interleave~`, and `join~` described in the discussion on views. Give examples of algorithms where such views are helpful. ◁

▷ **Exercise 10.21** Define a `partition~` deconstructor for a non-empty list of integers L that decomposes it into three parts:

a. the first element of L (known as the **pivot**);

b. a list of all elements in the tail of L less than or equal to the pivot (with the same relative order as in L);

c. a list of all elements in the tail of L that are strictly greater than the pivot (with the same relative order as in L).

```
(define-data int-tree
  (leaf)
  (hnode val left right height))

(define (node v l r)
  (+ 1 (max (height l) (height r))))

(define (sum t)
  (match t
    ((leaf) 0)
    ((hnode v l r _) (+ v (sum l) (sum r)))))

(define (height t)
  (match t
    ((leaf) 0)
    ((hnode v l r h) h)))
```

Figure 10.32: Adding a height component requires changing all `node` patterns.

Using your `partition~`, it should be possible to define the quicksort algorithm for sorting lists:

```
(define (quicksort nums)
  (match nums
    ((null) nums)
    ((cons _ (null)) nums)
    ((partition pivot lesses greaters)
     (append (quicksort lesses)
             (cons pivot (quicksort greaters))))))
```

◁

▷ **Exercise 10.22** The convention of naming deconstructors by extending the constructor name with the suffix "~" is really just a crude but simple way of associating a deconstructor with a constructor. Here we consider an alternative way to specify this association.

Suppose that FL is extended with a declaration construct, `define-constructor`, that associates a constructor name with *two* procedures: a constructor and its associated deconstructor. Using this construct, new list constructors `kons` and `knull` could be specified as follows:

```
(define-constructor kons
  (lambda (elt lst) (pair elt lst)) ; Constructor
  (lambda (val succ fail)           ; Deconstructor
    (if (pair? val)
        (succ (left pair) (right pair))
        (fail)))
  )

(define-constructor knull
  (lambda () #u)              ; Constructor
  (lambda (val succ fail)     ; Deconstructor
    (if (unit? val) (succ) (fail)))
  )
```

The intention is that the name declared by `define-constructor` can be used within expressions to denote the constructor procedure and within patterns to denote the deconstructor procedure. Sometimes it is necessary to access to the deconstructor procedure within an expression; for this case, FL is also extended with a new expression `(decon` *I*`)` that accesses the "deconstructor part" of *I*. For example:

```
(kons 1 (kons 2 (knull)))  ⟶FL  [1, 2]

(match (kons 1 (kons 2 (knull)))
  ((kons x (kons y (knull))) (+ x y)))  ⟶FL  3

((decon kons) (kons 1 (kons 2 (knull)))
   (lambda (hd tl) (kons hd (kons hd tl)))
   (lambda () (kons 5 (knull)))  ⟶FL  [1, 1, 2]

((decon kons) (knull)
   (lambda (hd tl) (kons hd (kons hd tl)))
   (lambda () (kons 5 (knull)))  ⟶FL  [5]
```

The `match` desugaring for this extended version of FL is the same as before except that within $\mathcal{D}_{\text{pat}}$, the occurrence of $I\bowtie\!\!\sim$ is replaced by `(decon` *I*`)`.

a. One way to model the semantics of `(define-constructor` *I* E_1 E_2`)` is to say that it binds the name *I* to the *pair* of values that result from evaluating E_1 and E_2. Extend the denotational semantics of FL to reflect this model, and explain (1) the meaning of `define-constructor`, (2) the invocation of constructors, and (3) the semantics of `decon`.

b. Another way to model the semantics of `(define-constructor` *I* E_1 E_2`)` is to say that the extended version of FL has *two* namespaces: one for "normal" values (including constructors) and one for deconstructors. Extend the denotational semantics of FL to reflect this model, and explain (1) the meaning of `define-constructor`, (2) the invocation of constructors, and (3) the semantics of `decon`.

c. What are the benefits and drawbacks of using `define-constructor` and `decon` vs. the convention of naming deconstructors with a `~`? ◁

Chapter 11

Concurrency

When you come to a fork in the road, take it.

— *Yogi Berra*

11.1 Motivation

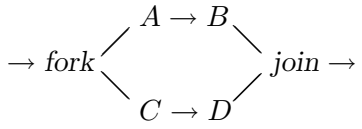
Thus far, all of the languages we have discussed have been characterized by a single locus of control that flows through a program in a deterministic fashion. Assuming all inputs are known in advance, there is only one path that control can take through the program. This single thread of control can be viewed as a time line along which all operations performed by the computation are arranged in a total order. For example, a computation that sequentially performs the operations A, B, C, and D can be depicted as the following total order:

$$\ldots \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow \ldots$$

$X \rightarrow Y$ means that X is performed before Y.

Control can be visualized as a token that resides on the edges of such a diagram and moves according to a set of rules. In a simple diagram like the one above, the only rule is that a token on the input edge to an operation can pass through the operation and end up on its output edge. This step corresponds to performing the operation. The path taken by control can be notated by the sequence of observable actions it performs. If we assume that all operations are observable, the path taken in this case is $ABCD$. In the languages we have considered so far (even those with conditional branching, non-local exits, and exception handling), every program, given a particular input, has exactly one sequential control path.

Purely sequential orderings are too rigid for describing all of the computations we might like to specify. Sometimes it is desirable to specify a relative order between some operations but leave the ordering of other operations unspecified. Here is a sample partial order that declares that A precedes B and C precedes D, but does not otherwise constrain the operation order:



The diagram introduces two new nodes labeled *fork* and *join*. The purpose of these nodes is to split and merge control paths in such a way that a computation has a distinguished starting edge and a distinguished ending edge.

The rules governing how control moves through *fork* and *join* nodes are different from the rules associated with the labeled operations. In one step, a control token on the input edge of a *fork* node splits into two subtokens on the output edges of the node. Each subtoken independently moves forward on its own branch. When tokens are on both input edges of a *join* node, they merge into a single token on the output node. If only one input edge to a *join* has a token, it cannot move forward until the other edge contains a token. Any node like join that forces one control token to wait for another is said to **synchronize** them.

Together, a control token and the sequential subpath along which it moves are called a **thread**. Although *fork* splits a single thread into two, it is common to view the original thread as continuing through one branch of the fork and a new thread as originating on the other branch of the fork.

Programs that may exhibit more than one thread of control are said to be **multi-threaded** or **concurrent**. The interesting feature of concurrency is that multiple threads represent a partially ordered notion of time. Concurrent programs are **non-deterministic** in the sense that control can flow through them in more than one way. Non-determinism at the level of control is often exhibited as non-determinism at the level of program behavior: it is possible for a single program to yield different answers on different executions. While it is possible to add non-determinism to a purely sequential language[1], non-determinism is most commonly associated with concurrent languages.

[1] As an example, consider a `(choose` E_1 E_2`)` form that randomly chooses to return the value of E_1 or E_2.

Suppose that on any step of a multi-threaded computation, only one control token is allowed to move.[2] Then a particular execution of a concurrent program is associated with the sequence of its observable actions, known as its **interleaving**. The **behavior** of the concurrent program is the set of all possible interleavings that can be exhibited by the program. For example, assuming that all operations (except for *fork* and *join*) are observable, then the behavior of the branching diagram above is:

$$\{ABCD,\ ACBD,\ ACDB,\ CABD,\ CADB,\ CDAB\}$$

The behavior of a concurrent program may be the empty set (no interleavings are possible), a singleton set (exactly one interleaving is possible), or a set with more than one element (many interleavings are possible).

We will distinguish concurrency from **parallelism**.[3] In our terminology, concurrency refers to any model of computation supporting partially ordered time. In contrast, we use parallelism to refer to hardware configurations that are capable of simultaneously executing multiple threads on multiple physical processors. Thus, concurrency is a semantic notion and parallelism is a pragmatic one. A concurrent program may be executed by time-slicing its threads on a single processor, or by executing each thread on a separate physical processor.

Concurrent programming languages are important for several reasons:

- *Modularity*: Multiple threads can be simulated in a sequential language by interleaving code fragments or by explicitly managing the transfer of control between program parts. Concurrent languages reap modularity benefits by abstracting over these idioms. They enhance modularity by permitting threads to be specified as separate entities that interact with each other via communication and synchronization. They also separate the specification of the threads from policy issues (such as scheduling threads or allocating threads to processors).

  For example, consider the interaction between the processor(s) of a computer and its numerous input/output devices (keyboards, mice, disks, tape drives, networks, video displays, printers, plotters, etc.). Writing a monolithic program to control all these devices would be a recipe for disaster. Such a program would be hard to understand and modify. In contrast, representing the controller for each device as a separate thread improves readability and facilitates the modification of existing controllers as well as the addition of new ones.

[2] There are concurrent models in which multiple control tokens can move in a single step, but we shall not consider these.

[3] While there is much disagreement about the definition of these terms in the programming languages community, we will strive to use them consistently here.

- *Specifying Parallelism*: Concurrent languages allow programmers to declare which parts of programs can safely be run in parallel on multiple processors or time-sliced on a single processor. Language implementations can use this information to take advantage of the available hardware.

- *Modelling*: The real world can be viewed as a collection of interacting agents. Concurrent languages are natural for simulating the world, because they allow each agent to be represented as a thread.

- *New programming paradigms*: Concurrent languages support programming paradigms that are cumbersome to express in sequential languages. Consider event-based systems, in which entities generate and respond to events. Such systems are more straightforward to express in concurrent languages than sequential languages.

11.2 Threads

We will explore concurrency by providing a precise semantics for a multi-threaded variant of FL!. It is fairly simple to provide an operational semantics for concurrency, and we will use the SOS framework to do so. The reason that operational semantics approaches to concurrency are the easiest is that they simply incorporate the inherent ambiguity about process interleaving into rules, and an SOS semantics can derive alternative meanings for a program based upon your choice of transition ordering. An operational semantics does not necessarily give you any help to find transition orderings that will yield all of the unique meanings for a given program.

An alternative, more complex approach to modeling concurrency is to treat the meaning of a program as the set of all possible answers the program can compute. A denotational semantics of concurrency takes this approach. A denotational semantics represents the meaning of a concurrent program as a powerdomain. A powerdomain of D is the set of all subsets of D. Thus, if D is the set of integers, the powerdomain of D contains all possible sets of integers. A denotational approach considers all possible interleavings of program operations to produce the meaning of a program. We will not pursue denotational approaches to concurrency further because concurrency is readily described within the SOS framework. Denotational approaches are complex because the functional nature of denotational semantics is hard to adapt to a world where functions are one to many, and interfering concurrent commands produce complex valuation function constructions. The interested reader can explore denotational approaches to concurrency to see how they can be adapted to our discussion [Sch86a]).

11.2.1 MUFL!, a Multi-threaded Language

Our main vehicle for studying concurrency will be MUFL!, a *Mu*ltithreaded version of FL!. MUFL! is FL! extended with the following thread constructs:

E ::= ...| `(fork` E`)` | `(join` E`)` | `(thread?` E`)`

- `(fork` E`)`: Creates a new thread to evaluate E and returns a unique **thread handle** identifying the thread.
- `(join` E`)`: E must evaluate to a thread handle t; otherwise, the expression is an error. `join` waits for the thread t to compute a value and then returns it. While `join` is waiting for t, the thread t' in which the `join` is executing is said to be **blocked**.

  `join` may be called more than once on the same thread. After the first call to `join` returns with a value, that value is effectively memoized at the thread handle for subsequent `join`s.
- `(thread?` E`)`: tests whether the value of E is a thread, returning *true* if it is and *false* otherwise.

`join` and `thread?` can actually be treated as new primitive operators, and we will do so below in describing the semantics of these constructs. However, because `fork` returns before evaluating its operand, it must be a new special form.

We consider a few simple examples of the new constructs. `(join (fork` E`))` is equivalent to E:

```
(join (fork (+ 1 2))) ⟶ 3
                  MUFL!
```

A thread handle can be `join`ed more than once:

```
(let ((c (cell 0)))
  (let ((t (fork (begin (cell-set! c (+ 1 (cell-ref c)))
                        (cell-ref c)))))
    (+ (join t) (join t)))) ⟶ 2
                        MUFL!
```

Here is a procedure for summing the leaves of a binary tree that explores subtrees concurrently:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? tree)
        (leaf-value tree)
        (let ((thread1 (fork (tree-sum (left-branch tree))))
              (thread2 (fork (tree-sum (right-branch tree)))))
          (+ (join thread1) (join thread2))))))
```

Without the `fork`s and `join`s, the left-to-right operand evaluation order inherited from FL! would specify that the sum of a left subtree be computed before the right subtree is visited. The order would be overconstrained even if the `let` expression were replaced by the result of substituting the `fork` expressions for `thread1` and `thread2`:

```
(+ (join (fork (tree-sum (left-branch tree))))
   (join (fork (tree-sum (right-branch tree)))))
```

The reason is that the result of the first `join` would have to be computed before control passed to the second `join`.

MUFL! programs can exhibit non-determinism. For example, suppose that `display` prints (uninterruptably) a symbol, `map` maps a procedure over a list to create a new one, and `for-each` performs a procedure on each element of a list. Then the following `jumble` procedure may print any permutation of the elements of its argument list before returning `#u`.

```
(define (jumble lst)
  (for-each join
            (map (lambda (obj) (fork (display obj)))
                 lst)))
```

For instance, `(jumble '(a b c))` may print any of the six sequences `abc`, `acb`, `bac`, `bca`, `cab`, `cba`.

The following `choose` procedure, which may return either one of its two arguments, underscores the non-deterministic behavior of MUFL!:

```
(define (choose a b)
  (let ((c (cell 'ignore)))
    (let ((t (fork (cell-set! c a))))
      (begin (cell-set! c b)
             (join t)
             (cell-ref c)))))
```

The final value returned by `choose` depends on the order in which the two cell assignments are performed. This example exploits what is known as a **race condition**. A race condition exists when two operations vie for the use of some shared resource. In this case, there is contention for the use of the shared cell `c`, and the thread that gets the cell last is the one that determines its value.

Threads introduce a new failure mode for programs: **deadlock**. Deadlock describes a situation in which program execution cannot proceed because threads are waiting for each other (or themselves) to complete. Consider the following expression:

```
(let ((thread-cell (cell 'later)))
  (begin
    (cell-set! thread-cell
               (fork (begin (join (cell-ref thread-cell))
                            1)))
    (join (cell-ref thread-cell))))
```

The forked `begin` can only return `1` after the thread stored in `thread-cell` runs to completion. But the thread stored in `thread-cell` is the one executing the `begin` expression! Since the `begin` expression can never return, the final `join` waits forever in a deadlocked state.

There is also a race condition in this example that shows why debugging multi-threaded code can be very difficult. In this case, the `fork` begins the execution of its `begin` expression while execution also procedes in the `cell-set!`. If the `cell-ref` in the `begin` executes before the `cell-set!` completes, then the `join` in the `begin` expression is an error. If the `cell-set!` expression wins the race, then the example deadlocks. In fact, executions that contain the error may also deadlock, depending on the precise semantics of errors and exceptions in MUFL!.

11.2.2 An Operational Semantics for MUFL!

Given the subtleties introduced by concurrency, it is more important than ever to precisely specify the semantics of programs. Here we introduce a complete SOS for MUFL!. The SOS for MUFL! introduces two domains:

$T \in$ Thread-Handle $=$ Intlit
$A \in$ Agenda $=$ Thread-Handle $\rightarrow$ MixedExp

A thread handle is just an integer literal that serves as a unique identifier for threads. An agenda is a partial function that maps a thread handle to an element of MixedExp, the domain of intermediate expressions. We assume that the grammar defining ValueExp for FLK! is extended to include a new intermediate form, `(*thread*` T`)`, that represents a first class thread handle value.

$V \in$ ValueExp
$V ::= \ldots$ [Value Expressions]
| `(*thread*` T`)` [Thread Values]

Because an agenda is a partial function, it may be defined only on a subset of thread handles. *dom(A)* is the notation for the set of inputs on which A is defined. We will use the notation $A_\emptyset$ to stand for an agenda that is nowhere defined and the notation $A[T = M]$ to stand for an agenda that maps T to M and maps every other T' in *dom(A)* to $(A\ T')$. These notations are useful for any partial function; in fact, we shall use them for stores as well as agendas.

Suppose that T_{top} is a distinguished thread handle for top-level MUFL! expressions. Then an SOS for the MUFLK!, the MUFL! kernel, is

$$\langle CF, \Rightarrow, FC, IF, OF \rangle$$

where

$$\begin{aligned}
CF &= \text{Agenda} \times \text{Store} \\
FC &= \langle A[T_{top} = V], S \rangle \\
IF &= \lambda E \,.\, \langle A_{\emptyset}[T_{top} = E], S_{\emptyset} \rangle \\
OF &= \lambda \langle A[T_{top} = V], S \rangle \,.\, (output\ \ V) \\
(output\ \texttt{(*thread* T)}) &= \texttt{thread} \\
(output\ \ V) &= (output_{FLK!}\ \ V) \text{ where } V \neq \texttt{(*thread*}\ T\texttt{)}
\end{aligned}$$

In a configuration, the agenda keeps track of the evaluation of the expression associated with each thread handle, while the store manages cell states. The input function maps a top-level expression E_{top} to a configuration with an empty store and an agenda whose only thread T_{top} evaluates E_{top}. The transition rules rewrite the configuration until the top-level expression is evaluated or no more rules are applicable. If the top-level expression has been rewritten to a value expression by this point, then a representation of this value is returned. If the top-level expression is not a value and no more progress is possible, the computation is deadlocked.

The transition rules for the MUFLK! SOS are summarized in Figure 11.1.

Each rule allows the one-step progress of the expression associated with a single thread handle. In any configuration, at most one transition is possible at a given thread handle. However, transitions may be possible at several thread handles within one configuration, so the rules are non-deterministic.

The [*fork*] rule allocates a new thread handle for the body of the `fork` and returns it to the thread containing the `fork` expression. `fork` is the only means by which threads are created; there is no mechanism for destroying threads (removing them from the agenda). The [*join*] rule indicates that the `join` of a thread handle T' cannot proceed until the expression associated with T' has progressed to a value. `join` is treated as a primop so that it is not necessary to specify a progress rule for evaluating its operand. Similarly, `thread?` is handled as a primop that determines whether its argument is a thread handle.

Figure 11.1 also presents two **meta-rules** for deriving MUFLK! rewrite rules from the rewrite rules for FLK!. The [*FLK!-axiom*] meta-rule says that any axiom for rewriting FLK! expressions can be applied to the expression of a single MUFLK! thread. The [*FLK!-progress*] meta-rule similarly specifies how

REWRITE RULES

$$\langle A[T = \texttt{(fork } E\texttt{)}], S\rangle \Rightarrow \langle A[T = \texttt{(*thread* } T'\texttt{)}][T' = E], S\rangle, \quad \text{where } T' \notin dom(A) \qquad [fork]$$

$$\langle A[T = \texttt{(primop join (*thread* } T'\texttt{))}][T' = V], S\rangle \Rightarrow \langle A[T = V][T' = V], S\rangle \qquad [join]$$

$$\langle A[T = \texttt{(primop thread? (*thread* } T'\texttt{))}], S\rangle \Rightarrow \langle A[T = \texttt{\#t}], S\rangle \qquad [thread?\text{-}true]$$

$$\langle A[T = \texttt{(primop thread? } V\texttt{)}], S\rangle \Rightarrow \langle A[T = \texttt{\#f}], S\rangle, \quad \text{where } V \text{ is not of the form (*thread* } T'\text{)} \qquad [thread?\text{-}false]$$

META-RULES

For each axiom
$\langle E, S\rangle \Rightarrow \langle E', S'\rangle$
in the FLK! SOS, include the following axiom in the MUFLK! SOS:
$\langle A[T = E], S\rangle \Rightarrow \langle A[T = E'], S'\rangle$
[FLK!-*axiom*]

If X is an FLK expression context and $X\{E\}$ is the result of filling the hole of the context with E, then for each FLK! progress rule of the form

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle X\{E\}, S\rangle \Rightarrow \langle X\{E'\}, S'\rangle} \qquad [FLK!\text{-}progress]$$

include the following progress rule in the MUFLK! SOS:

$$\frac{\langle A[T = E], S\rangle \Rightarrow \langle A'[T = E'], S'\rangle}{\langle A[T = X\{E\}], S\rangle \Rightarrow \langle A'[T = X\{E'\}], S\rangle}$$

Figure 11.1: Rewrite rules and meta-rules for MUFLK!.

to derive MUFLK! progress rules from FLK! progress rules. Note that the [*FLK!-progress*] meta-rule uses two agenda meta-variables, A and A'. This is necessary for handling antecedent transitions in which an inner `fork` extends the agenda with a new thread. If instead A were used on both sides of the antecedent transition, the effect of a `fork` nested within an expression could never be propagated.

For deterministic languages like PostFix and FL!, behavior is defined as the partial function that maps a program to the unique result determined by the operational semantics. But in the presence of non-determinism, there may be more than one result. So it is necessary to extend the notion of behavior to be a *relation* between programs and results. Equivalently, we can define behavior as a total function that maps programs to the powerset of results. In this approach, behavior maps a program to the set of all the results that can be determined for the program via the operational semantics.

As an example of the non-determinism of MUFL! programs, consider the following expression:

```
(let ((c (cell 0)))
  (let ((t (fork (cell-set! c 1))))
    (begin (cell-set! c 2)
           (join t)
           (cell-ref c))))
```

The configuration representing the state of the computation after the allocation of the cell and the thread handle is:

$\langle A_\emptyset[T_{top}=$ `(begin (cell-set! (*cell*` L_1`) 2)`
`(join (*thread*` T_1`))`
`(cell-ref (*cell*` L_1`)))]`
$[T_1=$`(cell-set! (*cell*` L_1`) 1)`$]$,
$S_\emptyset[L_1=0]\rangle$

(For convenience, we are being somewhat loose in our notation, allowing threads to name expressions from the full MUFL! language rather than just kernel expressions.) Figures 11.2 and 11.3 show two possible transition paths that can be taken from this configuration. The first computes a final value of `1`, while the second computes a final value of `2`. Since these are the only two possible results, the behavior of the expression is $\{1, 2\}$.

11.2.3 Other Thread Interfaces

The `fork`/`join` mechanism introduced above is only one interface to threads. Here we discuss some other approaches.

TRANSITION PATH 1:

$\langle A_\emptyset\ [T_{top}=$ `(begin (cell-set! (*cell* `L_1`) 2)`
`(join (*thread* `T_1`))`
`(cell-ref (*cell* `L_1`)))]`
$[T_1=$`(cell-set! (*cell* `L_1`) 1)`$]$,
$S_\emptyset[L_1=0]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset\ [T_{top}=$ `(begin (join (*thread* `T_1`))`
`(cell-ref (*cell* `L_1`)))]`
$[T_1=$`(cell-set! (*cell* `L_1`) 1)`$]$,
$S_\emptyset[L_1=2]$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset\ [T_{top}=$ `(begin (join (*thread* `T_1`)) (cell-ref (*cell* `L_1`)))]
$[T_1=$ `#u`$]$,
$S_\emptyset[L_1=1]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top}=$ `(cell-ref (*cell* `L_1`))`$][T_1=$ `#u`$], S_\emptyset[L_1=$ `1`$]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top}=$ `1`$][T_1=$ `#u`$], S_\emptyset[L_1=$ `1`$]\rangle$

$(OF\ \langle A_\emptyset[T_{top}=$ `1`$][T_1=$ `#u`$], S_\emptyset[L_1=$ `1`$]\rangle) = 1$

Figure 11.2: Sample transition paths demonstrating the non-determinism of MUFL! programs, Part I.

TRANSITION PATH 2:

$\langle A_\emptyset$ [T_{top}= `(begin (cell-set! (*cell* L1) 2)`
`(join (*thread* T1))`
`(cell-ref (*cell* L1)))`]
[T_1=`(cell-set! (*cell* L1) 1)`],
$S_\emptyset[L_1 = 0]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset$ [T_{top}= `(begin (cell-set! (*cell* L1) 2)`
`(join (*thread* T1))`
`(cell-ref (*cell* L1)))`]
[T_1 = `#u`],
$S_\emptyset[L_1 = 1]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset$ [T_{top}= `(begin (join (*thread* T1)) (cell-ref (*cell* L1)))`]
[T_1 = `#u`],
$S_\emptyset[L_1 = 2]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top} =$ `(cell-ref (*cell* L1))`$][T_1 =$ `#u`$], S_\emptyset[L_1 = 2]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top} = 2][T_1 =$ `#u`$], S_\emptyset[L_1 = 2]\rangle$

$(OF\ \langle A_\emptyset[T_{top} = 2][T_1 =$ `#u`$], S_\emptyset[L_1 = 2]\rangle) = 2$

Figure 11.3: Sample transition paths demonstrating the non-determinism of MUFL! programs, Part II.

11.2.3.1 Other thread operations

Many concurrent languages support variants of `fork` and `join`. In some languages, a thread can be `join`ed only once. In such languages, `join` has the effect of destroying the thread from the agenda as well as synchronizing, and joining a thread more than once is an error.

Some languages support other thread operations, such as destroying the thread, suspending it (temporarily removing it from the agenda), and resuming it (adding it back to the agenda). Sometimes there is a parent/child relationship between the thread that contains the `fork` and the resulting thread. In this case, thread operations on a parent can affect the children and vice versa. For example, destroying a parent thread might also destroy all its descendants.

11.2.3.2 Thread Abstractions

There are many useful thread abstractions that can be built on top of `fork` and `join`. For example, it is common to use `fork` and `join` to evaluate a set of expressions concurrently and then manipulate their results when all are finished. This idiom is captured in the `cobegin` construct, which is defined by the following desugaring:

```
(cobegin E1 ... En)
 = (let ((I1 (fork E1))
         ⋮
         (In (fork En)))
     (list (join I1) ... (join In)))
```

This version of `cobegin` returns a list of results, but other options are to return the first value, last value, or the unit value.

A variant on `cobegin` is a `colet` construct that binds names to the results of expressions that are computed concurrently:

```
(colet ((I1 E1) ... (In En)) Ebody)
 = (let ((I1 (fork E1))
         ⋮
         (In (fork En)))
     (let ((I1 (join I1))
           ⋮
           (In (join In)))
       Ebody))
```

Using `colet`, the concurrent tree-sum example can be expressed as:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? n)
        (leaf-value n)
        (colet ((left-sum (tree-sum (left-branch tree)))
                (right-sum (tree-sum (right-branch tree))))
          (+ left-sum right-sum)))))
```

11.2.3.3 Futures

A **future** is a thread that is implicitly `join`ed in any context that needs to examine its value. (In the context of futures, `join` is usually called `touch`.) Examples of such contexts are operands of strict primitives, the test position of a conditional, and the operator position of a `call`. Futures are supported in a number of LISP dialects [Hal85, Mil87, For91].

Futures are created by the construct `(future` E`)`. Using futures, the concurrent tree summation program can be expressed as:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? n)
        (leaf-value n)
        (+ (future (tree-sum (left-branch tree)))
           (future (tree-sum (right-branch tree)))))))
```

Futures are more modular than `fork/join` because it is possible to sprinkle `future`s into a program without having to guarantee that `join`s are placed wherever a future might be used. However, in a straightforward implementation, every touching context must check whether or not a value is a future, and, if it is, touch it. This overhead is similar to that incurred by memoization in lazy evaluation.

11.2.3.4 Eager Evaluation

The similarity between futures and lazy evaluation suggests a parameter passing mechanism in which every argument is implicitly wrapped in a `future`. This mechanism is called **eager evaluation**. We will refer to it as **call-by-eager** (CBE) by analogy with call-by-lazy. An example of a language supporting CBE is ID, a mostly functional language designed to be run on parallel computers [AN89].

Under CBE, an argument is evaluated concurrently with other arguments and the body of the procedure. In contrast with CBL, which only evaluates an

argument if it is used in the body, CBE evaluates the argument whether or not it is needed.

In a version of MUFL! with CBE parameter passing, the concurrent tree summation program would be expressed as:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? n)
        (leaf-value n)
        (+ (tree-sum (left-branch tree))
           (tree-sum (right-branch tree))))))
```

Note that this program is indistinguishable from a program in CBV FL. CBE removes inessential constraints governing evaluation order of expressions and emphasizes that the only fundamental constraints are induced by data dependencies.

An interesting feature of CBE is that a procedure may return before its arguments have been evaluated. For example, suppose the following expression is evaluated under CBE:

```
(begin (call (lambda (x) (display 'b))
             (display 'a))
       (display 'c))
```

Assuming `display` does not return until the output operation is successful, the possible displayed outputs of this expression are `abc`, `bac`, and `bca`. In each case, the fact that the procedure body must be fully evaluated before the `call` returns forces `b` to be displayed before `c`. However, because the argument `x` is never used within the body of the `lambda`, the `a` can be displayed at any time, even after the procedure has returned.

11.3 Communication and Synchronization

Concurrent programs are often patterned after interacting entities in the real world. Completely independent threads are inadequate for modelling the interactions between such entities. Here we explore various ways in which threads may interact. We focus on two kinds of interaction:

- **Communication**: Information computed by one thread often needs to be transmitted to another thread.
- **Synchronization**: When concurrently executing threads share state, their accesses and updates must often be carefully choreographed in order to achieve a desired effect.

11.3.1 Shared Mutable Data

Shared mutable data structures (like mutable cells and mutable variables) are the most obvious mechanisms for inter-thread communication, but they are also the ones with the most pitfalls. For example, suppose we want to define a counter that can be incremented by several threads. A straightforward approach is to define a cell and an associated incrementing procedure:

```
(define counter (cell 0))

(define increment!
  (lambda ()
    (begin (cell-set! counter (+ 1 (cell-ref counter)))
           (cell-ref counter))))
```

But then concurrent threads can interact in some nasty ways through the shared cell. For instance, what are the possible values of a call to the following MUFL! procedure?

```
(define test-increment!
  (lambda ()
    (begin
      (cell-set! counter 0)
      (colet ((a (increment!))
              (b (increment!)))
        (+ a b)))))
```

If one call to `increment!` executes to completion before the other begins, the value of this expression is `3`. But other results are possible because the execution of the two calls to `increment!` may be interleaved. The expression can return a `2` if the first `cell-ref` within each `increment!` executes before either `cell-set!`, and can return a `4` if both `cell-set!`s complete before either of the second `cell-ref`s within `increment!` are performed.

The problem here is that concurrency allows the internal operations of two different calls to `increment!` to be interleaved. We often want procedures like `increment!` to be **atomic** in the sense that they appear to execute indivisibly with respect to other computations that manipulate the same shared state. Without some means of guaranteeing atomicity, mutable data is not generally a reliable communication mechanism between threads. Even simple primitives like `cell-set!` and `cell-ref` may not be atomic, further complicating reasoning about threads that interact via shared mutable data.

11.3.2 Locks

The goal of atomicity is to constrain concurrency to avoid undesirable interleaving. Atomicity can be achieved by introducing a **lock** that guards access to a logical collection of shared mutable data elements. Typically, a lock can be "owned" by a single thread in such a way that another thread cannot acquire the lock until the owning thread determines it is safe to release it. A lock is like a lockable door to a room that can be used by only one person at a time. When no one is inside, the door is open to allow the next person to enter. But upon entering the room, an individual locks the door and only unlocks it upon leaving.

There are numerous locking schemes for concurrent languages. Here we extend MUFL! with a simple locking mechanism that has the following interface:

`(lock)`: Return a new lock that is not owned by any thread.

`(acquire!` *E*`)`: If the lock *l* computed by *E* is not owned by a thread, acquire possession of the lock. If *l* is already owned, wait until it is not owned before acquiring it. It is an error if *E* does not evaluate to a lock.

`(release!` *E*`)`: If the lock *l* computed by *E* is owned by the current thread, then release possession of the lock. It is an error if the current thread does not own the lock or if *E* does not evaluate to a lock.

`(lock?` *E*`)`: Determine whether the value of *E* is a lock.

All of these constructs can be regarded as new primitives rather than special forms.

Using a lock, we can implement an atomic version of `increment!`:

```
(define counter (cell 0))
(define counter-lock (lock))

(define increment!
  (lambda ()
    (begin (acquire! counter-lock)
           (cell-set! counter (+ 1 (cell-ref counter)))
           (let ((ans (cell-ref counter)))
             (begin (release! counter-lock)
                    ans)))))
```

Since only one thread can own `counter-lock` at any time, the operations of two calls to `increment!` running in separate threads cannot be interleaved. Thus, the new version of `increment!` guarantees that `(test-increment!)` will return `3`.

If a thread "forgets" to release a lock that it owns, or if it releases it at the wrong time, it is easy to get unintentional deadlocks. For this reason, it is wise to build some abstractions on top of locks. For example, the following `with-lock` procedure takes a lock and a thunk, and executes the thunk while holding the lock. A result is returned only when the lock is released:

```
(define with-lock
  (lambda (lock thunk)
    (begin (acquire! lock)
           (let ((ans (thunk)))
             (begin (release! lock)
                    ans)))))
```

Using `with-lock`, the atomic version of `increment!` can be implemented as:

```
(define increment!
  (lambda ()
    (with-lock counter-lock
      (lambda ()
        (begin
          (cell-set! counter (+ 1 (cell-ref counter)))
          (cell-ref counter))))))
```

If we want multiple incrementers, it is easy to bundle a lock together with each cell:

```
(define make-incrementer
  (lambda ()
    (let ((c (cell 0))
          (l (lock)))
      (lambda (msg)
        (cond ((eq? msg 'increment!)
               (with-lock l
                 (lambda ()
                   (begin
                     (cell-set! c (+ 1 (cell-ref c)))
                     (cell-ref c)))))
              ((eq? msg 'reset!)
               (with-lock l
                 (lambda ()
                   ;; Don't assume CELL-SET! is atomic
                   (cell-set! c 0))))
              (else (error unknown-message)))))))
```

Because each incrementer is equipped with its own lock, the incrementing operations of two separate incrementers can be interleaved, but the incrementing

operations of two `increment!`s to the same incrementer cannot be interleaved.

Figure 11.4 presents the rewrite rules for the lock constructs. We assume that the domain MixedExp of intermediate expressions and the domain ValueExp of value expressions have been extended with the form `(*lock*` L`)`, which represents a first-class lock value. The location in this form contains either `#f`

$$\langle A[T = \texttt{(primop lock)}], S\rangle \Rightarrow \langle A[T = \texttt{(*lock*}\ L\texttt{)}], S[L = \texttt{\#f}]\rangle, \quad \text{where } L \notin dom(S) \qquad [lock]$$

$$\langle A[T = \texttt{(primop acquire! (*lock*}\ L\texttt{))}], S[L = \texttt{\#f}]\rangle \Rightarrow \langle A[T = \texttt{\#u}], S[L = \texttt{(*thread*}\ T\texttt{)}]\rangle \qquad [acquire!]$$

$$\langle A[T = \texttt{(primop release! (*lock*}\ L\texttt{))}], S[L = \texttt{(*thread*}\ T\texttt{)}]\rangle \Rightarrow \langle A[T = \texttt{\#u}], S[L = \texttt{\#f}]\rangle \qquad [release!]$$

[*lock?-true*] and [*lock?-false*] as usual for predicates.

Figure 11.4: The semantics of locks.

(indicating that the lock is currently not owned by a thread) or a thread handle (indicating which thread owns the lock). Since a lock can only be acquired when the associated location contains `#f` and acquiring the lock mutates the location, a lock can be owned by at most one thread.

11.3.3 Channels

It is tedious to use cells and locks for all instances of communication and synchronization. Sometimes higher order abstractions are better suited for the task. One very useful abstraction is the **channel**, a conduit through which values may be communicated. A channel is a First-In/First-Out (FIFO) queue that can be accessed by multiple threads. A thread can **send!** a value to a channel (enqueue it) or **receive!** a value from a channel (dequeue the first value from the channel). Channels have the following synchronization feature built in: an attempt to receive a value from an empty channel blocks a thread until a value is sent.

Here is a procedural interface to channels (all of these can be primitives):

`(channel)`: Create and return a new channel.

`(send!` E_{chan} E_{val}`)`: Send the value computed by E_{val} over the channel computed by E_{chan} and return `#u`. It is an error if the value of E_{chan} is not a channel.

`(receive!` E_{chan}`)`: Receive and return the next value from the channel computed by E_{chan}. It is an error if the value of E_{chan} is not a channel.

`(channel?` E`)`: Determine whether the value of E is a channel.

Note that several threads may be sending to, and several threads may be receiving from, the same channel. It is even possible for a single thread to send and receive on the same channel.

Channels are an alternative to streams for programming in the "signal processing style." Figure 11.5 shows how to encode stream-like operations with channels. For example, we can use such operations to find the sum of the squares of the prime numbers between 0 and 100 as follows:

```
(let ((c1 (channel))
      (c2 (channel))
      (c3 (channel)))
  (colet ((ignore1 (gen-channel 0
                     (lambda (x) (+ x 1))
                     (lambda (x) (> x 100))
                     c1))
          (ignore2 (filter-channel prime? c1 c2))
          (ignore3 (map-channel (lambda (x) (* x x)) c2 c3))
          (ans     (accum-channel 0 + c3)))
    ans))
```

Figure 11.6 presents an operational semantics for channels. We assume that the domain MixedExp of intermediate expressions and the domain ValueExp of value expressions have been extended with the form `(*channel*` L`)`, which represents a first-class channel value. We also assume that storable values are extended to include elements

$Q \in \text{Queue} = \text{ValueExp}^*$,

value sequences that implement the queue underlying a channel.

A few notes:

- The channels described above use unbounded queues. A **bounded buffer** is a channel that has an upper size limit. A sending thread blocks if the queue implementing the buffer is at its maximum size, and will unblock if the size decreases (due to `receive!`s).

- In some concurrent languages, there is no buffer associated with a channel (i.e., the queue size is effectively zero). Instead, a channel defines a **rendezvous** point between a sending thread and a receiving thread. The first thread to reach the rendezvous point must wait for the complementary thread to arrive. When both threads are at the rendezvous point, a

```
(define eoc '*end-of-channel*)

(define (eoc? s) (sym=? s eoc))

(define (gen-channel first next done? out)
  (if (done? first)
      (send! out eoc)
      (begin (send! out first)
             (gen-channel (next first) next done? out))))

(define (map-channel proc in out)
  (let ((val (receive! in)))
    (if (eoc? val)
        (send! out eoc)
        (begin (send! out (proc val))
               (map-channel proc in out)))))

(define (filter-channel pred in out)
  (let ((val (receive! in)))
    (if (eoc? val)
        (send! out eoc)
        (begin (if (pred val) (send! out val) #u)
               (filter-channel pred in out)))))

(define (accum-channel null combine in)
  (letrec ((loop (lambda (ans)
                   (let ((val (receive! in)))
                         (if (eoc? val)
                             ans
                             (loop (combine val ans))))))))
    (loop null)))
```

Figure 11.5: Signal processing style procedures implemented with channels.

$$\langle A[T = \texttt{(primop channel)}], S\rangle \Rightarrow \langle \texttt{agenda}[T = \texttt{(*channel*}\ L\texttt{)}], S[L = [\,]]\rangle \quad [\textit{channel}]$$

$$\langle A[T = \texttt{(primop send! (*channel*}\ L\texttt{)}\ V\texttt{)}], S[L = Q]\rangle \Rightarrow \langle A[T = \texttt{\#u}], S[L = [V]\ @\ Q]\rangle \quad [\textit{send!}]$$

$$\langle A[T = \texttt{(primop receive! (*channel*}\ L\texttt{))}], S[L = Q\ @\ [V]]\rangle \Rightarrow \langle A[T = V], S[L = Q]\rangle \quad [\textit{receive!}]$$

[*channel?-true*] and [*channel?-false*] as usual for predicates.

Figure 11.6: The semantics of channels.

value is communicated from sender to receiver, after which both threads proceed.

- In several concurrent process languages (e.g., Hoare's CSP and Milner's CCS), channels are not first-class values, but are names that must match up between sending and receiving threads. There are renaming operators that permit code written in terms of one channel name to use another.
- Some languages support a kind of write-once channel called a **single-assignment cell**. Single-assignment cells have the following properties:
  - A cell has no initial value.
  - A read of the cell blocks until the cell has a value.
  - An attempt to write to a cell that has already been written is an error.

  Id's I-structures are arrays of such cells. Variables in some logic programming languages are similar to single-assignment cells.

Reading

References:

- Classic: Andrew Birrel on threads [Bir89], Melvin Conway coroutines [Con63], E. W. Dijkstra semaphores [Dij68], C. A. R. Hoare monitors [Hoa74], P. Brinch-Hansen monitors [Bri77].

- Languages: Concurrent ML ([CM90]), MULTI-LISP([Hal85]) , MULTI-SCHEME([Mil87]), ID I-structures ([ANP89]) and M-Structures ([Bar92b]), LINDA([CG89]), Concurrent Objects ([DF96])
- Denotational Semantics: Schmidt on resumption semantics [Sch86a].
- Process Algebras: [Hoa85], [Mil89]. ATP, Pi-Calculus, Pratt's Pomset model, trace semantics, Milner's overview paper on bi-simulation.
- Data Parallel: [HS86], [Ble92], [Ble90], [Sab88], Connection Machine, Connection Machine Lisp.

Chapter 12

Simple Types

Type of the wise who soar, but never roam,
True to the kindred points of heaven and home

— To a Skylark, William Wordsworth

12.1 Static Semantics

Our emphasis until this point has been the **dynamic semantics** of programming languages, which covers the meaning of programming language constructs and the run time behavior of programs. We will now shift our focus to **static semantics**, in which we describe and determine properties of programs that are independent of many details of program execution (e.g., the particular values manipulated by the program).

Programs have both dynamic and static properties:

- A **dynamic property** is one that can be determined in general only by executing the program. Such a property is determined at **run time** – i.e., when the program is executed.

- A **static property** is one that can be determined without executing the program. A static property can be determined at **compile time** – i.e., when the program is analyzed before execution.

For instance, consider the following FL program:

```
(fl (n)
  (let ((sq (lambda (x) (* x x))))
    (if (integer? n)
        (+ (sq (- n 1)) (sq (+ n 1)))
        0))
```

We assume that the program input `n` can be any s-expression. The result of the program is a dynamic property, because it cannot be known until run time what input will be entered by the user. However, there are numerous static properties of this program that can be determined at compile time:

- the free variables of the expression are `integer?`, `+`, `-`, and `*`;
- the result of the expression is an a non-negative even integer;
- the program is guaranteed to terminate.

In general, we're interested in static properties that aid in the verification, optimization, and documentation of programs. For instance, we'd like to ask the following kinds of questions about a given program:

- is this program consistent with a given specification?
- can this program possibly encounter a certain error situation?
- when the program executes, is this variable guaranteed to contain a value consistent with its declared type?
- can this program be optimized in a particular way without changing its meaning?

Of course, there are certain questions that simply cannot be answered in general. "Does this program halt?" is the most famous example of an undecidable question. Yet undecidability does not necessarily spell defeat for the goal of determining static properties of programs. There are two ways that undecidability is finessed in practice:

1. Make a conservative approximation to the desired property. E.g., for the halting problem, allow three answers:
   (a) yes, it definitely halts;
   (b) it might not halt (but I'm not sure);
   (c) no, it definitely doesn't halt.

A termination analysis is sound if it answers (a) or (b) for a program that halts and (b) or (c) for a program that doesn't halt. Of course, a trivial sound analysis answers (b) for all programs. In practice, we're interested in sound analyses that answer (a) or (c) in as many cases as possible.

2. Restrict the language to the point where it is possible to determine the property unequivocally. Such restrictions reduce the expressiveness of the language, but in return give precise static information. The ML language is an example of this approach; in order to provide static type information, it forbids many programs that would not give run-time type errors.

The notion of restricted subclasses of programs is at the heart of static semantics. We will typically start with a general language about which we can determine very few properties, and then remove features or add restrictions until we can determine the kinds of properties we're interested in. Unfortunately, the increase in our ability to reason about the programs is offset by a decrease in the expressive power of the programming language. This is a fundamental tension in programming languages: the more we can say *about* programs, the less we can say *with* them. Taking into account considerations of static semantics greatly enlarges the number of dimensions in the programming language design space. Points in the design space can often be characterized by different tradeoffs between expressive power and static properties.

12.2 An Introduction to Types

12.2.1 What is a Type?

When reading or writing code, it is common to describe expressions in terms of the kinds of values they manipulate. This is especially true when talking about procedures. For example, we typically describe `>` as a procedure that takes two integers and returns a boolean. At a more detailed level, `>` certainly performs an operation much more specific than indicated by this fuzzy description, but in many situations the fuzzy description is all we need.

For example, suppose we just want to know whether `>` would make sense as the contents of the box in the following FL expression:

```
(if (□ 1 2) (symbol three) (symbol four))
```

We can reason as follows about the contents of the box: because the box appears as the leftmost subexpression of a combination, it must be a procedure; because it is supplied with `1` and `2` as arguments, it must take two integer arguments; and

because the result of the application is used as the test in an `if` expression, the procedure in the box must return a boolean. Thus, `>` *would* make sense as the contents of the box. But more important, *any* value satisfying the description "a procedure of two integers returning a boolean" would be viable as the contents of the box.

This simple example underscores the fact that it is not necessary to know precise values in order to perform computations with a program. The reasoning used above was based on classes of values rather than on particular values. Classes of values are known as **types**.

There are many ways to think about types. In its most general form a type is just a description of a value. From another perspective, a type is an approximation to a value, or a value with partial information. For example, the type "integer" is an approximation to the integers `1` and `2`, while the type "procedures from two integers to a boolean" is an approximation to `>` and `=`. From yet another point of view, types are arbitrary sets. Some examples of such sets include the integers, the natural numbers less than 5, the prime integers, and procedures that halt on the input `3`.

The last example (procedures that halt on the input `3`) shows that types we might like to describe may not even be computable. In other cases (e.g., the prime numbers), types might be exceedingly difficult to reason about. It is often necessary to restrict these very general notions of type to ones that are less general, but simpler to reason about. However, if we hope to assign types to all expressions in a language, such simplification entails restrictions on the kinds of programs we can write. This is an example of the general tradeoff between expressive power and determination of static properties introduced above. In our study of types, we shall consider several points in the design space that handle this tradeoff in different ways.

12.2.2 Dimensions of Types

Types are not a monolithic feature that are either present or absent in a language. Rather, there is a rich diversity of ways that types may appear in a programming language, and almost all languages have some sort of type system. (Examples of completely typeless languages include the untyped lambda calculus and most assembly-level languages.) Here we shall examine three dimensions along which type systems may vary.

12.2.2.1 Dynamic vs. Static

In the programming languages we have studied so far, values have types associated with them. FL!, for instance, divides the class of (non-error) values into six types: unit, integers, booleans, symbols, procedures, and references. The operational and denotational semantics for these languages make use of the type information to determine the meanings of programs. For example, whether or not the expression `(primop +` E_1 E_2`)` denotes an integer depends on the types of values found for E_1 and E_2. Both operational and denotational semantics provide some method for checking the types of these subparts in order to determine the value of the whole. Languages in which values carry types with them and type checks are made at run time are said to be **dynamically typed**. Examples of dynamically typed languages include LISP, SMALLTALK, and APL.

An alternative to dealing with types at run time is to statically analyze a program at compile time to determine if type information is consistent. Here, types are associated with expressions in the language rather than with run time values. A program that can be assigned a type in this approach is said to be **well-typed**. Programs that are well-typed are guaranteed not to contain certain classes of errors (e.g., a procedure call with the wrong number of arguments). A program that cannot be described with a type is said to contain a **type error**. The set of well-typed programs is a subset of all of the programs that are syntactically well formed. Languages in which types are associated with expressions and are computed at compile time are said to be **statically typed**. Examples of statically typed languages include JAVA, PASCAL, ADA, ML, and HASKELL. Practical statically typed languages are equipped with a **type checker** that can automatically verify that programs are well-typed.

The choice between dynamic and static typing has been the source of a great debate in the programming language community. Adherents of static typing offer the following arguments in favor of static types:

- *Safety:* Type checking reduces the class of possible errors that can occur at run time. In certain situations it is extremely desirable to catch as many errors as possible before the program is run (e.g., programs to control a space shuttle or nuclear power plant).

- *Efficiency:* Statically typed programs can be more efficient than dynamically typed ones. In implementation terms, dynamic typing implies space and time costs at run time. Space is necessary to encode the type of a value at the bit level. Since types must be checked when performing certain primitive operations (e.g., binary integer addition can only be applied when both operands are integers), dynamic typing has a time cost

as well. In statically typed languages, most values do not require any run time storage for type representations. In addition, the compile time type checks eliminate the need to check types at run time.

- *Documentation* Static types provide documentation about the program that can facilitate reasoning about the program, both by humans and by other programs (e.g. compilers). Such information is especically valuable in large programs.

- *Program Development:* Static types help programmers catch errors in their programs before running them and help programmers make representation changes. For example, suppose a programmer decides to change the interface of a procedure in a large program. The type checker helps the programmer by finding all the places in the program where there is a mismatch between the old and new interfaces.

Proponents of dynamic typing counter that the restrictions placed on a language in order to make it type checkable force the programmer into a straight jacket of reduced expressive power. They argue that in many statically typed languages (e.g., JAVA and PASCAL), types mainly serve to make the language easier to implement, not easier to write programs in. Furthermore, they discount the importance of finding type errors at compile time; they argue that the hard-to-find errors that occur in practice are logical errors, not type errors. Finding such errors requires testing programs with extensive test suites that would also find type errors.

12.2.3 Explicit vs. Implicit

Another dimension on which type systems vary is the extent to which they force a programmer to declare explicit types. Although some dynamically typed languages require some form of type declaration (e.g., array variables in BASIC), dynamically typed languages typically have no explicit types. The converse is true in static typing, where explicit types are the norm. In traditional statically typed languages (e.g., PASCAL, , and JAVA) it is necessary to explicitly declare the types of all variables, formal parameters, procedure return values, and data structure components. However, some recent languages (e.g., FX, ML, MIRANDA, and HASKELL) achieve static typing without explicit type declarations via a method called **type reconstruction** or **type inference**. We shall study type reconstruction in Chapter 14.

One argument for explicit types is that the types serve as important documentation in a program and therefore make programs easier to read and write.

Often, however, explicit types make programs easier for compilers to read, not easier for humans to read; and explicit types are generally cumbersome for the program writer as well. Implicitly typed programming languages thus have clear advantages in terms of readability and writability. Unfortunately, certain restrictions must be placed on a language in order to make type reconstruction possible. This means that some programs that can be written with explicit types cannot be written with implicit ones. A compromise between the two approaches, adopted by ML and HASKELL, is to make most types implicit by default, but to allow explicit declarations in situations where types cannot be reconstructed.

12.2.4 Simple vs. Expressive

A third dimension along which typed languages can vary is the expressiveness of their type systems. Languages with very simple type systems facilitate type checking and type reconstruction, but generally severely restrict the kinds of programs that can be written. For example, in PASCAL,[1] the length of an array is a part of its type; this makes it impossible to write a sorting procedure that can accept an array of any length. In languages with **polymorphic** types, it is possible to have procedures that are parameterized over the types of their inputs. This makes it possible to express programs more naturally, but at the cost of making the type system more complex. We will study polymorphism in Chapter 13.

12.3 FL/X: A Language with Monomorphic Types

12.3.1 FL/X

We begin our exploration of types by studying FL/X, a statically typed dialect of FL with eXplicit types. FL/X is a **monomorphic** language, which means that each legal expression is described by exactly one type. In a monomorphic language, procedures cannot be parameterized over the types of their arguments. For example, a procedure that reverses lists of integers cannot be used to reverse lists of strings, even though the reversal procedure never needs to examine the components of the list.

Despite this lack of expressiveness, a monomorphic language is worth studying because (1) it simplifies the discussion of many type issues and (2) a number

[1] At least in pre-ANSI PASCAL.

of popular languages (e.g., FORTRAN, PASCAL, and C) are monomorphic.[2] As evidenced by the success of these languages, monomorphic languages can still be very useful in practice. As we shall see, monomorphic languages can even support features like higher-order procedures and recursive types.

The grammar for FL/X is presented in Figure 12.1. It is similar to the FL grammar, but there are some important differences, which we discuss in detail.

There is a new syntactic domain Type that is used to specify the types of FL/X expressions. An FL/X type has one of two forms:

1. a **base type** specifies one of the built-in types of primitive data:

   - `unit`, the type of the one-point set {`#u`};
   - `bool`, the type of the two-point set {`#t`, `#f`};
   - `int`, the type of integers; and
   - `sym`, the type of symbols.

2. an **arrow type** of the form `(->` (*T*$_{arg_1}$ ... *T*$_{arg_n}$)` ` *T*$_{result}$`)` specifies the type of an n-argument procedure that takes arguments of type T_{arg_1} through T_{arg_n} and returns a result of type T_{result}. For example, an incrementing procedure on integers has type `(-> (int) int)`, an addition procedure on integers has type `(-> (int int) int)`, and a less-than procedure on integers has type `(-> (int int) bool)`.

   Arrow types can be nested, in which case they describe higher-order procedures. For example:

   - a procedure that returns either an incrementing or decrementing procedure based on a boolean argument has type

     ```
     (-> (bool) (-> (int) int))
     ```

   - a procedure that takes an integer predicate and determines if any numbers in the range [1 . . . 10] satisfy this predicate has type

     ```
     (-> ((-> (int) bool)) bool)
     ```

   - a procedure that approximates the derivative of an integer function has type

     ```
     (-> ((-> (int) int)) (-> (int) int))
     ```

[2] These languages provide ad hoc overloading and type casting mechanisms that make it possible to go beyond monomorphism in limited ways. However, because they provide no principled mechanisms for polymorphism, we consider them to be monomorphic.

$P \in$ Program
$D \in$ Def
$E \in$ Exp
$T \in$ Type
$I \in$ Identifier
$B \in$ Boollit = {`#t`, `#f`}
$N \in$ Intlit = {..., `-2`, `-1`, `0`, `1`, `2`, ...}
$L \in$ Lit
$O \in$ Primop = Usual FL primitives except list ops and predicates.

P ::= `(flx ((`I_{formal} T`)*)` E_{body} $D_{definitions}$`*)` [Program]

D ::= `(define` I_{name} T_{type} E_{defn}`)` [Value Definition]
| `(define-type` I_{name} T_{defn}`)` [Type Definition]

E ::= L [Literal]
| I [Variable Reference]
| `(if` E_{test} $E_{consequent}$ $E_{alternate}$`)` [Branch]
| `(lambda ((`I_{formal} T`)*)` E_{body}`)` [Abstraction]
| `(`E_{rator} E_{rand}`*)` [Application]
| `(let ((`I_{name} E_{defn}`)*)` E_{body}`)` [Local Value Binding]
| `(letrec ((`I_{name} T_{type} E_{defn}`)*)` E_{body}`)` [Local Value Recursion]
| `(primop` O_{name} E_{arg}`*)` [Primitive Application]
| `(error` $I_{message}$ T`)` [Error]
| `(the` T E`)` [Type Ascription]
| `(tlet ((`I_{name} T_{defn}`)*)` E_{body}`)` [Local Type Binding]

L ::= `#u` [Unit Literal]
| B [Boolean Literal]
| N [Integer Literal]
| `(symbol` I`)` [Symbol Literal]

T ::= `unit` | `bool` | `int` | `sym` [Base Types]
| `(->` `(`T`*)` T_{body}`)` [Arrow Type]

Figure 12.1: Grammar for FL/X, a monomorphic, explicitly typed language.

Because `->` is used to combine simpler types into more complex types, it is known as a **type constructor**. It is the first of several type constructors that we will encounter in FL/X. We will see several others in Section 12.4.

The prefix form of FL/X arrow types may seem unusual to those accustomed to the infix type notation that is standard in the types literature and in languages like SML and HASKELL. The following table shows examples of the two notations side by side:

FL/X **types**	SML **types**
`(-> (bool) (-> (int) int))`	`bool -> (int -> int)`
`(-> ((-> (int) bool)) bool)`	`(int -> bool) -> bool`
`(-> ((-> (int) int)) (-> (int) int))`	`(int -> int) -> (int -> int)`
`(-> ((-> (int int) bool)) (-> (int int) int))`	`(int * int -> bool) -> int * int -> int`

Some FL/X expressions — literals, variable references, conditionals, primitive applications, and `let` — are unchanged from FL. But other expressions have been extended with type annotations that will be used to determine the types of the expressions:

- In abstractions, parameters are specified by a sequence of bindings of the form `(`*I* *T*`)` that specify both the name and the type of each formal parameter. For example, an averaging abstraction can be written as
```
(lambda ((a int) (b int))
  (/ (+ a b) 2)
```
and an abstraction that chooses an incrementing or decrementing procedure based on a boolean argument can be written as
```
(lambda ((b bool))
  (if b
      (lambda ((x int)) (+ x 1))
      (lambda ((x int)) (- x 1)))).
```
- Unlike `let` expression bindings, each binding in a `letrec` expression has a type in addition to the name and definition expression. For example, the following `letrec` expression introduces a `summer` procedure that sums all the integer values in the range `lo` to `hi` that satisfy a predicate `f`. The `letrec` syntax requires that the type of `summer` be written down explicitly.
```
(letrec
  ((summer (-> ((-> (int) bool) int int) int)
     (lambda ((f (-> (int) bool)) (lo int) (hi int))
       (if (> lo hi)
           0
           (+ (if (f lo) lo 0)
              (summer f (+ lo 1) hi))))))
  (summmer (lambda (x) (= (rem x 3) 0)) 1 100))
```

- FL/X requires the programmer to specify the type of an `error` construct explicitly. For example, consider the following higher-order procedure:

```
(lambda ((n int))
  (if (< n 0)
      (error negative (-> (int) bool))
      (lambda ((d int))
        (if (= d 0)
            (error zero bool)
            (= (rem n d) 0)))))
```

  Although `error` expressions never return a value, the type annotations specify that the first `error` expression should be treated as if it returns a procedure with type `(-> (int) bool)` and the second `error` expression should be treated as if it returns a boolean value. These type declarations allow the `error` expressions to have the same type as the other arms of their corresponding `if` expressions.

You may wonder why FL/X has type annotations for some expressions but not others. For instance:

- Why are types required in `letrec` bindings but not `let` bindings?
- Why do abstractions require specifying parameter types but not the type of the returned value? After all, procedure and method declarations in languages like C, JAVA, and PASCAL require explicit return types.
- Why are types required in `error` expressions?

The answer is that type annotations in FL/X were chosen to be the minimal annotations that allow the type of any expression in a program to be determined without "guessing" the types of any expressions. We will formalize this notion when we study the type checking of FL/X expressions in Section 12.3.2.

There are several other differences between FL/X and FL:

- For simplicity, the version of FL/X we study here has no data structures (unlike FL, which has pairs and lists). We will study typed data later in Section 12.4.
- FL/X supports fewer primitive operations than FL. In particular, because the type of every FL/X expression is known at type checking time, there is no need for type predicates like `boolean?`, `integer?`, and `procedure?`. Because we are ignoring lists for now, the variant of FL/X we study here does not support any list operations either.

- FL/X has a new type ascription construct `(the` *T E*`)` that asserts that expression *E* has type *T*. In other languages, type ascription is often written via a notation like *E* `:` *T*. The expression `(the` *T E*`)` returns the value of *E*, so it can be used wherever *E* is used. For example, it can be used to explicitly declare the return type of a procedure:

```
(lambda ((b bool) (x int))
  (the int (if b (+ x 1) (* x 2))))
```

  The `the` construct is not strictly necessary, but it is handy for documenting the types of expressions.[3] Assertions made with `the` are automatically verified by a type checker. For example:

```
(+ 1 (the int (* 2 3))) ; Type Checks; Value = 7
(+ 1 (the bool (* 2 3)))  ; Doesn't type check: * returns int
```

- Later we will see that types in FL/X can become large and cumbersome. The `tlet` construct improves the readability and writability of types by allowing type expressions to be abbreviated by names. The abbreviations are local to the body expression of the `tlet`. For example:

```
(tlet ((intfun (-> (int) int)))
  (tlet ((intfun-transformer (-> (intfun) intfun)))
    (the intfun-transformer
      (lambda ((f intfun))
        (lambda ((x int))
          (* 2 (f (+ x 1))))))))
```

  We will assume that there is a single namespace for types and values. So the first of the following expressions is reasonable, but the second and third are nonsensical:

```
;; Reasonable expression
(lambda ((x int))
  (tlet ((z bool))
    (lambda ((y z))
      x)))

;; nonsensical expression
(lambda ((x int))
  (tlet ((x bool))
    (lambda ((y x))
      x))) ; This X bound by TLET
```

[3] Unlike, for example, casts in C, FL/X's `the` is *not* a coercion operator that can be used to create type loopholes.

```
;; nonsensical expression
(tlet ((x bool))
  (lambda ((x int))
    (lambda ((y x)) ; This X bound by LAMBDA
      x)))
```

It is possible to put types and values in different namespaces, but this complicates the definitions of syntactic operations (finding free variables, performing substitution) that we will use later.

- In the `program` sugar of FL/X (see Figure 12.2), `define` declarations (which desugar into a `letrec`) must include an explicit type. There is also a new `define-type` declaration that is sugar for a global `tlet`. The desugaring for `program` assumes that all `define-type` forms come before all `define` forms. This is not required, but it is always possible to "bubble-up" the `define-type` forms to the top of the program without changing its meaning (assuming the names used in `define`s and `define-type`s are disjoint).

  Unlike FL, FL/X does not have any syntactic sugar for expressions. Multi-argument abstractions, `let`, and `letrec` do not desugar to simpler FL/X forms but are considered primitives. The reason for this is that such desugarings would not preserve expressions types. In FL/X, a two argument addition procedure, whose type is `(-> (int int) int)`, is not equivalent to the curried form of this procedure, which has type `(-> (int) (-> (int) int))`. An FL/X `let` construct cannot desugar into an application of a multi-argument abstraction because the parameter types necessary for the abstraction are not manifest.

  FL/X could easily be extended to support other sugared expressions from FL, such as `and`, `or`, `cond`, but we omit these here to avoid clutter.

12.3.2 FL/X Type Checking

12.3.2.1 Introduction to Type Checking

In a statically typed language, a program phrase is said to be **well-typed** if it is possible to assign a type to the phrase based on a process known as **type checking**. This process is typically expressed by a collection of formal rules and a reasoning system that uses these rules. A phrase is said to be **ill-typed** if it is not possible to assign it a type. Only well-typed phrases are considered legal phrases of the language.

```
D_prog [[(flx ((I T)*) E_body
         (define-type I_t1 T_t1) ... (define-type I_tk T_tk)
         (define I_v1 T_v1 E_v1) ... (define I_vn T_vn E_vn))]]
= (flx ((I T)*)
          (let ((not? (lambda ((x bool)) (primop not? x))))
          (and? (lambda ((x bool) (y bool)) (primop and? x y)))
            ⋮ ; Similar for or? and bool=?
          (+ (lambda ((x int) (y int)) (primop + x y))))
            ⋮ ; Similar for -, *, /, rem, <, <=, =, /=, >=, >
          (sym=? (lambda ((x sym) (y sym)) (primop sym=? x y)))
          (unit #u)
          (true #t)
          (false #f)
          )
       (tlet ((I_t1 T_t1))
             ⋮
          (tlet ((I_tk T_tk))
            (letrec ((I_v1 T_v1 E_v1)
                        ⋮
                     (I_vn T_vn E_vn))
              E_body)))
```

Figure 12.2: Desugaring for FL/X syntactic sugar.

Type checking is similar to evaluation, except that rather than manipulating the run-time values associated with expressions, it manipulates the static types associated with the expressions. Recall that it is possible to view types as approximations to values. From this perspective, a type checker evaluates the program with approximations rather than actual values. The notion that a program can be "run" in a way that is guaranteed to terminate on a finite set of value approximations is the basis of a style of program analysis known as **abstract interpretation**.

As a simple example of the kind of reasoning used in type checking, consider the type analysis of the following FL/X abstraction:

```
(lambda ((b bool) (x int) (f (-> (int int) int)))
  (if b x (f 0 1)))
```

The type annotations on the parameters indicate that `b` is assumed to be a boolean, `x` is assumed to be an integer, and `f` is assumed to be a procedure that maps two integer arguments to an integer result. Based on the assumption for `f`, the type of `(f 0 1)` is `int`, because applying a procedure of type `(-> (int int) int)` to two integers yields an integer. Based on this conclusion and the assumptions for `b` and `x`, the body expression `(if b x (f 0 1))` is well-typed, because the test subexpression has type `bool`, and the two branches both have the same type, `int`. The type of the `if` expression is `int`, because that is the type of the value returned by the expression for any values of `b`, `x`, and `f` satisfying the type assumptions. Since the abstraction takes three parameters, a `bool`, an `int`, and a procedure of type `(-> (int int) int)`, and it returns an `int`, the abstraction has the arrow type `(-> (bool int (-> (int int) int)) int)`.

If we changed the body of the example to `(if x x (f 0 1))`, the `if` expression would not be well-typed because the test subexpression does not have type `bool`. Similarly, the body would not be well-typed if it were `(if b b (f 0 1))`, because then the two conditional branches would have incompatible types: `bool` and `int`.[4] Even the expression `(if #t b (f 0 1))` is not considered to be well-typed, even though it is guaranteed to return a boolean value when executed. Why? The type checker only manipulates approximations to values. It does not "know" that the test expression is the constant true value. All it "knows" is that the test expression is a boolean, and so it cannnot determine which branch

[4]There are sophisticated type systems in which `(if b b (f 0 1))` would be considered well-typed, with a so-called **union type** that is *either* `bool` *or* `int`. In order to guarantee type soundness (see Section 12.3.3), such systems must constrain the ways in which a value with union type may be manipulated. In this presentation, we focus on simpler type systems that do not allow union types.

is taken.[5]

From the above examples, it is clear that just as the value of an expression is determined from the values of its subexpressions, so too is the type of the expression determined from the type of its subexpressions. However, the actual rules for determining the type of the whole from the type of the parts may be very different from the rules for determining the value of the whole from the value of the parts. For instance:

- an evaluator only evaluates *one* branch of a conditional, but a type checker checks *both* branches of a conditional.
- an evaluator does not evaluate the body of a procedure until it is applied to arguments, but a type checker checks the body of an abstraction regardless of whether or not it is applied.
- an evaluator associates the actual arguments with the formal parameters when applying a procedure to arguments, but a type checker simply checks that the types of the actual arguments are compatible with the argument types expected by the procedure.

12.3.2.2 Type Environments

Just as expressions are evaluated with respect to a dynamic **value environment** that associates free identifiers with their run-time values, they are type checked with respect to a static **type environment** that associates free identifiers with their types. Type environments are partial functions from identifiers to types:

$$A \quad \in \quad \textit{Type-Environment} \quad = \quad \text{Identifier} \rightharpoonup \text{Type}$$

If A is a type environment and $I \in dom(A)$, then the notation $A(I)$ designates the type assigned to I in A.

The association of a type T with a name I is known as a **type assignment**, which we will write using the notation $I\colon T$ and pronounce as "I has type T." We will write type environments as sets of type assignments whose names are pairwise disjoint. For instance, {} is the empty type environment, and the type environment used to check the abstraction body `(if b x (f 0 1))` in the above example is:

$$A_1 = \{\texttt{b:bool, x:int, f:(-> (int int) int)}\}$$

[5] Again, in some more sophisticated type systems, `(if #t b (f 0 1))` would be considered well-typed with type `bool`.

The body of an FL/X program is type checked with respect to a **standard type environment** A_{std} (Figure 12.3) that assigns types to the global names that may be used within the body. For example, A_{std}(+) = `(-> (int int) int)` and A_{std}(<) = `(-> (int int) bool)`.

```
{ unit:unit,
  true:bool,
  false:bool,
  not?:(-> (bool) bool),
  and?:(-> (bool bool) bool),
  or?:(-> (bool bool) bool),
  bool=?:(-> (bool bool) bool),
  +:(-> (int int) int),
  -:(-> (int int) int),
  *:(-> (int int) int),
  /:(-> (int int) int),
  rem:(-> (int int) int),
  <:(-> (int int) bool),
  <=:(-> (int int) bool),
  =:(-> (int int) bool),
  /=:(-> (int int) bool),
  >=:(-> (int int) bool),
  >:(-> (int int) bool),
  sym=?:(-> (sym sym) bool) }
```

Figure 12.3: Standard type environment A_{std} for FL/X.

As with value environments, it is often necessary to extend a type environment with additional bindings. We use the notation

$$A[I_1 : T_1\,,\ \ldots,\ I_n : T_n]$$

to indicate the type environment that results from extending A with the given type assignments. The identifiers I_i must be distinct, and the extensions override any assignments that A may already have for these identifiers. For example, suppose that A_2 = A_1[`b:sym,t:bool`]. Then *dom*(A_2) = {b, f, x, z} and A_2(`b`) = `sym`, A_2(`f`) = `(-> (int int) int)`, A_2(`x`) = `int`, and A_2(`t`) = `bool`.

12.3.2.3 Type Rules for FL/X

We now describe a formal process by which the types of FL/X expressions can be determined. The assertion that an expression E has type T with respect to type environment A is known as a **type judgment** and is written as

$$A \vdash E\ :\ T$$

This is pronounced "E has type T in A" or, more loosely, "A proves that E has type T." When such an assertion is true, we say that the type judgment is **valid**. If $A \vdash E : T$ is valid, we say that E is **well-typed with respect to** A. Otherwise, E is **ill-typed with respect to** A. If the type environment (typically A_{std}) is understood from context, we just say that E is **well-typed** or **ill-typed**. When the type environment is omitted from a type judgment, as in $\vdash E : T$, this asserts that E has type T in every environment.

Valid type judgments can be determined via type rules that have a form similar to the rules we introduced for operational semantics in Chapter 3. Each type rule has the form

$$\frac{\mathit{Premise}_1; \ldots; \mathit{Premise}_n}{\mathit{Conclusion}} \quad [\mathit{name\text{-}of\text{-}rule}]$$

where *Conclusion* and each $\mathit{Premise}_i$ are type judgments. If all of the premises of a rule are valid, then the type judgment in the conclusion of the rule is valid.

The type rules for FL/X are presented in Figure 12.4. The [*unit*], [*bool*], [*int*], [*sym*], and [*error*] rules are axioms that are independent of the type environment. The other axiom, [*var*], says that the type of an identifier is looked up in the type environment.

The [*if*] rule requires that (1) the test expression denotes a boolean and (2) the two branches have the same type. If these requirements are met, the type of the `if` expression is the type of the branches. The constraint that the two branch types and return type must all be the same is specified by using the same type metavariable, T, for all three types.

As in the operational semantics rules, type rules are really rule schemas in which every metavariable can be instantiated by any element of the domain ranged over by the metavariable. So [*if*] stands for an infinite number of rules in which A can be any type environment, T can be any type, and E_{test}, E_{con}, and E_{alt} can be any expressions. Many of these instantiations may not make sense at first glance. For example, here is one instantiation of the `if` rule:

$$\frac{\{\} \vdash \texttt{1 : bool} \quad ; \quad \{\} \vdash \texttt{2 : bool} \quad ; \quad \{\} \vdash \texttt{3 : bool}}{\{\} \vdash \texttt{(if 1 2 3) : bool}}$$

Certainly we should not be able to prove that `(if 1 2 3)` has type `bool`! But the rule doesn't say that `(if 1 2 3)` has type `bool`. Rather, it says that `(if 1 2 3)` *would* have type `bool` *if* the integers `1`, `2`, and `3` all had type `bool`. But it is impossible to prove these false premises, and so the false conclusion will never be declared to be a valid judgment by the type system.

The [`->`-*intro*] and [`->`-*elim*] rules are the rules for abstractions and applications, respectively. The rule names emphasize that abstractions are the source

$$\vdash \texttt{\#u} : \texttt{unit} \quad [unit] \qquad \vdash N : \texttt{int} \quad [int] \qquad \vdash B : \texttt{bool} \quad [bool]$$

$$\vdash \texttt{(symbol } I\texttt{)} : \texttt{sym} \quad [sym] \qquad \vdash \texttt{(error } I\ T\texttt{)} : T \quad [error]$$

$$A \vdash I : A(I), \text{ where } I \in dom(A) \qquad [var]$$

$$\frac{A \vdash E_{test} : \texttt{bool} \;\; ; \;\; A \vdash E_{con} : T \;\; ; \;\; A \vdash E_{alt} : T}{A \vdash \texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)} : T} \qquad [if]$$

$$\frac{A[I_1 : T_1, \ \ldots, \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{) } \ldots \texttt{ (}I_n\ T_n\texttt{)) } E_{body}\texttt{)} : \texttt{(->} \ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\texttt{->}\text{-}intro]$$

$$\frac{A \vdash E_{rator} : \texttt{(->} \ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{result}\texttt{)} \quad \forall_{i=1}^{n} . \ A \vdash E_i : T_i}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)} : T_{result}} \qquad [\texttt{->}\text{-}elim]$$

$$\frac{\forall_{i=1}^{n} . \ A \vdash E_i : T_i \quad A[I_1 : T_1, \ \ldots, \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{) } \ldots \texttt{ (}I_n\ E_n\texttt{)) } E_{body}\texttt{)} : T_{body}} \qquad [let]$$

$$\frac{\forall_{i=1}^{n} . \ A' \vdash E_i : T_i \quad A' \vdash E_{body} : T_{body}}{A \vdash \texttt{(letrec ((}I_1\ T_1\ E_1\texttt{) } \ldots \texttt{ (}I_n\ T_n\ E_n\texttt{)) } E_{body}\texttt{)} : T_{body}} \qquad [letrec]$$

where $A' = A[I_1 : T_1, \ \ldots, \ I_n : T_n]$

$$\frac{A_{std} \vdash O_{name} : \texttt{(->} \ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{result}\texttt{)} \quad \forall_{i=1}^{n} . \ A \vdash E_i : T_i}{A \vdash \texttt{(primop } O_{name}\ E_1\ \ldots\ E_n\texttt{)} : T_{result}} \qquad [primop]$$

$$\frac{A \vdash E : T}{A \vdash \texttt{(the } T\ E\texttt{)} : T} \qquad [the]$$

$$\frac{A \vdash ([T_i/I_i]_{i=1}^{n}) E_{body} : T_{body}}{A \vdash \texttt{(tlet ((}I_1\ T_1\texttt{) } \ldots \texttt{ (}I_n\ T_n\texttt{)) } E_{body}\texttt{)} : T_{body}} \qquad [tlet]$$

$$\frac{A[I_1 : T_1, \ \ldots, \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(flx ((}I_1\ T_1\texttt{) } \ldots \texttt{ (}I_n\ T_n\texttt{)) } E_{body}\texttt{)} : \texttt{(->} \ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [prog]$$

Figure 12.4: Typing rules for FL/X.

expressions that produce values of arrow type and that applications are the sink expressions that use values of arrow type. In our study of typed data in Section 12.4, we shall see many other examples of introduction and elimination rules. In the [`->`-*intro*] rule, the type of an abstraction is an arrow type that maps the explicitly declared parameter types to the type of the body, where the body type is determined relative to an extended environment that includes type assignments for the parameters. The [`->`-*elim*] rule requires that the operator of an application be an arrow type whose number of parameters is the same as the number of supplied operands and whose parameter types are the same as the corresponding operand types. In this case, the type of the application is the result type of the operator type.

The [*let*] and [*letrec*] rules are similar. Both type check a body expression with respect to the given type environment A extended with type assignments for the named definition expressions E_i in the bindings. The difference is that `let` definitions are not in the scope of the bindings, and so can be type checked relative to A. However, `letrec` definitions *are* in the scope of the bindings, and so must be type checked relative to an environment A' that extends A with type assignments for the bindings. Since the definition types in a `let` can be determined from the supplied type environment A, there is no need for types of the definitions to be explicitly declared. But in the `letrec` case, determining the extended type environment A' in general requires finding a fixed point over type environments. FL/X requires the programmer to explicitly declare the types of `letrec` definitions so that the type checker does not need to compute fixed points.

The [*primop*] rule treats primitive operators as if they have arrow types determined by the standard type environment, A_{std}. This allows the type checker to handle primitive applications via what is essentially a specialized version of [`->`-*elim*].

The [*tlet*] rule type checks the result of substituting the types $T_1, \ldots, T_n$ for the identifiers $I_1, \ldots, I_n$ in the body expression E_{body}. All the rules except for `tlet` are **purely structural** in the sense that the premise judgments involve subexpressions of the expression that appears in the conclusion judgment. When rules are purely structural, it is easy to show by structural induction that the type checking process will terminate. The initial expression being type checked is finite, and in any rule each premise subexpression is necessarily strictly smaller than the conclusion expression, so the recursion process must eventually bottom out at the axioms. But `tlet` is *not* structural, because the substituted body expression is not a subexpresion of the original `tlet` expression. With non-structural rules like `tlet`, care must be taken that each of the premise expressions is strictly smaller than the conclusion expression by an appropriate

metric. In the case of `tlet`, such a metric is expression height, which measures the height of an expression tree ignoring the height of any type nodes.

The [*prog*] rule is the type checking rule for a top-level program. Since a program maps input values to an output value, it has an arrow type. Indeed, from a type-checking perspective, the [*prog*] rule is identical to the [`->`-*intro*] rule. Although FL/X allows program parameters of any type, in practice the parameter types are often restricted. For example, JAVA programs have a single program parameter that must be an array of strings. In our examples, we will assume that the parameters to FL/X programs correspond to values that can be expressed with s-expressions. In particular, we will assume that parameter types *cannot* contain arrow types.

12.3.3 FL/X Dynamic Semantics and Type Soundness

Intuitively, types specify a static property of an FL/X expression, not a dynamic property. We formalize this intuition by defining the dynamic semantics of FL/X via a transformation that erases the types of FL/X. As the target of this transformation, we introduce a variant of FL that we will call FL*. The FL* language has the same syntax as FL, but its multiple parameter abstractions, multiple argument applications, and multiple binding `letrec`s are treated as indecomposable constructs rather than as syntactic sugar. (The multiple binding `let`, on the other hand, desugars into an application of a manifest abstraction.) The essence of the operational semantics of CBN FL* is presented in Figure 12.5. Rewrite rules for FL* constructs not in the figure are the same as those for FL.

$$\texttt{((lambda (}I_1\ \ldots I_n\texttt{)}\ E_{body}\texttt{)}\ E_1\ \ldots E_n\texttt{)} \Rightarrow ([E_i/I_i]_{i=1}^{n})E_{body} \qquad \text{[FL*-}apply\text{]}$$

$$\frac{E_{rator} \Rightarrow E_{rator}{}'}{\texttt{(}E_{rator}\ E_1\ \ldots E_n\texttt{)} \Rightarrow \texttt{(}E_{rator}{}'\ E_1\ \ldots E_n\texttt{)}} \qquad \text{[FL*-}rator\text{]}$$

$$\begin{array}{c}\texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} \\ \Rightarrow([\texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\texttt{(}I_n\ E_n\texttt{))}\ E_i\texttt{)}/I_i]_{i=1}^{n})E_{body}\end{array} \qquad \text{[FL*-}letrec\text{]}$$

Figure 12.5: Operational rules distinguishing CBN FL* from CBN FL. Rules for all other FL* constructs are the same as those for FL.

The types of an FL/X expression E can be erased via type erasure (written $\lceil E \rceil$) to yield an FL* expression (see Figure 12.6). We define the meaning of

an FL/X expression E as the meaning of its type erasure $\lceil E \rceil$. For example, suppose that E_{test} is:

```
(let ((f (lambda ((b bool) (x int))
           (if b x (primop + x 1))))
      (y (primop * 3 4)))
  (f (primop = 10 y) y))
```

Then $\lceil E_{test} \rceil$ is:

```
((lambda (f y) (f (primop = 10 y) y))
 (lambda (b x) (if b x (primop + x 1)))
 (primop * 3 4))
```

Since the latter expression reduces to 13 in FL*, the meaning of the FL/X expression E_{test} is 13.

We will say that an FL* expression **is a type error** if it is stuck under the operational rewrite rules for some reason other than (1) division or remainder by zero or (2) an explicit `error` construct. For instance, the following FL* expressions are type errors:

```
(primop + 1 true)  ; wrong argument type to +
(primop + 1)  ; too few arguments to +
(primop + 1 2 3)  ; too many arguments to +
(if 1 2 3)  ; non-boolean if test
(1 2 3)  ; application of non-abstraction
((lambda (x y) x) 1)  ; too few arguments in application
((lambda (x y) x) 1 2 3)  ; too many arguments in application
```

We will say that an FL* expression **has a type error** if it can be operationally rewritten to an expression that is a type error.

The advantage of types is that they guarantee an expression has no type errors. This is captured in the following type soundness result for FL/X:

Theorem 1 (Type Soundness of FL/X**)** *If E is a well-typed* FL/X *expression, then $\lceil E \rceil$ does not have a type error.*

The above theorem is the consequence of the following two theorems:

Theorem 2 (Progress for FL/X**)** *If E is a well-typed* FL/X *expression, then it is not stuck – i.e., either it is a normal form or it can be rewritten via the operational rules to another* FL/X *expression.*

$\lceil . \rceil$: $\text{Exp}_{FL/X} \rightarrow \text{Exp}_{FL*}$

$\lceil L \rceil = L$

$\lceil I \rceil = I$

$\lceil (\texttt{if}\ E_{test}\ E_{consequent}\ E_{alternate}) \rceil = (\texttt{if}\ \lceil E_{test} \rceil\ \lceil E_{consequent} \rceil\ \lceil E_{alternate} \rceil)$

$\lceil (\texttt{lambda}\ ((I_1\ T_1)\ \ldots\ (I_n\ T_n))\ E_{body}) \rceil = (\texttt{lambda}\ (I_1\ \ldots\ I_n)\ \lceil E_{body} \rceil)$

$\lceil (E_{rator}\ E_{rand1}\ \ldots\ E_{randn}) \rceil = (\lceil E_{rator} \rceil\ \lceil E_{rand1} \rceil\ \ldots\ \lceil E_{randn} \rceil)$

$\lceil (\texttt{let}\ ((I_1\ E_1)\ \ldots\ (I_n\ E_n))\ E_{body}) \rceil =$
$\quad ((\texttt{lambda}\ (I_1\ \ldots\ I_n)\ \lceil E_{body} \rceil)\ \lceil E_1 \rceil\ \ldots\ \lceil E_n \rceil)$

$\lceil (\texttt{letrec}\ ((I_1\ T_1\ E_1)\ \ldots\ (I_n\ T_1\ E_n))\ E_{body}) \rceil =$
$\quad (\texttt{letrec}\ ((I_1\ \lceil E_1 \rceil)\ \ldots\ (I_n\ \lceil E_n \rceil))\ \lceil E_{body} \rceil)$

$\lceil (\texttt{primop}\ O_{name}\ E_1\ \ldots\ E_n) \rceil = (\texttt{primop}\ O_{name}\ \lceil E_1 \rceil\ \ldots\ \lceil E_n \rceil)$

$\lceil (\texttt{tlet}\ ((I_1\ T_1)\ \ldots\ (I_n\ T_n))\ E_{body}) \rceil = \lceil E_{body} \rceil$

$\lceil (\texttt{the}\ T\ E) \rceil = \lceil E \rceil$

$\lceil (\texttt{error}\ I_{message}\ T) \rceil = (\texttt{error}\ I_{message})$

Figure 12.6: Type erasure rules transforming FL/X to FL* expressions.

Theorem 3 (Subject Reduction for FL/X**)** *If E is a well-typed* FL/X *expression with type T and $\lceil E \rceil$ is not a normal form or stuck, then there is a well-typed* FL/X *expression E' with type T such that $\lceil E \rceil \Rightarrow \lceil E' \rceil$.*

12.4 Typed Data

We now consider how to extend FL/X with typed versions of the forms of data studied in Chapter 10. We will see that the goal of maintaining static type checking constrains the ways in which we create and manipulate some data structures.

12.4.1 Typed Products

Figure 12.7 shows the syntax and type rules needed to extend FL/X with pairs, the simplest kind of product. Types formed by the `pairof` type constructor keep track of the types of the first and second components of a pair. This type is introduced by the `pair` construct and eliminated by either `fst` or `snd`. For example, the type of `(pair (+ 1 2) (= 3 4))` is `(pairof int bool)`.

Although `fst` and `snd` are primitive operators in FL, they cannot be primitive operators in FL/X due to the monomorphic nature of the language. For example, the [*pair-elim-F*] rule says that `fst` returns a value whose type is the first type component of the type `(pairof` T_f T_s`)`. Without some form of polymorphism (see Chapter **??**) it is not possible to describe this behavior via a single type assignment for `left` in the standard type environment.

Pairs can be generalized to arbitrary positional products, whose syntax and type rules are presented in Figure 12.8. The `productof` type tracks the number of components and type of each component in a product value. For example, the type of

```
(product (+ 1 2) (= 3 4) (lambda (x) (> x  5)))
```

is

```
(productof int bool (-> (int) bool)).
```

The [*productof-elim*] rule clarifies why the index in a `proj` form must be a manifest integer literal rather than the result of evaluating an arbitrary expression. Otherwise, the type checker would not "know" which component was being extracted and different types could not be allowed at different indices.

Handling named products in a typed language requires additional complexity. As shown in Figure 12.9, the `recordof` type needs to associate record field names with types. Although the [*recordof-elim*] rule is concise, the ellipses in the

Syntax

$E ::= \ldots \mid$ `(pair` E_{fst} E_{snd}`)` [Pair Intro]
$\quad \mid$ `(fst` E_{pair}`)` [Pair Elim First]
$\quad \mid$ `(snd` E_{pair}`)` [Pair Elim Second]

$T ::= \ldots \mid$ `(pairof` T_{fst} T_{snd}`)` [Pair Type]

Type Rules

$$\frac{A \vdash E_f : T_f \;\; ; \;\; A \vdash E_s : T_s}{A \vdash \texttt{(pair } E_f \; E_s\texttt{)} : \texttt{(pairof } T_f \; T_s\texttt{)}} \quad [\textit{pairof-intro}]$$

$$\frac{A \vdash E_{pair} : \texttt{(pairof } T_f \; T_s\texttt{)}}{A \vdash \texttt{(fst}E_{pair}\texttt{)} : T_f} \quad [\textit{pairof-elim-F}]$$

$$\frac{A \vdash E_{pair} : \texttt{(pairof } T_f \; T_s\texttt{)}}{A \vdash \texttt{(snd } E_{pair}\texttt{)} : T_s} \quad [\textit{pairof-elim-S}]$$

Figure 12.7: Handling pairs in FL/X.

Syntax

$E ::= \ldots \mid$ `(product` E*`)` [Product Intro]
$\quad \mid$ `(proj` N_{index} E_{prod}`)` [Product Elim]

$T ::= \ldots \mid$ `(productof` T*`)` [Product Type]

Type Rules

$$\frac{\forall_{i=1}^{n} \; . \; A \vdash E_i : T_i}{A \vdash \texttt{(product } E_1 \; \ldots \; E_n\texttt{)} : \texttt{(productof } T_1 \; \ldots \; T_n\texttt{)}} \quad [\textit{productof-intro}]$$

$$\frac{A \vdash E_{prod} : \texttt{(productof } T_1 \; \ldots \; T_n\texttt{)}}{A \vdash \texttt{(proj } N_{index} \; E_{prod}\texttt{)} : T_{N_{index}}}, \text{ where } 1 \leq N_{index} \leq n \quad [\textit{productof-elim}]$$

Figure 12.8: Handling products in FL/X.

premise type `(recordof ... (I T) ...)` obscure the fact that the type checker must somehow find the binding associated with the selected field name in the list of name/type bindings. Moreover, the fact that the name/type bindings may be in any order complicates the notion of type equality — an issue discussed in the next subsection.

Syntax

$E ::= \ldots \mid$ `(record (`I E`)*)` [Record Intro]
$\quad \mid$ `(select` I E`)` [Record Elim]

$T ::= \ldots \mid$ `(recordof (`I T`)*)` [Record Type]

Type Rules

$$\frac{\forall_{i=1}^{n} \,.\, A \vdash E_i : T_i}{A \vdash \texttt{(record (}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{))} : \texttt{(recordof (}I_1\ T_1\texttt{) ... (}I_n\ T_n\texttt{))}} \quad [\textit{recordof-intro}]$$

$$\frac{A \vdash E : \texttt{(recordof ... (}I\ T\texttt{) ...)}}{A \vdash \texttt{(select}\ I\ E\texttt{)} : T} \quad [\textit{recordof-elim}]$$

Figure 12.9: Handling records in FL/X.

▷ **Exercise 12.1** Consider extending FL/X with a construct `(pair=?` E_{pair_1} E_{pair_2}`)` that returns *true* if the respective components of the pair values of E_{pair_1} and E_{pair_2} are equal, and returns *false* otherwise.

a. Give a type rule for `pair=?`.

b. In dynamically typed FL, write `pair=?` as a user-defined procedure (using the generic `equal?` procedure to compare components).

c. In FL/X, is it possible to write `pair?` as a user-defined procedure? Explain. ◁

12.4.2 Digression: Type Equality

Before the introduction of `recordof` types, it was safe to assume that two types were equal only if they were syntactically identical. But this assumption is no longer valid in the presence of `recordof` types, because two `recordof` types with different binding orders can be considered equal. For example, `(recordof (a int) (b bool))` and `(recordof (b bool) (a int))` are equivalent types.

One way to handle record type equality is to require that all `recordof` types be put into a canonical form – e.g., with name/type bindings alphabetically ordered by name. Another approach is to develop a collection of rules that formalize when two types are equal. In this approach, two types are equal if and only if a proof of equality can be derived from the rules. This second approach is more general than the first because it handles notions of type equality that are not so easily addressed by canonical forms.

Figure 12.10 presents a set of type equality rules for the FL/X types studied thus far. The [*reflexive-=*], [*symmetric-=*], and [*transitive-=*] rules guarantee that $=$ is an equivalence relation. The [`->`-=], [*pairof-=*], and [*productof-=*] rules ensure that $=$ is a congruence over the `->`, `pairof`, and `productof` type constructors. The [*recordof-=*] rule allows the type and the tag names of a `recordof` type to appear in permuted order as long as the named component types are equivalent.

From now on, we assume that the type equality rules in Figure 12.10 are used whenever it is necessary to determine the equality of two FL/X types. Such tests are often implicit in the type constraints of type rules. For example, here is a version of the [*if*] rule in which type equality tests are made explicit:

$$\frac{A \vdash E_{test} : T_{test} \quad ; \quad A \vdash E_{con} : T_{con} \quad ; \quad A \vdash E_{alt} : T_{alt}}{A \vdash \texttt{(if } E_{test} \ E_{con} \ E_{alt}\texttt{)} : T_{result}} \quad [\mathit{if}]$$

where $T_{test} = \texttt{bool}$, $T_{con} = T_{alt}$, and $T_{alt} = T_{result}$

12.4.3 Typed Mutable Data

Mutable data, such as mutable cells, tuples, records, and arrays, are straightforward to handle in an explicitly typed framework. The type rules for mutable cells are presented in Figure 12.11. Both subexpressions of a `begin` form are required to be well-typed, but only the type of the second expression appears in the result type. The `cellof` type constructor tracks the type of the cell contents. In [*cell-set*], the new value is constrained to have the same type as the value already in the cell. The `unit` result type of a `cell-set` form indicates that it is performed for side effect, not for its value.

Note that Figure 12.11 includes a type equality rule for `cellof` types. From now on, we must specify type equality rules for each new type constructor in order to test for equality on the types it constructs.

$$T = T \qquad [\textit{reflexive-=}]$$

$$\frac{T_1 = T_2}{T_2 = T_1} \qquad [\textit{symmetric-=}]$$

$$\frac{T_1 = T_2 \;\; ; \;\; T_2 = T_3}{T_1 = T_3} \qquad [\textit{transitive-=}]$$

$$\frac{\forall_{i=1}^{n} \; . \; S_i = T_i \;\; ; \;\; S_{body} = T_{body}}{\texttt{(->}\;\texttt{(}S_1 \;\ldots\; S_n\texttt{)}\;\; S_{body}\texttt{)} = \texttt{(->}\;\texttt{(}T_1 \;\ldots\; T_n\texttt{)}\;\; T_{body}\texttt{)}} \qquad [\texttt{->-=}]$$

$$\frac{\forall_{i=1}^{2} \; . \; S_i = T_i}{\texttt{(pairof}\;\; S_1 \;\; S_2\texttt{)} = \texttt{(pairof}\;\; T_1 \;\; T_2\texttt{)}} \qquad [\textit{pairof-=}]$$

$$\frac{\forall_{i=1}^{n} \; . \; S_i = T_i}{\texttt{(productof}\;\; S_1 \;\ldots\; S_n\texttt{)} = \texttt{(productof}\;\; T_1 \;\ldots\; T_n\texttt{)}} \qquad [\textit{productof-=}]$$

$$\texttt{(recordof (}J_1 \;\; S_1\texttt{)} \;\ldots\; \texttt{(}J_n \;\; S_n\texttt{))} = \texttt{(recordof (}I_1 \;\; T_1\texttt{)} \;\ldots\; \texttt{(}I_n \;\; T_n\texttt{))} \qquad [\textit{recordof-=}]$$

where $\{J_1, \ldots, J_n\} = \{I_1, \ldots, I_n\}$ and
$\forall_{i=1}^{n} \; . \; \forall_{j=1}^{n} \; . \;\; J_j = I_i$ implies $S_j = T_i$

Figure 12.10: Type equality rules for FL/X.

Syntax

$E ::= \ldots$	`(begin` E_1 E_2`)`	[Sequential Execution]
	`(cell` E`)`	[Cell Creation]
	`(cell-ref` E`)`	[Cell Get]
	`(cell-set!` E_{cell} E_{val}`)`	[Cell Set]
$T ::= \ldots$	`(cellof` T`)`	[Cell Type]

Type Rules

$$\frac{\forall_{i=1}^{2} \,.\, A \vdash E_i \,:\, T_i}{A \vdash \texttt{(begin}\ E_1\ E_2\texttt{)} \,:\, T_2} \quad [\textit{begin}]$$

$$\frac{A \vdash E \,:\, T}{A \vdash \texttt{(cell}\ E\texttt{)} \,:\, \texttt{(cellof}\ T\texttt{)}} \quad [\textit{cellof-intro}]$$

$$\frac{A \vdash E_{cell} \,:\, \texttt{(cellof}\ T\texttt{)}}{A \vdash \texttt{(cell-ref}\ E_{cell}\texttt{)} \,:\, T} \quad [\textit{cellof-elim}]$$

$$\frac{A \vdash E_{cell} \,:\, \texttt{(cellof}\ T_{val}\texttt{)} \;\;;\;\; A \vdash E_{val} \,:\, T_{val}}{A \vdash \texttt{(cell-set!}\ E_{cell}\ E_{val}\texttt{)} \,:\, \texttt{unit}} \quad [\textit{cell-set}]$$

Type Equality

$$\frac{T_1 = T_2}{\texttt{(cellof}\ T_1\texttt{)} = \texttt{(cellof}\ T_2\texttt{)}} \quad [\textit{cellof-=}]$$

Figure 12.11: Handling cells in FL/X.

12.4.4 Typed Sums

Although sums are in some sense the duals of products, the type rules for named sums (Figure **??**) are far more complex than the type rules for named products in an explicitly typed language. Like the `recordof` type constructor,

Syntax

$E ::= \ldots \mid$ `(one` T_{oneof} I_{tag} E_{val}`)` [Oneof Intro]

$\quad \mid$ `(tagcase` E_{disc} I_{val} `(`I_{tag} E_{body}`)*` `[(else` E_{else}`)])` [Oneof Elim]

$T ::= \ldots \mid$ `(oneof (`I T`)*)` [Oneof Type]

Type Rules

$$\frac{A \vdash E_{val} : T_{val}}{A \vdash \texttt{(one } T_{oneof} \ I_{tag} \ E_{val}\texttt{)} : T_{oneof}} \quad [\textit{oneof-intro}]$$

where $T_{oneof} = \texttt{(oneof } \ldots \ \texttt{(}I_{tag} \ T_{val}\texttt{)} \ \ldots\texttt{)}$

$$\frac{\begin{array}{c} A \vdash E_{disc} : \texttt{(oneof (}I_{\pi(1)} \ T_{\pi(1)}\texttt{)} \ \ldots \ \texttt{(}I_{\pi(n)} \ T_{\pi(n)}\texttt{))} \\ \forall_{i=1}^{n} \ . \ A[I_{val} : T_{\pi(i)}] \vdash E_i : T_{result} \end{array}}{A \vdash \texttt{(tagcase } E_{disc} \ I_{val} \ \texttt{(}I_1 \ E_1\texttt{)} \ \ldots \ \texttt{(}I_n \ E_n\texttt{))} : T_{result}} \quad [\textit{oneof-elim1}]$$

where π is a permutation on the integer range $[1..n]$

$$\frac{\begin{array}{c} A \vdash E_{disc} : \texttt{(oneof (}J_1 \ T_1\texttt{)} \ \ldots \ \texttt{(}J_m \ T_m\texttt{))} \\ \forall_{i=1}^{n} \ . \ A[I_{val} : T_{f(i)}] \vdash E_i : T_{result} \\ A \vdash E_{default} : T_{result} \end{array}}{\begin{array}{c} A \vdash \texttt{(tagcase } E_{disc} \ I_{val} \ \texttt{(}I_1 \ E_1\texttt{)} \ \ldots \ \texttt{(}I_n \ E_n\texttt{)} \\ \texttt{(else } E_{default}\texttt{))} : T_{result} \end{array}} \quad [\textit{oneof-elim2}]$$

where $n \leq m$ and for all $i \in [1..n]$ there is a unique $f(i) \in [1..m]$ such that $I_i{=}J_{f(i)}$

Type Equality

$$\texttt{(oneof (}J_1 \ S_1\texttt{)} \ \ldots \ \texttt{(}J_n \ S_n\texttt{))} = \texttt{(oneof (}I_1 \ T_1\texttt{)} \ \ldots \ \texttt{(}I_n \ T_n\texttt{))} \quad [\textit{oneof-=}]$$

where $\{J_1, \ldots, J_n\} = \{I_1, \ldots, I_n\}$ and
$\forall_{i=1}^{n} \ . \ \forall_{j=1}^{n} \ . \ J_j{=}I_i$ implies $S_j{=}T_i$

Figure 12.12: Handling oneofs in FL/X.

the `oneof` type constructor combines a sequence of named types whose order is irrelevant (as specified by [*oneof-=*]). But the oneof introduction form `(one` T_{oneof} I_{tag} E_{val}`)` must include the explicit type T_{oneof} of the resulting oneof value for use in the [*oneof-intro*] rule. This is necessary to preserve the FL/X property that the type of any expression in a given type environment is

unambiguous and can be determined without any guessing. In the dual [*recordof-elim*] rule for checking `(select` I E`)`, the record type of E, which includes all field types, can be determined from the type environment. In contrast, a `one` form would only determine the type of *one* field type if the type T_{oneof} were not explicitly included.

The elimination rules [*oneof-elim1*] and [*oneof-elim2*] are also more complex than the dual record introduction form. Having to bind the tagged value to the name I_{val}, dealing with an optional `else` clause, and handling the fact that the ordering of bindings is irrelevant all contribute to the complexity of the `tagcase` rules.

As a concrete example of sum and product types, consider the typed shape example in Figure 12.13. The `shape` type is an abbreviation for a `oneof` type with three tags. Such abbreviations are crucial for enhancing code readability; the example would be much more verbose without the abbreviation. We could have consistently used `productof` or `recordof` types for all of the the oneof components, but have chosen to use different type constructors for different components just to show that this is possible. Note that in `perim` and `double`, the variable `v` in the `tagcase` forms assumes different types in different clauses: `v` has type `int` in a `square` clause, type `(pairof int int)` in a `rectangle` clause, and type `(productof int int int)` in a `triangle` clause. All clauses of a `tagcase` are required to return the same type. This return type is `int` for `perim` and `shape` for `double`.

▷ **Exercise 12.2** Construct type derivations showing that the `perim` and `double` functions in Figure 12.13 are well-typed. ◁

12.4.5 Typed Lists

The geometric shape examples above shows that simple sum-of-product data types can be expressed in FL/X by composing sums and products. However, as it stands, FL/X does not have the power to express recursively structured sum-of-product data types like lists and trees. Here we extend FL/X with a built-in list data type. In Section 12.5, we extend FL/X with a recursive type mechanism that allows lists and trees to be constructed by programmers.

Figure 12.14 presents the essence of lists in FL/X. Unlike in FL, where lists are just sugar for idiomatic uses of pairs, FL/X supplies special forms for creating lists (`cons` and `null`), decomposing non-empty lists (`car` and `cdr`), and testing for empty lists (`null?`). All of these manipulate values of types created with the `listof` type constructor. The type `(listof` T`)` describes lists whose components all have the same type T. FL/X lists are said to be **homogeneous**,

```
(define-type shape
  (oneof (square int)
         (rectangle (pairof int int))
         (triangle (productof int int int))))

(define perim (-> (shape) int)
  (lambda ((shp shape))
    (tagcase shp v
      (square (* 4 v))
      (rectangle (* 2 (+ (left v) (right v))))
      (triangle (+ (proj 1 v) (proj 2 v) (proj 3 v))))))

(define double (-> (shape) shape)
  (lambda ((shp shape))
    (tagcase shp v
      (square (one shape square (* 2 v)))
      (rectangle (one shape rectangle
                  (pair (* 2 (left v)) (* 2(right v)))))
      (triangle (one shape triangle
                  (product (* 2 (proj 1 v))
                           (* 2 (proj 2 v))
                           (* 2 (proj 3 v))))))))
```

Figure 12.13: Typed shapes in FL/X.

in constrast to the **heterogeneous** lists, of FL. To model heterogeneous lists within FL/X, such as a list of integers and booleans, it is necessary to inject the different types into an explicit sum type.

Syntax

$E ::= \ldots$ | `(cons` E_{head} E_{tail}`)`	[List Creation]
| `(car` E_{list}`)`	[List Head]
| `(cdr` E_{list}`)`	[List Tail]
| `(null` T`)`	[Empty List]
| `(null?` E_{list}`)`	[Empty List Test]
$T ::= \ldots$ | `(listof` T`)`	[List Type]

Type Rules

$$\frac{A \vdash E_{head} : T \quad A \vdash E_{tail} : \texttt{(listof } T\texttt{)}}{A \vdash \texttt{(cons } E_{head} \ E_{tail}\texttt{)} : T_{(listof T)}} \quad [cons]$$

$$\frac{A \vdash E_{list} : \texttt{(listof } T\texttt{)}}{A \vdash \texttt{(car } E_{list}\texttt{)} : T} \quad [car]$$

$$\frac{A \vdash E_{list} : \texttt{(listof } T\texttt{)}}{A \vdash \texttt{(cdr } E_{list}\texttt{)} : \texttt{(listof } T\texttt{)}} \quad [cdr]$$

$$A \vdash \texttt{(null } T\texttt{)} : \texttt{(listof } T\texttt{)} \quad [null]$$

$$\frac{A \vdash E_{list} : \texttt{(listof } T\texttt{)}}{A \vdash \texttt{(null? } E_{list}\texttt{)} : \texttt{bool}} \quad [null?]$$

Type Equality

$$\frac{T_1 = T_2}{\texttt{(listof } T_1\texttt{)} = \texttt{(listof } T_2\texttt{)}} \quad [listof\text{-}=]$$

Figure 12.14: Handling lists in FL/X.

The `null` form includes the element type of the empty list. So `(null int)` is an empty integer list, `(null bool)` is an empty boolean list, and `(null (listof int))` is an empty list of integer lists.

```
(define map-shape-int (-> ((-> (shape) int) (listof shape)) (listof int))
  (lambda ((f (-> (shape) int)) (ss (listof shape)))
    (if (null? ss)
        (null int)
        (cons (f (car ss))
              (map-shape-int f (cdr ss))))))

(map-shape-int perim
  (cons (rectangle (pair 4 5))
        (cons (triangle (product 7 8 9))
              (cons (square 3)
                    (null shape)))))
```

12.5 Recursive Types

Recursive procedures often manipulate recursively-structured data that cannot be described in terms of compound types alone. The `recof` and `rectype` type constructs (Figure 12.15) are used to specify the types of such data. `recof` allows the specification of a single recursive type in the same manner that the FL `rec` construct specifies a single recursive value. For example, here `recof` is used to specify the type of a binary tree with integer leaves:

```
(recof int-tree
  (oneof (leaf int)
         (node (recordof (left int-tree)
                         (right int-tree)))))
```

`rectype` is the type domain analog to `letrec`. It permits a mutually recursive set of named types to be used in a body type expression. For example, here is a use of `rectype` to specify a binary tree that has integers at odd-numbered levels and booleans at even-numbered levels:

```
(rectype ((int-level
            (oneof (leaf int)
                   (node (recordof (left bool-level)
                                   (right bool-level)))))
          (bool-level
            (oneof (leaf bool)
                   (node (recordof (left int-level)
                                   (right int-level))))))
  int-level)
```

What does it mean for two recursive types to be equivalent? For example, consider the four types in Figure 12.16. All of the types describe infinite lists of

```
E ::= ... | (recof I T)                    [Recursive Type]
    | (rectype ((I T)*) T_body)            [Mutually Recursive Types]
```

Figure 12.15: Syntax for recursive types in FL/X.

alternating integer and boolean values. T_2 is a copy of T_1 in which the `recof` bound type variable has been consistently renamed. T_3 is a copy of T_1 in which the definition of `iblist` has been unwound one level. In T_4, the recursive type `bilist` describes an infinite list of alternating boolean and integer values. Which pairs of these four types equivalent?

```
T1 = (recof iblist (pairof int (pairof bool iblist)))

T2 = (recof int-bool-list (pairof int (pairof bool int-bool-list)))

T3 = (recof iblist
       (pairof int
               (pairof bool
                       (pairof int
                               (pairof bool iblist)))))

T4 = (pairof int (recof bilist (pairof bool (pairof int bilist))))
```

Figure 12.16: Four types describing infinite lists with alternating integer and boolean values.

The so-called **iso-recursive** approach to formalizing type equality on recursive types is shown in Figure 12.17. The [*recof-α*] rule says that two `recof` types are equal if their bound variables can be consistently renamed. So $T_1 = T_2$ via [*recof-α*]. The [*recof-β*] rule says that a `recof` type is equivalent to the result of substituting the entire `recof` type expression for its bound variable in the body of the `recof`. So $T_1 = T_3$ via [*recof-β*], and $T_2 = T_3$ by the transitivity of type equivalence.

$$\texttt{(recof } I \ T\texttt{)} = \texttt{(recof } I_{new} \ [I_{new}/I]T\texttt{)}, \text{ where } I_{new} \notin \mathit{FreeIds}[\![T]\!] \qquad [\textit{recof-}\alpha]$$

$$\texttt{(recof } I \ T\texttt{)} = [\texttt{(recof } I \ T\texttt{)}/I]T \qquad [\textit{recof-}\beta]$$

Figure 12.17: Iso-recursive type equality rules for `recof` types.

Can T_4 be shown to be equivalent to T_1, T_2, or T_3 using [*recof-α*] and [*recof-β*]? No! We can prove this via the following observation. In each of T_1, T_2, and T_3, the number of occurrences of the type `int` is equal to the number of occurrences of the type `bool`. In T_4, the number of occurrencs of `int` is one more than the number of occurrences of `bool`. Since each application of [*recof-α*] and [*recof-β*] preserves the difference between the number of occurrences of `int` and of `bool`, T_4 can never be shown to be equivalent to the other types via these rules.

There is another approach to recursive type equivalence in which T_4 *is* equivalent to the other types. In this so-called **equi-recursive** approach, two recursive types are considered to be equivalent if their complete (potentially infinite) unwindings are equal. Under this criterion, all four of the types in Figure 12.16 are equivalent, because all of them unwind to an infinite type describing a list of alternating integers and booleans:

```
(pairof int (pairof bool (pairof int (pairof bool ...)))).
```

There are two approaches for formalizing equi-recursive type equality:

1. We can extend the type equivalence rules to maintain a set of assumed type equivalences. This set is initially empty. Whenever T_1 and T_2 are compared for equivalence and at least one of T_1 or T_2 is a `recof` or `rectype` type, the equivalence $T_1 \equiv T_2$ is added to the set of assumptions before unwinding a `recof`. If an equivalence already in the set of assumptions is encountered, the equivalence is assumed to hold. The basic idea of this approach is to assume that types are equivalent unless some contradiction can be found.

2. It turns out that there is a normal form for FL/X types. `recordof` and `oneof` types can be normalized by picking a convention for ordering their tag and field names. `recof` and `rectype` types can be viewed as finite state machines, which have normal forms. The existence of normal forms implies that it is possible to perform type equivalence by normalizing two types and syntactically comparing their normal forms.

▷ **Exercise 12.3** Give iso-recursive type equality rules for `rectype`. ◁

▷ **Exercise 12.4** Figure 12.18 presents five types. Which of these types are considered equivalent (a) under the iso-recursive approach and (b) under the equi-recursive approach?

◁

```
T_it1 = (recof it
         (oneof (leaf int)
                (node (recordof (left it)
                                (right it))))

T_it2 = (recof int-tree
         (oneof (leaf int)
                (node (recordof (left int-tree)
                                (right int-tree)))))

T_it3 = (recof it
         (oneof
           (leaf int)
           (node (recordof (left (recof it
                                   (oneof (leaf int)
                                          (node (recordof (left it)
                                                          (right it))))))
                           (right it)))))

T_it4 = (recof it
         (oneof
           (leaf int)
           (node (recordof (left it)
                           (right (recof it
                                    (oneof (leaf int)
                                           (node (recordof (left it)
                                                           (right it))))))))))


T_it5 = (recof it
         (oneof
           (leaf int)
           (node (recordof (left (recof it
                                   (oneof (leaf int)
                                          (node (recordof (left it)
                                                          (right it))))))
                           (right (recof it
                                    (oneof (leaf int)
                                           (node (recordof (left it)
                                                           (right it)))))))))
```

Figure 12.18: Five types for integer-leaved binary trees.

▷ **Exercise 12.5** Based on the idea of maintaining a set of type assumptions, write a program that computes type equivalence for FL/X types. Your program should effectively treat recursive types as their infinite unwindings. ◁

▷ **Exercise 12.6**

a. Show that a `recof` type can be viewed as a finite state machine.

b. Based on the first part, develop a notion of normal forms for FL/X types.

c. Write a program that determines the normal form for a FL/X type, and use this as a mechanism for testing type equivalence.

◁

Reading

A good introduction to various dimensions of types is a survey article written by Cardelli and Wegner [CW85]. A more in-depth discussion of these dimensions can be found in textbooks by Pierce [Pie02], by Mitchell [Mit96], and by Schmidt [Sch94]. For types in the context of the lambda calculus, see [Bar92a]. For work on types in object-oriented programming, see [GM94].

Chapter 13

Subtyping and Polymorphism

We need a quote for this chapter.

— *Mark A. Sheldon*

13.1 Subtyping

13.1.1 Motivation

The typing rules presented for FL/X are rather restrictive. For example, consider a `get-name` procedure that extracts the contents of the `name` field of a record:

```
(define get-name (-> ((recordof (name string))) string)
  (lambda ((r (recordof (name string))))
    (select name r)))
```

According to the FL/X typing rules, the following use of `get-name` does not type check:

```
(get-name (record (name "Paula Morwicz") (age 35) (student? #f)))
```

The problem is that the given record has three fields, but the type of `get-name` dictates that the argument record must have exactly one field. Yet the presence of the extra fields does not compromise the type safety of the expression. We can reliably extract a string from the `name` field of *any* record whose type binds `name` to `string`. No constraints on the number or nature of other fields is implied by the extraction of the `name` field.

Situations like `get-name` can be addressed by the notion of **type inclusion** (also called **subtyping**). We say that S is a **subtype** of T (written $S \sqsubseteq T$) if

all expressions of type S can be used (in a type-safe manner) in every situation where an expression of type T is used. Viewing types as sets, $S \sqsubseteq T$ means that $S \subseteq T$. If S is a subtype of T, we can also say that T is a **supertype** of S.

13.1.2 FL/XS

We shall consider a variant of FL/X, called FL/XS (the "S" stands for "Subtypes"), that supports subtyping. The typing rules of FL/XS are the same as those for FL/X except for the addition of the [*inclusion*] rule of Figure 13.1. This rule formalizes the notion that a subtype element can be used in any situation where a supertype element is expected.

$$\frac{(A \vdash E : T)\ ;\ (T \sqsubseteq T')}{A \vdash E : T'} \qquad [\textit{inclusion}]$$

Figure 13.1: Additional typing rule for FL/XS. This rule augments the typing rules for FL/X.

Subtyping is a relation on type expressions that can be defined by a collection of rules. The subtyping rules for FL/XS appear in Figure 13.2. The [*reflexive-*

$$T \sqsubseteq T \qquad [\textit{reflexive-}\sqsubseteq]$$

$$\frac{T_1 \sqsubseteq T_2 \ ;\ T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3} \qquad [\textit{transitive-}\sqsubseteq]$$

$$\frac{\forall \mathrm{i}\ .\ \exists \mathrm{j}\ .\ ((I_i = J_j) \wedge (S_j \sqsubseteq T_i))}{\texttt{(recordof } (J_1\ S_1) \ \ldots\ (J_m\ S_m)\texttt{)} \sqsubseteq \texttt{(recordof } (I_1\ T_1) \ \ldots\ (I_n\ T_n)\texttt{)}} \qquad [\textit{recordof-}\sqsubseteq]$$

$$\frac{\forall \mathrm{j}\ .\ \exists \mathrm{i}\ .\ ((J_j = I_i) \wedge (S_j \sqsubseteq T_i))}{\texttt{(oneof } (J_1\ S_1) \ \ldots\ (J_m\ S_m)\texttt{)} \sqsubseteq \texttt{(oneof } (I_1\ T_1) \ \ldots\ (I_n\ T_n)\texttt{)}} \qquad [\textit{oneof-}\sqsubseteq]$$

$$\frac{\forall \mathrm{i}\ .\ (T_i \sqsubseteq S_i) \ ;\ S_{body} \sqsubseteq T_{body}}{\texttt{(->} \ (S_1 \ \ldots\ S_n)\ S_{body}\texttt{)} \sqsubseteq \texttt{(->} \ (T_1 \ \ldots\ T_n)\ T_{body}\texttt{)}} \qquad [\texttt{->-}\sqsubseteq]$$

$$\frac{\forall T\ .\ ([T/I_1]T_1 \sqsubseteq [T/I_2]T_2)}{\texttt{(recof } I_1\ T_1\texttt{)} \sqsubseteq \texttt{(recof } I_2\ T_2\texttt{)}} \qquad [\textit{recof-}\sqsubseteq]$$

Figure 13.2: Subtyping rules for FL/XS.

$\sqsubseteq$] and [*transitive*-$\sqsubseteq$] rules guarantee that subtyping is a reflexive, transitive closure of the relation induced by the other rules. The [*recordof*-$\sqsubseteq$] rule says that a record type S is a subtype of a record type T if (1) S has at least the fields of T and (2) for each of the field names in T, the field types in S are in a subtype relation with the corresponding field types of T. Condition (1) says that extra fields in S can't hurt, since they will be ignored by any code that extracts only the fields mentioned in T. Condition (2) says that subtyping is allowed among the corresponding component types of the fields named by T. When corresponding type components are related via subtyping in the same direction as the entire type, the subtyping of the components is said to be **monotonic**. Thus, the second condition for record subtyping could be rephrased as "for each of the field names in T, the corresponding field types are related monotonically."

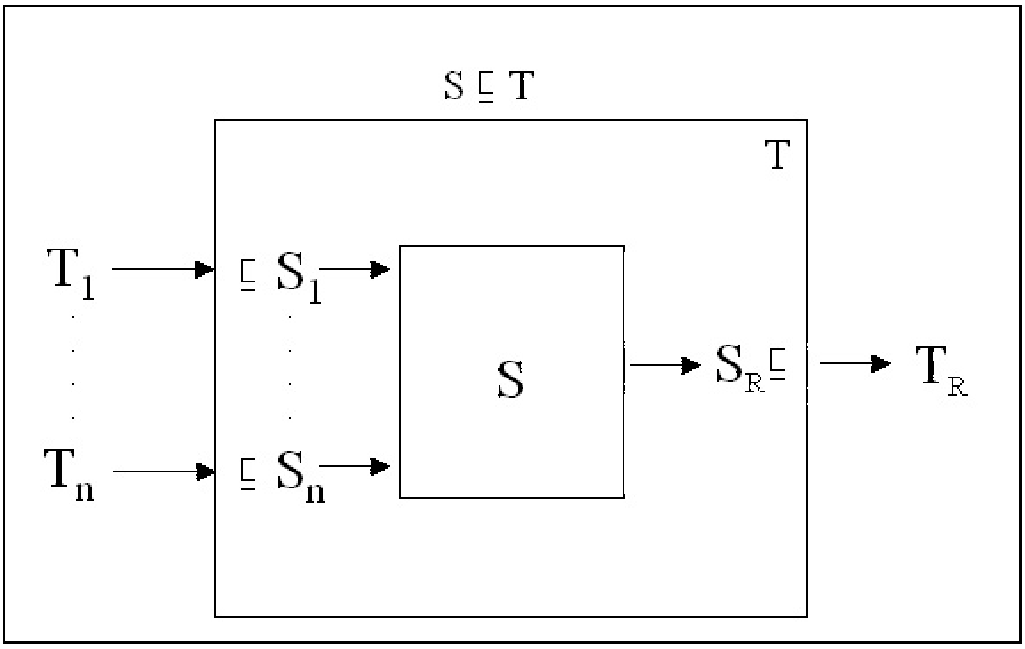


Figure 13.3: Procedure subtyping is monotonic on result of types and anti-monotonic on input types.

The [*oneof*-$\sqsubseteq$] rule is a dual of the [*record*-$\sqsubseteq$] rule: A oneof type S is a subtype of a oneof type T if it has *fewer* tags. The types of the shared tags are related monotonically. This makes sense because if a program is prepared to handle all the cases of the supertype (T), then it is prepared for the fewer possible cases of the subtype (S).

In the rule for procedure subtyping, the return types are monotonic, but

the parameter types are **anti-monotonic** — i.e., they are related via subtyping in a direction opposite to the subtyping of the procedure type as a whole. As shown in Figure 13.3, procedures are supposed to handle any element in the class specified by the type of the formal parameter. Thus if a procedure expects type S_1, it can be used in a context where it will be passed elements from the smaller class $T_1 \sqsubseteq S_1$. (Alternatively, one may always safely use a procedure that is defined on **more** values.) On the other hand, it is always safe to use a procedure that returns elements of type S_R where T_R is expected if $S_R \sqsubseteq T_R$.

The rule for the subtyping of `recof` types says that `recof` type S is a subtype of another `recof` type T if the result of instantiating the body of S with any type is a subtype of the result of instantiating the body of T with the same type.

Note that there are no special subtyping rules for `cellof`, `listof`, and `vectorof` types. This is not an oversight; types of these forms are only in a subtype relation if they are type equivalent! The following monotonic rule for `cellof` subtyping seems natural, but it is actually *incorrect*:

$$\frac{T_1 \sqsubseteq T_2}{\texttt{(cellof } T_1\texttt{)} \sqsubseteq \texttt{(cellof } T_2\texttt{)}} \qquad [\textit{incorrect-cellof-}\sqsubseteq]$$

It is possible to show expressions that are well-typed using this rule, but that would raise a run-time type error in the corresponding dynamically typed system. We leave the generation of such an example as an exercise for the reader. Corresponding rules for `listof` and `vectorof` subtyping suffer the same problem as the `cellof` rule above. In all cases, the fundamental problem is due to side effects. In fact, the expected monotonic rule for these types is valid if they are immutable.

Finally, in a language with subtyping, it is reasonable to define type equivalence as mutual inclusion.

13.1.3 Discussion

The [*inclusion*] rule is a simple way of extending the FL/X typing rules with subtyping, but it harbors some problems. In FL/X, every expression has exactly one type (actually, an expression may have many types, but they are all type equivalent). The [*inclusion*] rule destroys this unique typing property allowing a single expression to have many (non-equivalent) types. For instance, in FL/XS, it is possible to prove that the expression

```
(record (a 3) (b #t))
```

has each of the following types:

```
(recordof (a int) (b bool))
(recordof (a int))
(recordof (b bool))
(recordof)
```

The lack of unique typing is not in itself a problem, but it can complicate other analysis. In the case of FL/XS, the lack of unique typing makes it difficult to write a type checker. The problem is that a straightforward type checker needs to choose one of many possible types before enough information is known to make a correct decision. Consider type checking the following simple expression:

```
(let ((c (cell (record (a 3) (b #t)))))
  (begin (cell-set! c (record (a 4)))
         (select a (cell-ref c))))
```

This expression is well-typed according to the typing rules of FL/XS. But the proof of well-typedness requires invoking the [*inclusion*] rule to hide the `b` field of the first `record` so that `c` has the type `(cellof (recordof (a int)))`. A straightforward type checker needs to decide upon the type of `c` before it examines the rest of the program. At this point the type checker does not "know" which fields of the record content of `c` may be accessed later and how `c` will be mutated. (In fact, such details are undecidable in general.) But without such knowledge, the type checker may make an inappropriate choice. For instance, upon encountering the `cell` expression, it seems prudent to assume that `c` has the type

```
(cellof (recordof (a int) (b bool)))
```

Unfortunately, the program is not well-typed under this assumption. In this case, the correct type for `c` is

```
(cellof (recordof (a int)))
```

but this is only OK because it so happens that the program does not later extract the `b` field. Without backtracking or some sophisticated mechanism for managing constraints, simple expressions like the one above will not be type checked properly.

It is possible to restore unique typing and make type checking easier by restricting the contexts in which subtyping is allowable. Figure 13.4 presents an alternate set of type rules that can be used in place of the [*inclusion*] rule. The [*call-inclusion*] rule permits actual arguments to be subtypes of the formal parameters expected by the called procedure. This rule pinpoints procedure call boundary as the most useful spot where the power of subtyping is used implicitly. Implicit coercion is common in other languages: JAVA allows methods to accept arguments of a subclass of the expected class, and numerous languages allow implicit coercion among numeric types (e.g., converting an integer to a floating

$$\frac{A \vdash E_{rator} : \texttt{(->}\ \texttt{(}T_1\ \ldots T_n\texttt{)}\ T_{body}\texttt{)} \quad \forall \text{i}\ ((A \vdash E_i : S_i) \wedge (S_i \sqsubseteq T_i))}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots E_n\texttt{)} : T_{body}} \quad [\textit{call-inclusion}]$$

$$\frac{A \vdash E : S \quad S \sqsubseteq T}{A \vdash \texttt{(the}\ T\ E\texttt{)} : T} \quad [\textit{the-inclusion}]$$

Figure 13.4: Modified typing rules for FL/XS

point number).[1]

The alternate set of type rules also includes the [*the-inclusion*] for handling `the`. Under this rule, `the` is no longer merely a type declaration, but a means of **type coercion** — that is, a means of making a value appear to have as its type a supertype of its actual type. In FL/XS, an item can only be coerced to an object of a supertype, and thus no type loophole can arise. Some languages support arbitrary coercion as a deliberate type loophole. C's *casts* are a prime example.

```
;; This type checks
(select a (the (recordof (a int))
               (record (a 3) (b #t))))

;; This fails to type check, because B is hidden by coercion
(select b (the (recordof (a int))
               (record (a 3) (b #t))))
```

If the [*inclusion*], [*call*], and [*the*] rules are replaced with the rules in 13.4, then every FL/XS expression has a single type. This is because implicit subtyping is limited to argument positions while all other coercions must be explicitly made by the programmer. This limitation also makes it possible to implement a straightforward type checker that embodies the rules; the situations in which subtyping needs to be employed are very constrained. Of course, the price of increased simplicity is a reduction in the power of the type system. Under the alternate set of typing rules, some expressions well-typed under the [*inclusion*] rule are no longer well-typed. For example, reconsider an example from above:

[1] In the case of numeric coercion, there is an actual runtime change in representation that must be inserted.

```
(let ((c (cell (record (a 3) (b #t)))))
  (begin (cell-set! c (record (a 4)))
         (select a (cell-ref c))))
```

Under the alternate rules, this expression is not well-typed. The variable `c` is found to have the type

```
(cellof (recordof (a int) (b bool)))
```

However, because the [*cell-set*] rule requires the new value to have the same type as that stored in the cell, type checking fails at the `cell-set!` expression.

▷ **Exercise 13.1** Show that the following subtyping rule for `cellof` types is unsound:

$$\frac{T_1 \sqsubseteq T_2}{\texttt{(cellof } T_1\texttt{)} \sqsubseteq \texttt{(cellof } T_2\texttt{)}} \qquad [\textit{incorrect-cellof-}\sqsubseteq]$$

Do this by exhibiting an expression that is well-typed under this rule, but which would raise a run-time type error in a dynamically-typed version of FL/XS. ◁

▷ **Exercise 13.2** Suppose that FL/XS were extended to include immutable lists of type `(ilistof` T`)` with operations `icons`, `icar`, and `icdr`. Argue that the following subtyping rule for immutable list types is sound:

$$\frac{T_1 \sqsubseteq T_2}{\texttt{(ilistof } T_1\texttt{)} \sqsubseteq \texttt{(ilistof } T_2\texttt{)}} \qquad [\textit{immutable-listof-}\sqsubseteq]$$

◁

▷ **Exercise 13.3** The typing rules in Figure 13.4 can be extended to handle limited subtyping for certain cell, list, and vector operations while still maintaining the unique typing property. For example, if `lst` is defined as

```
(define lst (cons (record (a 3)) (null (recordof (a int)))))
```

then it seems reasonable that

```
(cons (record (a 7) (b #t)) lst)
```

should type check with `(listof (recordof (a int)))` as its type.

Extend the rules of Figure 13.4 to include special subtyping rules for cell, list, and vector special forms, where they make sense. Argue that your rules are (1) safe and (2) preserve the unique typing property of expressions. ◁

▷ **Exercise 13.4** Ben Bitddidle has decided to improve FL/XS by allowing user defined procedures to return multiple values instead of just one. Here's an example of a program that uses Ben's new improvement:

```
(let ((f (lambda ((x int)) (result (* x x) (< x 10)))))
  (result-bind (f 4) (i b)
    (if b (+ i 1) 2)))
```
$\xrightarrow[eval]{}$ *17*

Multiple values are returned from a procedure by `result` and they are bound at the point of application with `result-bind`. In the above example, `i` is bound to 16, and `b` is bound to `#t`. The syntax of FL/XS expressions is expanded to include `result` and `result-bind`:

E ::= ... | `(result` E^*`)` | `(result-bind` E_{app} `(`I^*`)` E_{body}`)`

Values created by `result` can only be used with `result-bind`. In particular, E and `(result` E`)` do not have the same type and are not equivalent. Procedures do not have to return multiple values. The type domain includes a new type constructor for multiple-value return values that are created by `result`:

T ::= ... | `(result-of` T^*`)`

a. Give the typing rules for the `result` and `result-bind` constructs.

b. What are the subtyping rules for `result-of` types? ◁

▷ **Exercise 13.5** Bud Lojack decides to add exceptions with termination semantics to FL/X. He extends the grammar as follows:

E ::= ... | `(raise` I_{except} E_{val}`)` | `(handle` I_{except} E_{handle} E_{body}`)`

Recall the informal semantics of `raise` and `handle` from Exercise 9.12 on page 407:

- `(raise` I_{except} E_{val}`)` evaluates E_{val} and applies the current handler for the exception named I_{except} to the resulting value. This application takes place in the handler environment and continuation of the handler.
- `(handle` I_{except} E_{handle} E_{body}`)` first evaluates E_{handle}. It is an error if the value of E_{handle} is not a procedure of one argument. Otherwise the (procedure) value of E_{handle} is installed as the current handler of the exception named I_{except}, and E_{body} is evaluated. If E_{body} returns normally, the value of E_{handle} is removed as the handler of I_{except}, and the value of E_{body} is returned.

Bud wants to modify the type system of FL/X to support the newly introduced constructs. First, he extends the type grammar to include a new type for exception handlers:

T ::= ... | `(handlerof` T`)`

Next, he suggests the following typing rules for `handle` and `raise`:

$$\frac{A \vdash E_{handle} \ : \ \texttt{(->}\ \texttt{(}T_1\texttt{)}\ T_2\texttt{)} \qquad A[I_{except} : \texttt{(handlerof}\ T_1\texttt{)}] \vdash E_{body} \ : \ T_2}{A \vdash \texttt{(handle}\ I_{except}\ E_{handle}\ E_{body}\texttt{)} \ : \ T_2} \quad [handle]$$

$$\frac{A \vdash I_{except} \ : \ \texttt{(handlerof}\ T\texttt{)}, \ A \vdash E_{val} \ : \ T}{A \vdash \texttt{(raise}\ I_{except}\ E_{val}\texttt{)} \ : \ T'} \quad [raise]$$

a. Show that Bud's typing rules result in an unsound type system by providing expressions for E_{first} and E_{second} in the following expression such that the expression is well-typed by Bud's rules, but will generate a dynamic type error.

```
(handle an-exn E_first
   (let ((f (lambda () (raise an-exn 17))))
     (handle an-exn E_second
        (f))))
```

Scared by this initial failure, Bud calls his more skilled friend Ty Pingnut and gives him the task of defining a sound type system for `raise` and `handle`. Ty makes the following change to the grammar of types:

$$\begin{array}{lcl} T & ::= & \ldots \text{ normal FL/X types except for } \texttt{->} \ldots \\ & | & \texttt{(->}\ S\ \texttt{(}T\texttt{*)}\ T\texttt{)} \\ & | & \texttt{void} \\ \\ S & \in & \text{Exn-Spec} \\ S & ::= & \{\ \langle I_1, T_1\rangle, \ldots, \langle I_n, T_n\rangle\ \} \end{array}$$

The type system is changed to have judgments of the form

$$A \vdash E : T \ \$ \ S$$

This can be read, "under type environment A, expression E has type T and may raise exceptions as specified by S." An Exn-Spec S is a set of $\langle I_i, T_i\rangle$ pairs that indicates exceptions that may be raised when E is evaluated, and the type of the value raised for each exception. For example, the judgment

$$A \vdash E : \texttt{bool} \ \$ \ \{\langle \texttt{x}, \texttt{bool}\rangle, \langle \texttt{y}, \texttt{int}\rangle\}$$

indicates that if E returns normally, its value will have type `bool`; and that evaluation of E could cause the exception `x` to be raised with a value of type `bool`, or the exception `y` to be raised with a value of type `int`. Ty's type system guarantees that no other exceptions can be raised by E. Note that in Ty's system, a procedure type includes an Exn-Spec S that describes the *latent* exceptions that might be raised *when the procedure is applied.*

Moreover, Ty uses *exception masking* to remove exceptions from judgments when it is clear that they will be handled:

$$A \vdash \texttt{(handle x (lambda ((z bool)) z)}\ E\texttt{)} : \texttt{bool} \ \$ \ \{\langle \texttt{y}, \texttt{int}\rangle\}.$$

Ty uses `void` as the type of a `raise` expression:

$$\vdash \texttt{(raise x 4)} : \texttt{void \$ } \{\langle \texttt{x}, \texttt{int} \rangle\}$$

It is a general property of Ty's system that any expression of type `void` is guaranteed to raise an exception, and therefore, will never return a result. Since an expression of type `void` can never return, it makes sense to think of `void` as a subtype of every type. Ty uses this idea to define a subtyping relation, $\sqsubseteq$, that is similar to the subtyping relation for FL/XS, but has the following additional rule:

$$\texttt{void} \sqsubseteq T \qquad [\texttt{void-}\sqsubseteq]$$

Also, the procedure subtyping rule has been modified to be monotonic on the set of possible exceptions:

$$\frac{\forall \texttt{i} \,.\, (T_i\texttt{'} \sqsubseteq T_i),\ T_{body} \sqsubseteq T_{body}\texttt{'},\ S \subseteq S'}{\texttt{(->}\ S\ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)} \sqsubseteq \texttt{(->}\ S'\ \texttt{(}T_1\texttt{'}\ \ldots\ T_n\texttt{')}\ T_{body}\texttt{')}} \qquad [\texttt{->-}\sqsubseteq]$$

All other subtyping rules for FL/XS are unchanged in Ty's system.

Here are some of Ty's typing rules:

$$A \vdash N : \texttt{int \$ } \{\} \qquad [\textit{int}]$$

$$A \vdash B : \texttt{bool \$ } \{\} \qquad [\textit{bool}]$$

$$\frac{\begin{array}{c} A \vdash E_1 : \texttt{bool \$ } S_1 \\ A \vdash E_2 : T \texttt{ \$ } S_2 \\ A \vdash E_3 : T \texttt{ \$ } S_3 \end{array}}{A \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : T \texttt{ \$ } S_1 \cup S_2 \cup S_3} \qquad [\textit{if}]$$

$$\frac{A \vdash E : T \texttt{ \$ } S,\ T \sqsubseteq T',\ S \subseteq S'}{A \vdash E : T' \texttt{ \$ } S'} \qquad [\textit{inclusion}]$$

Note in particular the rule [*inclusion*], which is crucial in typing the following examples:

$$\vdash \texttt{(if \#t \#f (raise x 4))} : \texttt{bool \$ } \{\langle \texttt{x}, \texttt{int} \rangle\}$$
$$\vdash \texttt{(if \#f (raise x 4) (raise x \#t))} : \texttt{void \$ } \{\langle \texttt{x}, \texttt{int} \rangle, \langle \texttt{x}, \texttt{bool} \rangle\}$$

In the second example, values of two incompatible types (`int` and `bool`) are raised for the same exception `x`. Because we are working in a language without polymorphism, it is impossible to write a handler for both values.

b. Give the typing rule for `raise`.

c. Give the typing rule for `handle`.

d. Suppose we alter the syntax of `error` to be `(error` *Y*`)`. What is the new typing rule for `error`? ◁

13.2 Polymorphic Types

Monomorphic type systems are easy to reason about, but they hinder the development of reusable code. In particular, monomorphic languages prevent the programmer from expressing **polymorphic** values — values (typically procedures) that can have different types in different contexts. In this section, we develop a type system that allows the expression of polymorphic values.

As an example of a polymorphic value, consider a `map` procedure written in FL:

```
(define map
  (lambda (fn lst)
    (if (null? lst)
        (null)
        (cons (fn (car lst)) (map fn (cdr lst))))))
```

We have seen that aggregate data operators like `map` are a powerful means of composing programs out of reusable, mix-and-match parts. In large part, this power is due to the fact that the same operator works over many types of operands. The `map` procedure, for instance, can be viewed as having an infinite number of possible types, including:

```
(-> ((-> (int) int) (listof int)) (listof int))

(-> ((-> (int) bool) (listof int)) (listof bool))

(-> ((-> (bool) int) (listof bool)) (listof int))

(-> ((-> (bool) bool) (listof bool)) (listof bool))

(-> ((-> ((listof int)) int) (listof (listof int))) (listof int))

(-> ((-> (int) (-> (bool) int)) (listof int))
    (listof (-> (bool) int)))
```

The type of `map` for any particular call depends on the types of its arguments. So, in the call

```
(map (lambda (x) (* x x)) (list 1 2 3)) ,
```

`map` effectively has type

```
(-> ((-> (int) int) (listof int)) (listof int)) ,
```

while in the call

```
(map (lambda (x) (< x 17)) (list 23 13 29)) ,
```

it has the type

```
(-> ((-> (int) bool) (listof int)) (listof bool)) .
```

Other common examples of useful polymorphic procedures include the identity function (`(lambda (x) x)`) and general sorting utilities.

Unfortunately, the type system of FL/X requires the type of values like `map` to be specified where it is created, not where it is called. A programmer wishing to use `map` on different types of arguments must write a different version of `map` for every different set of argument types. For example, here are two FL/X versions of `map` that correspond to the two calls mentioned above:

```
(define map (-> ((-> (int) int) (listof int)) (listof int))
  (lambda ((fn (-> (int) int)) (lst (listof int)))
    (if (null? lst)
        (null int)
        (cons (fn (car lst)) (map fn (cdr lst))))))

(define map (-> ((-> (int) bool) (listof int)) (listof bool))
  (lambda ((fn (-> (int) bool)) (lst (listof int)))
    (if (null? lst)
        (null bool)
        (cons (fn (car lst)) (map fn (cdr lst))))))
```

Except for type information, the two definitions are exactly the same.

Any language like FL/X that forces the programmer to reimplement functionality in order to satisfy the type system thwarts the goal of writing reusable software components. There is a broad class of general-purpose functions and data structures that are inexpressible in such languages due to the shackles of the type system. This lack of expressiveness is indicative of the price that programmers may have to pay for types. In fact, the primary limitations of languages such as PASCAL and C stem from their monomorphic type systems.

Polymorphism can be introduced into a language by generalizing the types of values where they are created, and then specializing these types where the values are used. Reconsider the types of `map` listed above. All of them are instances of a common pattern:

```
(-> ((-> (S) T) (listof S)) (listof T))
```

We would like to be able to declare that `map` has this general type, but then specialize this type (by specifying S and T) wherever `map` is applied.

We embody this approach in a polymorphic language FL/XSP (the "P" stands for "Polymorphism") by adding two new expression constructs and one new type construct to FL/XS:

E ::= ... | `(plambda (`I*`)` E`)` | `(pcall` E T*`)`
T ::= ... | `(forall (`I*`)` T`)`

`(plambda (`I*`)` E`)` creates a polymorphic value that is parameterized over the type variables I*.

`(pcall` E T*`)` instantiates, or **projects**, the type variables of the polymorphic object denoted by E. `pcall` is the "call" that supplies "arguments" to values created by `plambda`.

`(forall (`I*`)` T`)` is the type of a polymorphic value. In the literature, polymorphic types are often written using $\forall$ notation and referred to as "universally quantified." For example, the type of the mapping procedure,

```
(forall (s t) (-> ((-> s t) (listof s)) (listof t)))
```

is typically rendered

$\forall s,t\,.$ `(s` $\rightarrow$ `t)` $\times$ `(listof s)` $\rightarrow$ `(listof t)`

Here is a polymorphic version of `map` written in FL/XSP:

```
(define map (forall (s t)
              (-> ((-> (s) t) (listof s)) (listof t)))
  (plambda (s t)
    (lambda ((fn (-> (s) t)) (lst (listof s)))
      (if ((pcall null? s) lst)
          ((pcall null t))
          ((pcall cons t) (fn ((pcall car s) lst))
                          ((pcall map s t)
                           fn ((pcall cdr s) lst)))))))
```

The `(plambda (s t) ...)` creates a polymorphic value (in this case, a procedure) whose type is abstracted over the type variables `s` and `t`. The `pcall` construct specializes the type of a polymorphic value by filling in the types of these variables:

```
(the (-> ((-> (int) int) (listof int)) (listof int))
     (pcall map int int))

(the (-> ((-> (int) bool) (listof int)) (listof bool))
     (pcall map int bool))
```

Projection allows a polymorphic procedure to be used with different types of arguments:

```
((pcall map int int) (lambda (x) (* x x)) (list 1 2 3))

((pcall map int bool) (lambda (x) (< x 17)) (list 23 13 29))
```

`plambda` and `pcall` have a similar contract to `lambda` and procedure call. But whereas `lambda` and procedure call imply computation at run time, `plambda` and `pcall` imply computation during type checking. That is, `plambda` builds abstractions over types during static analysis; these abstractions are also unwound by `pcall` during static analysis. Every polymorphic value must have its types instantiated (via `pcall`) before it can be used.

FL/XSP requires the explicit projection of polymorphic values via `pcall`. But some polymorphic languages support **implicit projection**, in which the projected types are automatically deduced from context. Implicit projection makes polymorphic programming more palatable by removing some of the overhead of writing explicit types.

In a polymorphic language, general operations on data structures like cells, sums, products, and lists can once again be treated as first-class procedures rather than as special forms. For example, here are the types of the list operators in FL/XSP:

```
(the (forall (t) (-> (t (listof t)) (listof t))) cons)
(the (forall (t) (-> ((listof t)) t))  car)
(the (forall (t) (-> ((listof t)) (listof t))) cdr)
(the (forall (t) (-> ((listof t)) bool)) null?)
(the (forall (t) (-> () (listof t))) null)
```

In FL/XSP, it is even possible to have a polymorphic empty list `nil` with type `(forall (t) (listof t))`. This underscores the fact that polymorphism can be used with all values, not only procedures.

In order to type check expressions involving `plambda` and `pcall`, it is necessary to extend the typing rules, type inclusion rules, and type equivalence as shown in Figure 13.5.

The [$p\lambda$] rule gives a `forall` type to a `plambda`, while the [*project*] rule specifies a beta substitution in the type domain. The [$p\lambda$] rule includes a restriction on the identifiers that `plambda` can abstract over. The restriction uses

Typing Rules

$$\frac{A \vdash E \,:\, T}{A \vdash \texttt{(plambda (}I_1 \ \ldots\ I_n\texttt{)}\ E\texttt{)} \,:\, \texttt{(forall (}I_1 \ \ldots\ I_n\texttt{)}\ T\texttt{)}} \qquad [p\lambda]$$

where $\forall_{i=1}^{n} \,.\, I_i \notin (FTV(\mathit{FreeIds}[\![E]\!])A)$
and E is pure. *[purity restriction]*

$$\frac{A \vdash E \,:\, \texttt{(forall (}I_1 \ \ldots\ I_n\texttt{)}\ T_E\texttt{)}}{A \vdash \texttt{(pcall}\ E\ T_1\ \ldots\ T_n\texttt{)} \,:\, (\texttt{[}T_i/I_i\texttt{]}_{i=1}^{n})\ T_E} \qquad [\mathit{project}]$$

Type Inclusion Rules

$$\frac{(\texttt{[}I_i/J_i\texttt{]}_{i=1}^{n})\ S \ \sqsubseteq\ T, \quad \forall \mathrm{i}\ (I_i \notin \mathit{FreeIds}[\![S]\!])}{\texttt{(forall (}J_1 \ \ldots\ J_n\texttt{)}\ S\texttt{)} \ \sqsubseteq\ \texttt{(forall (}I_1 \ \ldots\ I_n\texttt{)}\ T\texttt{)}} \qquad [\mathit{forall}\text{-}\sqsubseteq]$$

Type Equivalence

$$\frac{(T_1 \ \sqsubseteq\ T_2)\quad (T_2 \ \sqsubseteq\ T_1)}{T_1 \equiv T_2} \qquad [\equiv]$$

Figure 13.5: New rules to handle polymorphism in FL/XSP.

a function FTV(which stands for Free Type Variables). (FTV I^* A) returns the collection of type variables that appear free in the type assignments that A gives to the identifiers I^*. This restriction prohibits a subtle form of variable capture. Consider the following example:

```
(define polytest
  (plambda (t)
    (lambda ((x t))
      (plambda (t) x))))
```

What is the type of `polytest`? To say that it is

```
(forall (t) (-> (t) (forall (t) t)))
```

is incorrect, because the `t` introduced by the outer `plambda` has been captured by the inner one. Because of this name capture, the following expression would not type check even though it should:

```
(pcall ((pcall polytest int) 3) bool)
```

In the [$p\lambda$] rule, we simply outlaw such situations. An implementation could insist programmers enforce the rule, or it could α-rename type variables to guarantee that no capture is possible no matter what names the programmer used.

Note that the rule for type equivalence is broadened to allow equivalence of `forall` types that are the same except for the names chosen for their variables. E.g., this rule allows us to show:

```
(forall (s) (-> (s) s)) ≡ (forall (t) (-> (t) t))
```

▷ **Exercise 13.6** Alyssa P. Hacker wants to remove `error` from the language as a special syntactic construct. She suggests that we add an `error` procedure to the standard environment.

a. Specify the type of the `error` procedure.

b. Illustrate its use by filling in the box in the following example to produce a well-typed expression:

```
(lambda ((x int) (y int))
  (if (= x 0)
      [          ]
      (/ y x)))
```

◁

▷ **Exercise 13.7** Louis Reasoner has had a hard time implementing `letrec` in a call-by-name version of FL/XSP, and has decided to use the fixed point operator **fix** instead. For example, here is the correct definition of factorial in Louis's approach:

```
(let ((fact-gen
        (lambda ((fact (-> (int) int)))
          (lambda ((n int))
            (if (= n 0) 1 (* n (fact (- n 1))))))))
  ((pcall fix (-> (int) int)) fact-gen))
```

Thus, `fix` is a procedure that computes the fixed point of a generating function. Ben Bitdiddle has been called on the scene to help, and he has ensured that Louis's FL/XSP supports recursive types using `recof`.

a. What is the type of `fact-gen`?

b. What is the type of `fix`?

c. What is the type of `((pcall fix (-> (int) int)) fact-gen)`?

Ben Bitdiddle defined the call-by-name version of `fix` to be:

```
(let ((fix (plambda (t)
             (lambda ((f T1))
               ((lambda ((x T2)) (f (x x)))
                (lambda ((x T2)) (f (x x))))))))
    ... fix can be used here ...
   )
```

d. What is T_1?

e. What is T_2?

f. Louis has decided that he would like `(fix` E`)` to be a standard expression in his language. What is the typing rule for `(fix` E`)`? ◁

13.3 Descriptions

The ability to abbreviate types with `tlet` is not sufficiently powerful to express many desirable abstractions. For example, the `define-type` construct in FL/X is too weak to simplify the definition of `make-tree`, the polymorphic version of `make-int-tree` shown in Figure 13.8. Here the tree type expressions cannot be replaced by some globally named type because they are parameterized over the type `t`, which is local to the definition of `make-tree`. What we'd like in this situation is a `lambda`-like construct in the type domain that would allow the construction of **type abstractions**.[2] In this case, we'd like to define a `treeof` operator in the type domain that would allow us to rewrite `make-tree` as:

[2]Not to be confused with abstract types.

```
(define make-int-tree
        (-> ((recof tree
               (oneof
                 (leaf int)
                 (node (recordof (left tree)
                                 (right tree)))))
             (recof tree
               (oneof
                 (leaf int)
                 (node (recordof (left tree)
                                 (right tree))))))
         (recof tree
           (oneof (leaf int)
                  (node (recordof (left tree)
                                  (right tree))))))
        (lambda ((left-branch (recof tree
                                (oneof
                                  (leaf int)
                                  (node (recordof
                                          (left tree)
                                          (right tree))))))
                 (right-branch (recof tree
                                 (oneof
                                   (leaf int)
                                   (node (recordof
                                           (left tree)
                                           (right tree)))))))
          (one
            (recof tree
              (oneof
                (leaf int)
                (node (recordof (left tree)
                                (right tree)))))
            node (record (left left-branch)
                         (right right-branch)))))
```

Figure 13.6: Lack of type abstraction greatly complicates the definition of a `make-int-tree` procedure.

```
(define-type int-tree
  (recof tree
    (oneof (leaf int)
           (node (recordof (left tree) (right tree))))))

(define make-int-tree (-> (int-tree int-tree) int-tree)
  (lambda ((left-branch int-tree) (right-branch int-tree))
    (one int-tree
         node
         (record (left left-branch)
                 (right right-branch)))))
```

Figure 13.7: Type abstractions simplify the definition of `make-int-tree`.

```
(define make-tree (forall (t) (-> ((treeof t) (treeof t)) (treeof t)))
  (plambda (t)
    (lambda ((left-branch (treeof t))
             (right-branch (treeof t)))
      (one (treeof t)
           node
           (record (left left-branch)
                   (right right-branch))))))
```

Note that a type operator such as `treeof` cannot be created by `lambda` or `plambda`; whereas `lambda` creates procedures that map values to values and `plambda` creates procedures that map types to values, a type operator maps types to types. Therefore, a new kind of `lambda` is needed.

In order to address the issues raised by the above examples, we consider a new language FL/XSPD that is a generalized version of FL/XSP. The grammar for FL/XSPD is given in Figures 13.9 and 13.10.[3] Whereas all type expressions in FL/XSP (generated by nonterminal T) denote types, the type expressions in FL/XSPD (generated by nonterminal D) denote **descriptions**. Descriptions encompass not only types, but also operators on types and, in fact, operators on arbitrary descriptions.[4]

The `define-desc` construct can be used to name descriptions globally. Thus,

[3]The grammar for `program` specifies that all `define-desc`s must precede all `define`s. In spite of this, we will assume that these two forms can be freely intermingled in practice. It is easy to imagine that the FL/X parser translates the more liberal form of `program` into the restricted form specified by the grammar.

[4]Descriptions can be extended to include other information as well, such as **effects**, which indicate the allocation, reading, or writing of a mutable data structure. FX uses descriptions with effects to perform static side-effect analysis on programs.

```
(define make-tree
        (forall (t)
          (-> ((recof tree
                 (oneof
                   (leaf t)
                   (node (recordof (left tree)
                                   (right tree)))))
               (recof tree
                 (oneof
                   (leaf t)
                   (node (recordof (left tree)
                                   (right tree))))))
           (recof tree
             (oneof (leaf t)
                    (node (recordof (left tree)
                                    (right tree)))))))
        (plambda (t)
          (lambda ((left-branch (recof tree
                                  (oneof
                                    (leaf t)
                                    (node (recordof
                                            (left tree)
                                            (right tree))))))
                   (right-branch (recof tree
                                   (oneof
                                     (leaf t)
                                     (node (recordof
                                             (left tree)
                                             (right tree)))))))
             (one
               (recof tree
                 (oneof
                   (leaf t)
                   (node (recordof (left tree)
                                   (right tree)))))
               node (record (left left-branch)
                            (right right-branch))))))
```

Figure 13.8: `make-tree`, a version of `make-int-tree` parameterized over the leaf type.

Abstract Syntax:

$P \in$ Program
$I,J \in$ Identifier
$E \in$ Exp
$Y \in$ Symlit
$L \in$ Lit = Unitlit∪Boollit∪Intlit∪Stringlit∪Symlit
$D \in$ Description

E ::= L | I | `(if` E_1 E_2 E_3`)` | `(begin` E_1 E_2`)` | `(the` D E`)`
| `(lambda ((`I D`)*)` E_{body}`)` | `(`E_{proc} E_{args}`*)`
| `(let ((`I E`)*)` E_{body}`)` |`(letrec ((`I D E`)*)` E_{body}`)`
| `(record (`I E`)*)` | `(with` E_{rec} E_{body}`)`
| `(one` D I_{tag} E_{val}`)` | `(tagcase` E_{disc} `(`I_{tag} I_{val} E_{body}`)`$^+$`)`
| `(tagcase` E_{disc} `(`I_{tag} I_{val} E_{body}`)`$^+$ `(else` $E_{default}$`))`
| `(plambda (`I`*)` E`)` | `(pcall` E D`*)`
| `(plet ((`I D`)*)` E_{body}`)` | `(pletrec ((`I D`)*)` E_{body}`)`
| `(error` D Y`)`

D ::= `int` | `unit` | `bool` | `string` | I
| `(->` `(`D_{arg}`*)` D_{body}`)`
| `(recordof (`I_{field} D_{val}`)*)` | `(oneof (`I_{tag} D_{val}`)*)` | `(cellof` D`)`
| `(forall (`I`*)` D`)`
| `(dlambda (`I`*)` D_{body}`)` | `(`D_{rator} D_{rand}`*)` | `(dlet ((`I D`)*)` D`)`
| `(drecof` I D`)` | `(dletrec ((`I D`)*)` D_{body}`)`

Figure 13.9: A kernel grammar for FL/XSPD.

Syntactic Sugar:

P ::= (program E_{body} (define-desc I D)* (define I D E)*)

E ::= ... | (begin E_1 E_2 ... E_n)
| (cond (E_{test} $E_{consequent}$)* (else $E_{default}$))
| (letrec (I D E)* E_{body})

(program E_{body}
(define-desc I_{d_1} D_1) ... (define-desc I_{d_m} D_m)
(define I_{v_1} E_1) ... (define I_{v_n} E_n))
=
(plet (I_{d_1} D_1)
⋮
(plet (I_{d_m} D_m)
(letrec ((I_{v_1} E_1) ... (I_{v_n} E_2))
E_{body})) ...)

The usual desugarings for `begin`, `cond`, and `letrec`.

Figure 13.10: A grammar for FL/XSPD's syntactic sugar.

it acts like the hypothetical `define-type` in the integer binary tree examples above:

```
(define-desc int-tree
  (drecof t
    (oneof (leaf int)
           (node (recordof (left t) (right t))))))
```

The `dlambda` construct denotes a description operator that takes descriptions as arguments and returns a description as a result. Using `dlambda`, the `treeof` type operator suggested above could be written as

```
(define-desc treeof
  (dlambda (leaf-type)
    (drecof tree
      (oneof (leaf leaf-type)
             (node (recordof (left tree) (right tree)))))))
```

This description operator can then be applied to another description. For example, an alternate definition of the `int-tree` type described above is:

```
(define-desc int-tree (treeof int))
```

Since `listof` can be defined in a way similar to `treeof`, `listof` need not be a primitive type constructor in FL/XSPD. It is worth noting in the above examples that `define-desc` can be used to name both types and operators on types.

Because the arguments and results of description operators may include arbitrary descriptions, it is possible to have higher-order description operators. As an example where this power can be put to use, suppose we are defining mapping procedures for several different homogeneous aggregate data structures. In particular, suppose that `listof` and `vectorof` are type constructors for lists and vectors, respectively. Then the types of the procedures `list-map` and `vector-map` would be as follows:

```
list-map : (forall (in-type out-type)
             (-> ((-> (in-type) out-type)
                  (listof in-type))
                 (listof out-type)))

vector-map : (forall (in-type out-type)
               (-> ((-> (in-type) out-type)
                    (vectorof in-type))
                   (vectorof out-type)))
```

Clearly there is a common pattern in the types of the two mapping procedures. We can capture this pattern by creating a description operator `map-type`.

```
(define-desc map-type
  (dlambda (type-constructor)
    (forall (in-type out-type)
      (-> ((-> (in-type) out-type)
           (type-constructor in-type))
          (type-constructor out-type)))))
```

Then the types of `list-map` and `vector-map` can be written more succinctly:

```
list-map : (map-type listof)

vector-map : (map-type vectorof)
```

The `dlet` construct names a description in a local scope. The `drecof` and `dletrec` constructs are used for creating recursive descriptions, such as `treeof`. We have seen versions of these before in FL/XSP, where `drecof` was called `recof` and `dletrec` was called `rectype`. `plet` and `pletrec` are similar to `dlet` and `dletrec` except that they return values rather than descriptions; e.g.,

```
;; DLET returns a description
(dlet ((pairof (dlambda (d1 d2)
                  (recordof (first d1) (second d2)))))
  (pairof int (pairof bool string)))

;; PLET returns a value
(plet ((pairof (dlambda (d1 d2)
                  (recordof (first d1) (second d2)))))
  (the (pairof int (pairof bool string))
       (record (first 3)
               (second (record (first #f)
                               (second "Alyssa"))))))
```

Just as `let` in a typed language cannot be desugared into a `lambda` combination (because type information is lost), it is similarly the case that `plet` cannot be desugared into `plambda` plus `pcall`, nor can `dlet` be desugared into `dlambda` plus a description application.

Intuitively, constructs in the description domain (`define-desc`, `dlambda`, description operator application, `dlet`, and `dletrec`) have a close correspondence with value domain constructs (`define`, `lambda`, value procedure application, `let`, and `letrec`). But how do we formally describe the meanings of the new description expressions that we have introduced? Two new typing rules are needed (see Figure 13.11), but these are for the value-producing `plet` and `pletrec`, not the description producing `dlambda`, `dlet`, `drecof`, and `dletrec`.

$$\frac{A \vdash ([D_i/I_i]_{i=1}^{n})E_{body} \ : \ D_{body}}{A \vdash \texttt{(plet ((}I_1 \ \ D_1\texttt{) ... (}I_n \ \ D_n\texttt{)) }E_{body}\texttt{)} \ : \ D_{body}} \quad [plet]$$

$$\frac{A \vdash ([I_i \ : \ \texttt{(dletrec ((}I_1 \ \ D_1\texttt{) ... (}I_n \ \ D_n\texttt{)) }D_i\texttt{)}/I_i]_{i=1}^{n})E_{body} \ : \ D_{body}}{A \vdash \texttt{(pletrec ((}I_1 \ \ D_1\texttt{) ... (}I_n \ \ D_n\texttt{)) }E_{body}\texttt{)} \ : \ D_{body}} \quad [pletrec]$$

Figure 13.11: New typing rules needed for FL/XSPD.

In order to perform type checking in the presence of general descriptions, we require **description equivalence** rules that tell us when two descriptions are the same. Earlier, we saw some type equivalence rules, including ones for `recof` and `rectype`. We need to extend those rules to handle arbitrary descriptions. Figure 13.12 shows the description equivalence rules that are necessary for FL/XSPD.

Some of the the description equivalence rules correspond to the α, β, and η conversion rules of the lambda calculus. Consider the following examples:

$$D \equiv D \qquad [\textit{reflexivity}]$$

$$\frac{D_1 \equiv D_2}{D_2 \equiv D_1} \qquad [\textit{symmetry}]$$

$$\frac{D_1 \equiv D_2 \;;\; D_2 \equiv D_3}{D_1 \equiv D_3} \qquad [\textit{transitivity}]$$

$$\frac{\forall \text{i} \,.\, (D_i \equiv D_i\,') \;;\; D_B \equiv D_B\,'}{\texttt{(->}\ (D_1\ \ldots\ D_n)\ D_B\texttt{)} \equiv \texttt{(->}\ (D_1\,'\ \ldots\ D_n\,')\ D_B\,'\texttt{)}} \qquad [\texttt{->}\text{-}\equiv]$$

$$\frac{\exists \text{ permuation } \pi \text{ such that } \forall \text{i} \,.\, ((I_i = I_{\prime}\pi(i)) \wedge (D_i \equiv D_{\pi(i)}\,'))}{\texttt{(recordof (}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))} \equiv \texttt{(recordof (}I_1\,'\ D_1\,'\texttt{)}\ \ldots\ \texttt{(}I_n\,'\ D_n\,'\texttt{))}} \qquad [\textit{recordof}\text{-}\equiv]$$

$$\frac{\exists \text{ permuation } \pi \,. \quad \text{such that } \forall \text{i} \,.\, ((I_i = I_{\prime}\pi(i)) \wedge (D_i \equiv D_{\pi(i)}\,'))}{\texttt{(oneof (}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))} \equiv \texttt{(oneof (}I_1\,'\ D_1\,'\texttt{)}\ \ldots\ \texttt{(}I_n\,'\ D_n\,'\texttt{))}} \qquad [\textit{oneof}\text{-}\equiv]$$

$$\frac{D \equiv D'}{\texttt{(refof }D\texttt{)} \equiv \texttt{(refof }D'\texttt{)}} \qquad [\textit{refof}\text{-}\equiv]$$

$$\frac{\forall \text{i} \,.\, (J_i \notin \mathit{FreeIds}[\![D_B]\!])}{\texttt{(forall (}I_1\ \ldots\ I_n\texttt{)}\ D_B\texttt{)} \equiv \texttt{(forall (}J_1\ \ldots\ J_n\texttt{)}\ ([J_i/I_i]_{i=1}^n)D_B\texttt{)}} \qquad [\textit{forall}\text{-}\equiv]$$

$$\frac{\forall \text{i} \,.\, (J_i \notin \mathit{FreeIds}[\![D_B]\!])}{\texttt{(dlambda (}I_1\ \ldots\ I_n\texttt{)}\ D_B\texttt{)} \equiv \texttt{(dlambda (}J_1\ \ldots\ J_n\texttt{)}\ ([J_i/I_i]_{i=1}^n)D_B\texttt{)}} \qquad [\textit{dlambda}\text{-}\equiv]$$

$$\frac{D_P \equiv D_P\,' \;;\; \forall \text{i} \,.\, (D_i \equiv D_i\,')}{(D_P\ D_1\ \ldots\ D_n) \equiv (D_P\,'\ D_1\,'\ \ldots\ D_n\,')} \qquad [\textit{dapply}\text{-}\equiv]$$

$$\texttt{((dlambda (}I_1\ \ldots\ I_n\texttt{)}\ D_B\texttt{)}\ D_1\ \ldots\ D_n) \equiv ([D_i/I_i]_{i=1}^n)D_B \qquad [\textit{dbeta}\text{-}\equiv]$$

$$\frac{\forall \text{i} \,.\, (I_i \notin \mathit{FreeIds}[\![D_P]\!])}{\texttt{(dlambda (}I_1\ \ldots\ I_n\texttt{)}\ \texttt{(}D_P\ I_1\ \ldots\ I_n\texttt{))} \equiv D_P} \qquad [\textit{deta}\text{-}\equiv]$$

$$\texttt{(dlet ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_B\texttt{)} \equiv ([D_i/I_i]_{i=1}^n)D_B \qquad [\textit{dlet}\text{-}\equiv]$$

$$\texttt{(drecof }I\ D\texttt{)} \equiv [D/I]D \qquad [\textit{drecof}\text{-}\equiv]$$

$$\begin{array}{c}\texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_B\texttt{)} \\ \equiv\ ([\texttt{(dletrec ((}I_1\ D_1\texttt{)}\ \ldots\ \texttt{(}I_n\ D_n\texttt{))}\ D_i\texttt{)}/I_i]_{i=1}^n)\ D_B\end{array} \qquad [\textit{dletrec}\text{-}\equiv]$$

Figure 13.12: Description equivalence rules for FL/XSPD.

```
(dlambda (t) t)   ≡   (dlambda (s) s)                          ; alpha

; Assume pairof defined as above
(pairof int bool)   ≡   (recordof (left int) (right bool))   ; beta

(dlambda (s t) (pairof s t))   ≡   pairof                      ; eta
```

It is enlightening to compare the three kinds of abstraction that exist in FL/XSPD:

Abstraction Constructor	Arguments	Results
`lambda`	values	values
`plambda`	descriptions	values
`dlambda`	descriptions	descriptions

It is also possible to imagine a fourth kind of abstraction that takes values as arguments and returns descriptions. These can result in what are called **dependent descriptions** — descriptions that contain values. The array type constructor in Pascal is a simple example of a dependent type; every array type has an integer which indicates the length of the array. Of course, in order to ensure static type checking, the argument values to such an abstraction would have to be statically determinable.

It is disturbing that there are three different constructs that are so similar in intent. The need for the differing constructs arises from the fact that we have maintained a rigid distinction between types and values. In the interest of conceptual economy, some languages, such as PEBBLE, blur the distinction between types and values; in these languages, a single operator constructor can do the job of `lambda`, `plambda`, and `dlambda`. Since types can be treated as values in these languages, however, type checking can generally not be performed statically. Instead, it may have to be interleaved with the execution of the program; in such cases, type checking is effectively dynamic. In fact, in some languages with first-class types, type checking might never even terminate!

13.4 Kinds and Kind Checking: FL/XSPDK

It is important to note that only a subset of descriptions serve as types of values. For example, there is no value that has the type `(dlambda (t) t)` or the type `listof`. Furthermore, many expressions generated by the grammar for descriptions are nonsensical. The description `(int bool)`, for instance, indicates that `int` is being applied to `bool` as a description operator. But since `int` is the type of a value and not a description operator, such an application is not

meaningful. Even the typing rules in Figure 13.11 are problematic as stated; the notation $I : D$ only makes sense if D is a type.

We'd like to ensure that descriptions make sense, both intrinsically and in context. This problem seems an awful lot like the one we already solved via types; showing that `(int bool)` is not a meaningful description is rather similar to showing that `(1 2)` is not a meaningful expression. Just as we had types for expressions, we'd like to have something akin to types for descriptions. These are called **kinds**; kinds are the types of descriptions.

We incorporate the notion of kinds into the language FL/XSPDK, which is just an extension of FL/XSPD. The grammatical changes necessary to extend FL/XSPD into FL/XSPDK are presented in Figure 13.13. The nonterminal K generates kind expressions, which are now required in `plambda`, `forall`, and `dlambda`.

E ::= ... | `(plambda ((`I K`)*)` E`)`

D ::= ... | `(forall ((`I K`)*)` D`)` | `(dlambda ((`I K`)*)` D_{body}`)`

K ::= `type` | `(->> (`K`*)` K`)`

Figure 13.13: The grammar for FL/XSPDK (the parts not listed are the same as in FL/XSPD).

The simplest kind is the base kind `type`. All legal FL/XSP types have kind `type`. For example, the following expressions all have kind `type`:

```
int
(-> (bool) string)
(recordof (name string) (age int))
```

Description operators have a kind that reflects the kinds of the operator's arguments and the kind of the operator's results. `listof`, for example, has kind `(->> (type) type)` because it takes a type and returns a type. Note the double arrow `->>` is used in the kind of a description operator, whereas `->` is used in the type of a procedure. This notational difference is not strictly necessary but serves to emphasize the distinction between the two levels.

A description is **well-kinded** if it can be assigned a kind according to a set of kind checking rules. Kind checking is analogous to type checking; the notation

$$B \vdash D :: K$$

means that kind environment B assigns kind K to description D. Figure 13.14 includes the kind checking rules for FL/XSPD. ϕ_k indicates the empty kind environment.

$$\phi_k \vdash \texttt{unit} :: \texttt{type}, \phi_k \vdash \texttt{bool} :: \texttt{type}, \phi_k \vdash \texttt{int} :: \texttt{type}, \phi_k \vdash \texttt{string} :: \texttt{type}, \phi_k \vdash \texttt{sym} :: \texttt{type} \quad [\textit{literal}]$$

$$[\ldots, I :: K, \ldots] \vdash I :: K \quad [\textit{var}]$$

$$\frac{\forall \text{i} \,.\, (B \vdash D_i :: \texttt{type}) \quad B \vdash D_{body} :: \texttt{type}}{B \vdash \texttt{(->}\ \texttt{(}D_1\ \ldots\ D_n\texttt{)}\ D_{body}\texttt{)} :: \texttt{type}} \quad [\texttt{->}]$$

$$\frac{\forall \text{i} \,.\, (B \vdash D_i :: \texttt{type})}{B \vdash \texttt{(recordof (}I_1\ D_1\texttt{) ... (}I_n\ D_n\texttt{))} :: \texttt{type}} \quad [\textit{recordof}]$$

$$\frac{\forall \text{i} \,.\, (B \vdash D_i :: \texttt{type})}{B \vdash \texttt{(oneof (}I_1\ D_1\texttt{) ... (}I_n\ D_n\texttt{))} :: \texttt{type}} \quad [\textit{oneof}]$$

$$\frac{B \vdash D :: \texttt{type}}{B \vdash \texttt{(refof}\ D\texttt{)} :: \texttt{type}} \quad [\textit{refof}]$$

$$\frac{B[I_1 :: K_1, \ldots, I_n :: K_n] \vdash D_B :: \texttt{type}}{B \vdash \texttt{(forall ((}I_1\ K_1\texttt{) ... (}I_n\ K_n\texttt{))}\ D_B\texttt{)} :: \texttt{type}} \quad [\textit{forall}]$$

$$\frac{B[I_1 :: K_1, \ldots, I_n :: K_n] \vdash D_B :: K_B}{B \vdash \texttt{(dlambda ((}I_1\ K_1\texttt{) ... (}I_n\ K_n\texttt{))}\ D_B\texttt{)} :: \texttt{(->>}\ \texttt{(}K_1\ \ldots\ K_n\texttt{)}\ K_B\texttt{)}} \quad [d\lambda]$$

$$\frac{B \vdash D_P :: \texttt{(->>}\ \texttt{(}K_1\ \ldots\ K_n\texttt{)}\ K_B\texttt{)} \quad \forall \text{i} \,.\, (B \vdash D_i :: K_i)}{B \vdash \texttt{(}D_P\ D_1\ \ldots\ D_n\texttt{)} :: K_B} \quad [\textit{dapply}]$$

$$\frac{\forall \text{i} \,.\, (B \vdash D_i :: K_i) \quad B[I_1 :: K_1, \ldots, I_n :: K_n] \vdash D_B :: K_B}{B \vdash \texttt{(dlet ((}I_1\ D_1\texttt{) ... (}I_n\ D_n\texttt{))}\ D_B\texttt{)} :: K_B} \quad [\textit{dlet}]$$

$$\frac{B[I :: \texttt{type}] \vdash D :: \texttt{type}}{B \vdash \texttt{(drecof}\ I\ D\texttt{)} :: \texttt{type}} \quad [\textit{drecof}]$$

$$\frac{\text{B'} = \text{B}[I_1 :: \texttt{type}, \ldots, I_n :: \texttt{type}] \quad \forall \text{i} \,.\, (B' \vdash D_i :: \texttt{type}) \quad B' \vdash D_B :: K_B}{B \vdash \texttt{(dletrec ((}I_1\ D_1\texttt{) ... (}I_n\ D_n\texttt{))}\ D_B\texttt{)} :: K_B} \quad [\textit{dletrec}]$$

Figure 13.14: Kind checking rules for FL/XSPDK

How do kinds and kind checking interact with types and type checking? Not only must all user-supplied descriptions in an expression be well-kinded, but the descriptions may also be used in a context that requires them to be of a particular kind, typically kind `type`. For example, the descriptions annotating the formal parameters to a `lambda` expression must be of kind `type`. We express these relationships by including constraints on kinds in the antecedents of type checking rules; see, for example, the type checking rules for FL/XSPDK shown in Figure 13.15. The notation

$$A, B \vdash E : D$$

means that type environment A assigns E type D in the presence of kind environment B. (Note that a double colon is used for the "has-kind" relation, whereas a single colon is used for the "has-type" relation.) The rules in Figure 13.15 suggest that the type checking and kind checking processes can be interleaved into a single process that uses both a type environment and a kind environment. Of course, it is also possible to perform kind checking and type checking in separate phases.

Several of the rules in Figure 13.15 extend the type environment with some bindings. Kind checking is used in these situations to guarantee that the extensions bind identifiers to types and not arbitrary descriptions. That is, the notation

$$A[I_1 : D_1 \ \ldots \ I_n : D_n]$$

only makes sense when all of the D_i have kind `type`.

The substitution $([D_i/I_i]_{i=1}^{n})E_B$ in the rule for `plet` is assumed to do the "right thing." That is, only occurrences of I in descriptions (not value expressions) are substituted for. We leave the formal definition of substitution in this situation as an exercise for the reader.

A desirable goal for typechecking is that it should be guaranteed to terminate. Has the introduction of general descriptions compromised this goal? For example, it is possible to imagine description operators which go into infinite loops when applied. Type checking an expression containing such a description might never terminate.

The kind checking and type checking rules we have presented are carefully constructed so that this situation can never occur. The `drecof`, `dletrec`, and `pletrec` constructs are constrained so that the descriptions they introduce must be of kind `type`. With these kind constraints, descriptions in FL/XSPDK have a property called **strong normalization**. This property means that all descriptions can be reduced to normal form in a finite number of steps.[5] Note

[5]There are typed versions of the lambda calculus that have the strong normalization property; in these systems it is impossible to write a Y operator.

$$\frac{\begin{array}{c}\forall \text{i} \ . \ (B \vdash D_i \ :: \ \texttt{type}) \\ A[I_1 : D_1 \texttt{ , } \ \ldots, \ I_n : D_n], B \vdash E_{body} : D_{body}\end{array}}{A, B \vdash \texttt{(lambda ((}I_1 \ D_1\texttt{) } \ldots \texttt{ (}I_n \ D_n\texttt{)) } E_{body}\texttt{) : (-> (}D_i \ \ldots \ D_n\texttt{) } D_{body}\texttt{)}} \quad [\lambda]$$

$$\frac{\begin{array}{c}\forall \text{i} \ . \ (B \vdash D_i \ :: \ \texttt{type}) \\ \text{A'} = \text{A}[I_1 : D_1 \texttt{ , } \ \ldots, \ I_n : D_n] \\ \forall \text{i} \ . \ (A', B \vdash E_i : D_i) \\ A', A \vdash E_{body} : D_{body}\end{array}}{A, B \vdash \texttt{(letrec ((}I_1 \ D_1 \ E_1\texttt{) } \ldots \texttt{ (}I_n \ D_n \ E_n\texttt{)) } E_{body}\texttt{) : } D_{body}} \quad [letrec]$$

$$\frac{\begin{array}{c}A, B[I_1 :: K_1 \ \ldots \ I_n :: K_n] \vdash E : D \\ \forall \text{i} \ . \ (I_i \not\in FTV \ (FreeIds[\![(]\!]E)))\end{array}}{A, B \vdash \texttt{(plambda ((}I_1 \ K_1\texttt{) } \ldots \texttt{ (}I_n \ K_n\texttt{)) } E\texttt{) : (forall ((}I_1 \ K_1\texttt{) } \ldots \texttt{ (}I_n \ K_n\texttt{)) } D\texttt{)}} \quad [p\lambda]$$

$$\frac{\begin{array}{c}A, B \vdash E : \texttt{(forall ((}I_1 \ K_1\texttt{) } \ldots \texttt{ (}I_n \ K_n\texttt{)) } D_{body}\texttt{)} \\ \forall \text{i} \ . \ (B \vdash D_i \ :: \ K_i)\end{array}}{A, B \vdash \texttt{(pcall } E \ D_1 \ \ldots \ D_n\texttt{) : (}[D_i/I_i]_{i=1}^{n}\texttt{) } D_{body}} \quad [project]$$

$$\frac{A, B \vdash ([D_i/I_i]_{i=1}^{n})E_B : D_{body}}{A, B \vdash \texttt{(plet ((}I_1 \ D_1\texttt{) } \ldots \texttt{ (}I_n \ D_n\texttt{)) } E_{body}\texttt{) : } D_{body}} \quad [plet]$$

$$\frac{\begin{array}{c}\forall \text{i} \ . \ (B \vdash \texttt{(dletrec ((}I_1 \ D_1\texttt{) } \ldots \texttt{ (}I_n \ D_n\texttt{)) } D_i\texttt{) :: type}) \\ A' = A[I_1 : \texttt{(dletrec ((}I_1 \ D_1\texttt{) } \ldots \texttt{ (}I_n \ D_n\texttt{)) } D_1\texttt{) } \ldots \ I_n : \texttt{(dletrec ((}I_1 \ D_1\texttt{) } \ldots \texttt{ (}I_n \ D_n\texttt{)) } D_n\texttt{)}] \\ A', B[I_1 :: \texttt{type } \ldots \ I_n :: \texttt{type}] \vdash E_{body} : D_{body}\end{array}}{A, B \vdash \texttt{(pletrec ((}I_1 \ D_1\texttt{) } \ldots \texttt{ (}I_n \ D_n\texttt{)) } E_{body}\texttt{) : } D_{body}} \quad [pletrec]$$

Figure 13.15: Type checking rules for FL/XSPDK. Rules not shown are analogous to those in FL/XSP.

that strong normalization implies that it is impossible to write the Y operator in the description language. Intuitively, this is due to the simplicity of the kind system; there are no recursive kind constructs. Thus, in FL/XSPDK, it is possible to write Y as an expression, but not as a description.

If you're wondering whether it's possible for kinds themselves have something similar to types or kinds, the answer is yes. The types of kinds are sometimes called **sorts**; all kind expressions we have examined are of sort `kind`. But it is possible to consider operators on kinds — kind operators — that would have more interesting sorts. Similarly, we could construct a "typing" system for sorts that distinguished sorts from operators on sorts. Clearly this process could be repeated ad infinitum (and ad nauseum!), giving rise to an infinite "tower" of typing systems. However, only the lowest levels of the tower — types and kinds — are useful in most practical situations.

Reading

The polymorphic typed lambda calculus was invented by Girard and later reinvented by Reynolds [Rey74]. See [Hue90] for some papers on the polymorphic lambda calculus.

For work on types in object-oriented programming, see [GM94].

Chapter 14

Type Reconstruction

The faculty of deduction is certainly contagious ...

— *Sherlock Holmes in* The Problem of Thor Bridge
by Sir Arthur Conan Doyle

14.1 Introduction

In the variants of FL/X that we've studied so far, it is necessary to specify explicit type information in certain situations. All variables introduced by a `lambda` must be explicitly typed, for instance. Not all type information needs to be explicitly declared, however. For example, the return type of a procedure need not be explicitly declared.

What determines the placement of explicit type information in a language? That is, why does some type information have to be provided while other type information can be elided? The answer to this question lies in the structure of the type checker. As noted earlier, a simple type checker has the structure of an evaluator. Consider the type checking of a `lambda` expression. When entering a `lambda` expression, the type checker has no information about the types of the formal parameters; these must be provided explicitly. However, once the types of the formals are known, it is easy for the type checker to determine the type of the body, so this information need not be declared.

Could a more sophisticated type checker do its job with even less explicit information? Certainly, programmers can reason proficiently about type information in many programs where there are no explicit types at all. Such reasoning is important because understanding the type of an expression, especially one

that denotes a procedure, is often a major step in figuring out what purpose the expression serves in the program. As an example of this kind of type reasoning, consider the following expression:

```
(lambda (f x y)
  (if (f x y) (f 3 y) (f x "twenty-three"))))
```

By studying the various ways in which `f`, `x`, and `y` are used in the body of the above `lambda` expression, we can piece together a lot of information about the types of these variables. The application `(f x y)`, for example, returns a boolean because it is used as the predicate in an `if` expression. Thus, `f` is a procedure of two arguments that returns a boolean. Since both branches of the `if` expression are calls to `f`, we know that the procedure created by the `lambda` expression must also return a boolean. In fact, looking at the consequent and alternative of the `if`, we can even say more about `f`: its first argument is a number and its second argument is a string. Thus, `x` must be a number and `y` must be a string.

There is no reason that a program cannot carry out the same kinds of reasoning exhibited above. Automatically computing the type of an expression that does not contain type information is known as **type reconstruction** or **type inference**. Type reconstruction is more complicated than type checking because type reconstruction must operate properly without programmer supplied type assertions.

Type reconstruction is the formalization of the kind of reasoning seen in the example above. A type reconstruction algorithm is an automatic way of determining the types of an expression (and all the subexpressions along the way). We can think of the different subexpressions in the above example as specifying constraints on the types of the expressions. It is possible to view these constraints as a set of simultaneous type equations that restrict the type of an expression. If these equations cannot be solved, then the expression is not well-typed. If these equations can be solved, then the **most general** typing for the expression results. In the event that several types may be assigned to an expression, then the most general type is the type, T, such that all other possible types are substitution instances of T. I.e., for any type, S, the expression may have, there is a substitution that can be applied to the most general type to get S. The type system of a type reconstructed language is usually designed so that there is a unique most general type for every typable expression. Most general types are often called **principal types**.

Consider the `lambda` expression studied above. Suppose that[1]

[1]Here we denote unknowns in the equations by names prefixed with a question mark, to

- `?l` is the type of the result of evaluating the `lambda` expression.
- `?i` is the type of the result of evaluating the `if` expression.
- `?f` is the type of `f`.
- `?x` is the type of `x`.
- `?y` is the type of `y`

Then the equations that are implied by the expression are

```
?l  =  (-> (?f ?x ?y) ?i)
?f  =  (-> (?x ?y) ?a)
?a  =  bool
?f  =  (-> (int ?y) ?b)
?f  =  (-> (?x string) ?c)
?b  =  ?c
?i  =  ?b
```

A solution to the above equations yields the following variable bindings:

```
?a  =  ?b = ?c = ?i = bool
?x  =  int
?y  =  string
?f  =  (-> (int string) bool)
?l  =  (-> ((-> (int string) bool) int string) bool)
```

Note that a system of type equations need not always have the neat form of solution indicated by the example. For example, the system associated with

```
(lambda (f x y)
  (if (f x y) (f 3 y) (f y "seven")))
```

has no solution since it is overconstrained: the `int` and `string` types are disjoint. On the other hand, the system may be underconstrained, as in the following perturbation of the example:

distinguish them from variables in the language.

```
(lambda (h x y)
  (if (h x y) (h x y) (h x "seven")))
```

In this case, the type of `x` is unknown, and the type deduced for the expression is

```
(-> ((-> (?x string) bool) ?x string) bool)
```

The appearance of an unknown type variable in this type is the way that this particular type language indicates the potential for polymorphism. We will see below under what conditions such a type is viewed as polymorphic. In other notations we have seen, a polymorphic type would be expressed as:

```
(forall (t) (-> ((-> (t string) bool) t string) bool))
```

or

$$\forall t\,.\ \texttt{((string t)} \rightarrow \texttt{bool)} \times \texttt{t} \times \texttt{string} \rightarrow \texttt{bool}$$

14.2 A Language with Type Reconstruction: FL/R

In this section, we'll consider issues in type reconstruction for a variant of FL called FL/R. The grammar for FL/R is given in Figure 14.1. Note that there are no explicit type declarations in the expressions of the language.

Figure 14.2 contains the typing rules for FL/R. The rules for literals, conditionals, abstractions, and applications are similar to the ones for FL/X. The only difference is in the `lambda` rule. Whereas the FL/X typing rule for `lambda` uses the explicit type declarations for each of the variables, the `lambda` rule in FL/R "guesses" the types of the variables, and then checks to see that its guess is correct. The typing rules do not explain how these guesses are made. The details of guessing will be specified by the reconstruction algorithm presented in Section 14.4 below.

The typing rules for `let`, `letrec` and variables contain some new ideas. The motivation for these concepts is that we'd like type reconstruction to be able to reconstruct polymorphic types, at least in some simple cases. As an example of where we'd like to infer a polymorphic type, consider:

```
(let ((f (lambda (x) x)))
  (if (f #t) (f 1) (f 2)))
```

Here, we would like `f` to have the type `(-> (bool) bool)` when applied to `#t`, and the type `(-> (int) int)` when applied to `1` and `2`. If we required each variable to have only one type associated with it, this kind of polymorphic behavior would not be allowed in the language.

$P \in \text{Program}_{FL/R}$ $I \in \text{Identifier}_{FL/R}$ = usual identifiers
$E \in \text{Exp}_{FL/R}$ $B \in \text{Boollit}_{FL/R} = \{\texttt{\#t}, \texttt{\#f}\}$
$AB \in \text{Abstraction}_{FL/R}$ $N \in \text{Intlit}_{FL/R} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$L \in \text{Lit}_{FL/R}$ $O \in \text{Primop}_{FL/R}$
$T \in$ Type
$TS \in$ Type-schema

P ::= `(flr (`I_{fml}`*) `E_{body} `(define `I_{name} E_{defn}`)*)`

E ::= L | I | `(error `I`)`
| `(if `E_{test} E_{then} E_{else}`)` | `(begin `E^+`)`
| `(lambda (`I_{fml}`*) `E_{body}`)` | `(`E_{rator} E_{rand}`*)` | `(primop `O_{op} E_{arg}`*)`
| `(let ((`I_{name} E_{defn}`)*) `E_{body}`)` | `(letrec ((`I_{name} E_{defn}`)*) `E_{body}`)`

L ::= `#u` | B | N | `(symlit `I`)`

$O_{FL/R}$ ::= `+` | `-` | `*` | `/` | `%` [Arithmetic ops]
| `<=` | `<` | `=` | `!=` | `>` | `>=` [Relational ops]
| `not` | `band` | `bor` [Logical ops]
| `pair` | `fst` | `snd` [Pair ops]
| `cons` | `car` | `cdr` | `null` | `null?` [List ops]
| `cell` | `^` | `:=` [Mutable cell ops]

T ::= `unit` | `int` | `bool` | `sym` [Base Types]
| I [Type Variable]
| `(->` (`$T$`*) T_{body}`)` [Arrow Type]
| `(pairof` T_1 T_2`)` [Pair Type]
| `(listof` T`)` [List Type]
| `(cellof` T`)` [Cell Type]

TS ::= `(generic (`I`*) `T`)` [Type Schema]

Figure 14.1: Grammar for FL/R

$$\vdash \texttt{\#u} : \texttt{unit} \qquad [unit]$$

$$\vdash B : \texttt{bool} \qquad [bool]$$

$$\vdash N : \texttt{int} \qquad [int]$$

$$\vdash \texttt{(symbol } I\texttt{)} : \texttt{sym} \qquad [symbol]$$

$$[\ldots, I\texttt{:}T, \ldots] \vdash I : T \qquad [var]$$

$$[\ldots, I\texttt{:(generic (}I_1 \ \ldots \ I_n\texttt{) } T_{body}\texttt{)}, \ldots] \vdash I : ([T_i/I_i]_{i=1}^{n})T_{body} \qquad [genvar]$$

$$A \vdash \texttt{(error } I\texttt{)} : T \qquad [error]$$

$$\frac{A \vdash E_{test} : \texttt{bool} \quad ; \quad A \vdash E_{con} : T \quad ; \quad A \vdash E_{alt} : T}{A \vdash \texttt{(if } E_{test} \ E_{con} \ E_{alt}\texttt{)} : T} \qquad [if]$$

$$\frac{\forall_{i=1}^{n} \ . \ A \vdash E_i : T_i}{A \vdash \texttt{(begin } E_1 \ \ldots \ E_n\texttt{)} : T_n} \qquad [begin]$$

$$\frac{A[I_1 : T_1\texttt{,} \ \ldots\texttt{,} \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{) } E_{body}\texttt{)} : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T_{body}\texttt{)}} \qquad [\lambda]$$

$$\frac{\begin{array}{c} A \vdash E_{rator} : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T_{body}\texttt{)} \\ \forall_{i=1}^{n} \ . \ A \vdash E_i : T_i \end{array}}{A \vdash \texttt{(}E_{rator} \ E_1 \ \ldots \ E_n\texttt{)} : T_{body}} \qquad [apply]$$

$$\frac{\begin{array}{c} A_{standard} \vdash O : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T\texttt{)} \\ \forall_{i=1}^{n} \ . \ A \vdash E_i : T_i \end{array}}{A \vdash \texttt{(primop } O \ E_1 \ \ldots \ E_n\texttt{)} : T} \qquad [primop]$$

$$\frac{\begin{array}{c} \forall_{i=1}^{n} \ . \ A \vdash E_i : T_i \\ A[I_1\texttt{:}\mathtt{GenPure}\texttt{(}E_1\texttt{,} \ T_1\texttt{,} \ A\texttt{)},\ldots,I_n\texttt{:}\mathtt{GenPure}\texttt{(}E_n\texttt{,} \ T_n\texttt{,} \ A\texttt{)}] \vdash E_{body} : T_{body} \end{array}}{A \vdash \texttt{(let ((}I_1 \ E_1\texttt{) } \ldots \ \texttt{(}I_n \ E_n\texttt{))) } E_{body}\texttt{)} : T_{body}} \qquad [let]$$

$$\frac{\begin{array}{c} \forall_{i=1}^{n} \ . \ A[I_1 : T_1\texttt{,} \ \ldots\texttt{,} \ I_n : T_n] \vdash E_i : T_i \\ A[I_1\texttt{:}\mathtt{GenPure}\texttt{(}E_1\texttt{,} \ T_1\texttt{,} \ A\texttt{)},\ldots,I_n\texttt{:}\mathtt{GenPure}\texttt{(}E_n\texttt{,} \ T_n\texttt{,} \ A\texttt{)}] \vdash E_{body} : T_{body} \end{array}}{A \vdash \texttt{(letrec ((}I_1 \ E_1\texttt{) } \ldots \ \texttt{(}I_n \ E_n\texttt{))) } E_{body}\texttt{)} : T_{body}} \qquad [letrec]$$

$$\frac{\begin{array}{c} A_{standard}[I_1 : T_1\texttt{,} \ \ldots\texttt{,} \ I_n : T_n] \vdash \\ \texttt{(letrec ((}I_{d_1} \ E_1\texttt{) } \ldots \ \texttt{(}I_{d_k} \ E_k\texttt{)) } E_{body}\texttt{)} : T \end{array}}{\begin{array}{c} \vdash_{prog} \texttt{(flr (}I_1 \ \ldots \ I_n\texttt{) } E_{body} \ \texttt{(define } I_{d_1} \ E_1\texttt{) } \ldots \ \texttt{(define } I_{d_k} \ E_k\texttt{))} \\ : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T\texttt{)} \end{array}} \qquad [program]$$

$$Gen(T, A) = \texttt{(generic (}I_1 \ \ldots \ I_n\texttt{) } T\texttt{)}, \text{where } \{I_i\} = FTV(T) - FTE(A)$$

$$GenPure(E, T, A) = \begin{array}{l} Gen(T, A) \text{ if } E \text{ is pure} \\ T \text{ otherwise} \end{array}$$

Figure 14.2: Typing rules for FL/R.

In order to handle this simple polymorphism, we introduce the notion of a **type schema** (*TS* in Figure 14.1). A type schema is a pattern for a type expression that is abstracted over variables; the schema can be instantiated by binding the variables to particular types. For example, a type schema for the identity procedure is

```
(generic (t) (-> (t) t))
```

Type schemas for various list operations are shown below:

```
cons:   (generic (t) (-> (t (list-of t)) (list-of t)))
car:    (generic (t) (-> ((list-of t)) t))
cdr:    (generic (t) (-> ((list-of t)) (list-of t)))
null?   (generic (t) (-> ((list-of t)) bool))
null    (generic (t) (-> () (list-of t)))
```

Unlike the `forall` type of FL/X, a type schema cannot appear as a subexpression of a type expression. That is, according to the grammar for type schemas, `generic` can only appear once, at the outermost level. A type schema thus represents types that are universally quantified over a set of variables.

The reason for this restriction on type schemas is that type reconstruction is greatly complicated in the more general case. In some cases, it is unknown whether it can even be accomplished; other situations involving general polymorphic types have been proven undecidable. Reconstruction using type schemas, though, is decidable. Type reconstruction based on type schemas is usually called **Hindley-Milner** type reconstruction, after its inventors.

FL/R's use of type schemas means that certain meaningful FL programs are not well-typed in FL/R, even if they could be assigned types in FL/X. Consider the following program:

```
(lambda (f)
  (if (f #t) (f 1) (f 2)))
```

Intuitively, the type for this procedure is something like

```
(-> ((generic (t) (-> (t) t))) int)
```

But this is not a legal type or type schema. The closest legal type schema we could make would be:

```
(generic (t) (-> ((-> (t) t)) int))
```

Although this bears some similarity to the desired type, it is not correct. For example, instantiating the schema with `t` bound to `int` would lead us to believe that the above procedure could take the integer successor procedure as its argument. But this yields a type error, since we shouldn't be able to apply the successor procedure to `#t`.

Type schemas are introduced into the language via `let` and `letrec`. The idea is that any new type variables introduced by the bindings (free type variables that do not appear already in the type environment, A) can be viewed as generic. For example,

```
(let ((f (lambda (x) x)))
  (if (f true) (f 3) (f 4)))
```

will now type check, because the type checker will guess that `(lambda (x) x)` has the type `(-> (t13) t13)` where `t13` is a newly minted type variable. Since this type variable is not constrained by information imported into the expression `(lambda (x) x)`, it can be generalized, and thus `f` can have a different type each time it is used (via the [*genvar*]). How these names are guessed will become clearer when we present a type reconstruction algorithm below.

One way to view this type of polymorphism is to imagine that the right hand sides of the `let` or `letrec` bindings are substituted for the identifiers to which they are bound in the body. This would allow the expression to have different types for any new type variables at each use. However, this transformation is only legitimate if the expressions are referentially transparent as discussed in Section 8.2.5.

The [*let*] and [*letrec*] rules restrict polymorphic values to be pure expressions, just as we restricted the body of a `plambda` to be pure, and for the same reason. We will see a more general way to introduce effect restrictions in Chapter 16, but for now, we will insist that E is pure only if it is a syntactic value, i.e., E is a literal, variable reference (note that FL/R does not have mutable variables), a `lambda` expression, or an `if`/`let`/`letrec` all of whose components are syntactic values. In other words, applications or compound expressions (except `lambda`) is not a syntactic value.

14.3 Unification

In order to solve type equations, we will use the **unification** method due to Robinson. Unification takes two types and attempts to find a substitution for special unification variables in the two types (here prefixed with `?`) such that the two expressions are equal. A substitution is a structure that represents constraints between unification variables. It is like a type environment in that it contains bindings of variables (in this case, unification variables, not expression variables) to types. These types may contain other unification variables. A substitution can be applied to a type or expression; this returns a new type or expression in which each unification variable has been replaced by the element to which it is bound in the substitution. A substitution S is said to be **more**

general than a substitution S' (written $S > S'$) if there exists an S'' such that $S' = (S'' \circ S)$.

More formally, a unification algorithm U unifying types T_1 and T_2 with respect to a substitution S (written $U(T_1, T_2, S)$) produces the most general substitution $S' < S$ such that $(S'\ T_1) = (S'\ T_2)$, where the notation $(S\ T)$ designates the result of applying substitution S to the type T. Unification can of course fail. We will represent the result of a failing unification by the token `fail`.

Here are some examples of unification. (Assume that S_0 is the empty substitution, i.e., one not specifying constraints on any variables.)

$$U(\texttt{(?x ?y)}, \texttt{(int ?x)}, S_0) = \{\texttt{?x = int, ?y = int}\}$$

$$U(\texttt{(-> (int) bool)}, \texttt{(-> (bool) ?x)}, S_0) = \texttt{fail}$$

$$U(\texttt{(-> (?x) (-> (?x) ?y))}, \texttt{(-> (int) ?z)}, S_0)$$
$$= \{\texttt{?x = int, ?z = (-> (int) ?y)}\}$$

Note in the examples how the same variable can be used in the two expressions being unified to express a constraint between them. For example, in $U(\texttt{(?x ?y)}, \texttt{(int ?x)}, S_0)$ the variable `?x` is used to say "the first element of the first pair must be the same as the second element of the second pair." This idea is also very important in logic programming, and, in fact, unification lies at the heart of both logic programming and type reconstruction.

14.4 A Type Reconstruction Algorithm

We now present a type reconstruction algorithm similar to one developed by Milner. This algorithm is the basis of type reconstruction in FX, ML, and HASKELL.

This is used (via input) both by handouts 41 and 44 We shall use the following notation to describe the steps of the algorithm:

$$(R[\![E]\!]\ A\ S) = \langle T, S' \rangle$$

The way to read this notation is "Reconstructing the type of expression E in type environment A with respect to substitution S yields the type T and the new substitution S'." Reconstruction may not always succeed; if it is not possible to perform type reconstruction, then

$$(R[\![E]\!]\ A\ S) = \texttt{fail}$$

The algorithm is defined such that the following relationship is satisfied:

$$(\textit{subst-in-type-env}\ S'\ A) \vdash E : (S'\ T)$$

where $(S'\ T)$ means the result of applying the substitution S' to the type T, and *subst-in-type-env* takes a substitution and a type environment, and returns a new type environment in which the substitution has been applied to all the types bound in the environment.

The type of an FL/R expression E can be found by the function *reconstruct*,

$$\mathit{reconstruct}(E) = (\textbf{let}\ \langle T, S\rangle\ \textbf{be}\ \ (R[\![E]\!]\ \ A_0\ \ S_0)\ \textbf{in}\ (S\ \ T))$$

where A_0 is the standard type environment (presumably containing the types for all names in the standard value environment) and S_0 is the empty substitution. Recall that the metalanguage notation

$$\textbf{let}\ \langle T, S\rangle\ \textbf{be}\ \ E_{val}\ \textbf{in}\ E_{body}$$

is a destructuring form of **let** . We assume for simplicity of presentation that **let** propagates failure as well. That is, if the result of evaluating E_{val} is `fail`, then **let** returns `fail` immediately, without evaluating E_{body}.

Figures 14.3 and 14.4 present the algorithm. The handling of literals, conditionals, abstractions, and applications are fairly straightforward. The handling of `let` and `letrec` is complicated by the desire to handle polymorphism. The functions *RgenPure* and *Rgen* are like the *GenPure* and *Gen* functions encountered before, except that they take a substitution as an additional argument. This substitution is applied to both the type and the type environment:

$$\mathit{RgenPure}(E,\ T,\ A,\ S) = \begin{cases} T & \text{if } E \text{ is not pure} \\ \mathit{Rgen}(T, A, S) & \text{otherwise} \end{cases}$$

$\mathit{Rgen}(T,\ A,\ S)$
$\quad = \mathit{Gen}((S\ \ T),$ `(subst-in-type-env` $S\ \ A$`)`$)$
$\quad =$ `(generic (`$J_1\ \ldots\ J_n$`)` $(S\ \ T)),$
$\qquad$ where $\{J_i\} = \mathit{FTV}((S\ \ T)) - \mathit{FTE}((\mathit{subst\text{-}in\text{-}type\text{-}env}\ S\ A))$

Here, *FTV* gives the free type variables of a type expression (i.e., those type variables that are not bound by `generic`), and *FTE* gives the free type variables of a type environment (i.e., all type variables that appear free in some type bound in the environment). A type variable J is a name prefixed with a `?`.

14.5 Discussion

Milner proved two theorems about his type reconstruction algorithm:

1. The semantic soundness theorem states that if an expression is well-typed (by his definition of well-typed, which is expressed as a set of reconstruction rules), then the expression cannot encounter a dynamic type error.

$(R[\![\texttt{\#u}]\!]\ A\ S) = \langle \texttt{unit}, S\rangle$
$(R[\![B]\!]\ A\ S) = \langle \texttt{bool}, S\rangle$
$(R[\![N]\!]\ A\ S) = \langle \texttt{int}, S\rangle$
$(R[\![S]\!]\ A\ S) = \langle \texttt{string}, S\rangle$
$(R[\![\texttt{(symbol } I\texttt{)}]\!]\ A\ S) = \langle \texttt{sym}, S\rangle$
$(R[\![I]\!]\ A[\ldots, I\texttt{:}T, \ldots]\ S) = \langle T, S\rangle$
$(R[\![I]\!]\ A[\ldots, I\texttt{:(generic (}I_1\ \ldots\ I_n\texttt{) }T\texttt{)}, \ldots]\ S) = \langle([\texttt{?v}_i/I_i]_{i=1}^{n})T, S\rangle$,
where $\texttt{?v}_i$ are fresh
$(R[\![I]\!]\ A\ S) = \texttt{fail}$,
$\texttt{where } I \texttt{ unbound in } A$
$(R[\![\texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!]\ A\ S) =$
let $\langle T_{test}, S_{test}\rangle$ **be** $(R[\![E_{test}]\!]\ A\ S)$ **in**
let $S_{test}{}'$ **be** $U(T_{test}, \texttt{bool}, S_{test})$ **in**
let $\langle T_{con}, S_{con}\rangle$ **be** $(R[\![E_{con}]\!]\ A\ S_{test}{}')$ **in**
let $\langle T_{alt}, S_{alt}\rangle$ **be** $(R[\![E_{alt}]\!]\ A\ S_{con})$ **in**
let $S_{alt}{}'$ **be** $U(T_{con}, T_{alt}, S_{alt})$ **in**
$\langle T_{alt}, S_{alt}{}'\rangle$
$(R[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) }E_{body}\texttt{)}]\!]\ A\ S) =$
let $\langle T_{body}, S_{body}\rangle$ **be** $(R[\![E_{body}]\!]\ A[I_1\texttt{:?v}_1\ \ldots\ I_n\texttt{:?v}_n]\ S)$ **in**
$\langle\texttt{(->(?v}_1\ \ldots\ \texttt{?v}_n\texttt{) }T_{body}\texttt{)}, S_{body}\rangle$,
where $\texttt{?v}_i$ are fresh
$(R[\![\texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)}]\!]\ A\ S) =$
let $\langle T_{rator}, S_{rator}\rangle$ **be** $(R[\![E_{rator}]\!]\ A\ S)$ **in**
let $\langle T_1, S_1\rangle$ **be** $(R[\![E_1]\!]\ A\ S_{rator})$ **in**
$\vdots$
let $\langle T_n, S_n\rangle$ **be** $(R[\![E_n]\!]\ A\ S_{n-1})$ **in**
let S_{final} **be** $U(T_{rator}, \texttt{(->(}T_1\ \ldots\ T_n\texttt{) ?v)}, S_n)$ **in**
$\langle\texttt{?v}, S_{final}\rangle$,
where ?v is fresh.

Figure 14.3: Type reconstruction algorithm for FL/R, Part I.

$$
\begin{array}{l}
(R[\![\texttt{(let ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{)) }E_{body}\texttt{)}]\!]\ A\ S) = \\
\quad \textbf{let}\ \langle T_1, S_1\rangle\ \textbf{be}\ (R[\![E_1]\!]\ A\ S)\ \textbf{in} \\
\qquad \vdots \\
\quad \textbf{let}\ \langle T_n, S_n\rangle\ \textbf{be}\ (R[\![E_n]\!]\ A\ S_{n-1})\ \textbf{in} \\
\qquad R(E_{body}, \\
\qquad\quad A[I_1 : \texttt{RgenPure}(E_1,\ T_1,\ A,\ S_n),\ \ldots,\ I_n : \texttt{RgenPure}(E_n,\ T_n,\ A,\ S_n)], \\
\qquad\quad S_n) \\
(R[\![\texttt{(letrec ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{)) }E_{body}\texttt{)}]\!]\ A\ S) = \\
\quad \textbf{let}\ A_1\ \textbf{be}\ A\texttt{[}I_1\texttt{:?v}_1\texttt{, ..., }I_n\texttt{:?v}_n\texttt{]}\ \textbf{in} \\
\quad\ \textbf{let}\ \langle T_1, S_1\rangle\ \textbf{be}\ (R[\![E_1]\!]\ A_1\ S)\ \textbf{in} \\
\qquad \vdots \\
\quad\ \textbf{let}\ \langle T_n, S_n\rangle\ \textbf{be}\ (R[\![E_n]\!]\ A_1\ S_{n-1})\ \textbf{in} \\
\quad\ \textbf{let}\ S_B\ \textbf{be}\ U(\texttt{(?v}_1\ \ldots\ \texttt{?v}_n\texttt{)}, \texttt{(}T_1\ \ldots\ T_n\texttt{)}, S_n)\ \textbf{in} \\
\qquad R(E_{body}, \\
\qquad\quad A[I_1 : \texttt{RgenPure}(E_1,\ T_1,\ A,\ S_B),\ \ldots,\ I_n : \texttt{RgenPure}(E_n,\ T_n,\ A,\ S_B)], \\
\qquad\quad S_B) \\
\text{where } \texttt{?v}_i \text{ are fresh}
\end{array}
$$

Figure 14.4: Type reconstruction algorithm for FL/R, Part II.

More generally, he showed that, if with respect to type environment A, an expression E has static type T (i.e., $A \vdash E : T$), then the denotation of E with respect to an environment e that respects A has type T (i.e., $(\mathcal{E}[\![E]\!]\ e) : T$). An environment e respects type environment A if for all variables `x` in A, $(e$ `x`$)$ has type $(A$ `x`$)$.

2. The syntactic soundness theorem states that if the type reconstruction algorithm R above discovers a type for an expression E, then the type it discovers is a provable type of E, and thus E is well-typed.

Of course there are limitations to type reconstruction. For example, in Milner's type system the following expressions are not well-typed:

```
(lambda (x) (x x))                ; Self-application

(lambda (f) (cons (f 1) (f #t))) ; First-class polymorphic values
```

The second restriction appears to be more severe than the first. In fact, there is presently no way of giving an independent characterization of the expressions that are not well-typed in Milner's system.

It is important to note that the kinds of types we can infer are closely related to the kinds of inference rules that we are using. We are often willing to reduce

the power of our type inference system (in terms of the range of types that can be inferred) for an increased simplicity in the type inference rules. Simpler rules are easier to prove sound; they generally imply an easier-to-implement type-inference algorithm as well.

For example, Milner's type-inference algorithm does not handle subtyping in any way. As exhibited by the above example, we assume that the types of procedure arguments in different calls within the same type environment must be exactly the same; we will not search for some "least upper bound" of the two in some type lattice. Similarly, we assume that both branches of an `if` expression have exactly the same type; no subtypting is allowed here either. This means that the type equation solver need only deal with strict equality constraints. The standard unification algorithm is very good at solving such equality constraints. But the minute subtyping is added to type inference, inequality constraints are introduced and the standard unification algorithm doesn't work any more. This doesn't necessarily mean that inference with subtyping is impossible; it's just a lot more complex.

A key advantage of Milner's approach to type inference is that it is decidable. If we try to make a type inference system more powerful by including features like first-class polymorphism, type inference may become undecidable. Here's a table of what is currently known about the decidability of various type inference schemes:

Type of Inference	First-class Polymorphism?	User Declarations	Decidability
Hindley-Milner	no	optional	decidable
Full 2nd-order λ-calculus	yes	none	undecidable
Full 2nd-order λ-calculus	yes	optional	undecidable
Full 2nd-order λ-calculus	yes	non-ML types declared	decidable
Full 2nd-order λ-calculus	yes	required	decidable

▷ **Exercise 14.1** After Alyssa P. Hacker finished her semantics for `producer` and `consumer` from Exercise 9.5, she realized that she also needed to specify typing rules for the new language constructs. Alyssa started by adding the new `producer-of` type to describe producer values:

T ::= ... | (`producer-of` T_{yield} T_{return})

A producer of type (`producer-of` T_{yield} T_{return}) yields values of type T_{yield}; if no more values are to be yielded, it returns a value of type T_{return}.

a. What is the type of the identifier `sum` defined in the following example?

```
(define sum
  (lambda (prod-fn)
    (let ((ans (cell 0)))
      (if (consume (prod-fn #u) n
            (cell-set! ans (+ n (cell-ref ans))))
          (cell-ref ans)
          -1))))
```

b. What are the typing rules for the `producer` and `consumer` constructs?

c. What are the type reconstruction algorithm clauses for `producer` and `consumer`?

◁

▷ **Exercise 14.2** Sam Antix realizes that FL/R supports only homogeneous compound datatypes. He decides to extend FL/R with heterogeneous values called **tuples**. An n-tuple is a value that contains n component values, all of which may have different types. Sam extends the syntax of FL/R as follows:

E ::= ... | `(tuple` E_1 ... E_n`)` | `(tuple-ref` E_{tuple} N_{index} N_{size}`)`

Here is an informal description of Sam's new expressions:

- `(tuple` E_1 ... E_n`)` packages up the values E_1 ... E_n into an n-tuple. Unlike lists and vectors, tuples are heterogeneous data structures: the values of the E_i expressions can all be of different types.
- `(tuple-ref` E_{tuple} N_{index} N_{size}`)` evaluates E_{tuple}, which should be an N_{size}-tuple t, and returns the N_{index}-th component of t. For example:

  ```
  (tuple-ref (tuple 17 #t (symbol captain) "abstraction") 2 4)
  ```

  yields the second element, `#t`, from a 4-tuple. In Sam's syntax, note that the index and size *must* be integer literals — they are *not* general expressions to be evaluated.

a. Extend the FL/R type grammar to handle tuples.

b. Give the typing rules for `tuple` and `tuple-ref`.

c. Specify the type reconstruction algorithm clauses for `tuple` and `tuple-ref`.

d. Louis Reasoner thinks that the form of `tuple-ref` is unwieldy. "I don't see why N_{size} is at all necessary," he complains. "Why don't you make `tuple-ref` have the following form instead?"

`(tuple-ref` E_{tuple} N_{index}`)`

Briefly explain why Louis' suggestion would complicate type reconstruction for FL/R.

e. After Sam successfully explains to Louis his rationale for N_{size}, Louis has another suggestion: "I don't see why the form of N_{index} has to be so restricted. Why not change the form of `tuple-ref` to be the following?"

$$\texttt{(tuple-ref } E_{tuple} \ E_{index} \ N_{size}\texttt{)},$$

where the component index is the computed value of E_{index} rather than a literal integer value.

Why is this a bad idea?

◁

▷ **Exercise 14.3** Ben Bitdiddle enhanced FL/R with several parallel programming constructs. His most important extension is a new construct called `go` that executes multiple expressions in parallel. Ben extended the FL/R grammar as follows:

E ::= ... | `(go (`I_1 ... I_n`)` E_1 ... E_m`)` | `(talk!` I E`)` | `(listen` I`)`

Here is the informal semantics of the newly added constructs: `go` terminates when all of E_1 ... E_m terminate; it returns the value of E_1. `go` includes the ability to use communication variables I_1 ... I_n in a parallel computation. A communication variable can be assigned a value by `talk!`. An expression in `go` can wait for a communication variable to be given a value with `listen`. `listen` returns the value of the variable once it is set with `talk!`. For a program to be well-typed, all E_1 ... E_m in `go` must be well-typed.

Ben extended the type grammar of FL/R as follows:

T ::= ... | `(commof` T`)`

Communication variables will have the unique type `(commof` T`)` where T is the type of value they hold. This will ensure that only communication variables can be used with `talk!` and `listen`, and that communication variables can not be used in any other expression. Ben has already written the typing rules for `talk!` and `listen`:

$$\frac{A \vdash E : \texttt{T} \qquad A \vdash I : \texttt{(commof T)}}{A \vdash \texttt{(talk!} \ I \ E\texttt{)} : \texttt{unit}} \quad [\textit{talk!}]$$

$$\frac{A \vdash I : \texttt{(commof T)}}{A \vdash \texttt{(listen} \ I\texttt{)} : \texttt{T}} \quad [\textit{listen}]$$

a. Give the typing rule for `go`.

b. Give the FL/R reconstruction algorithm clauses for `talk!`, `listen`, and `go`.

◁

Reading

Type reconstruction in programming languages is due to Milner [Mil78], who reinvented work previously done in logic by Curry and by Hindley. Examples of programming languages with type reconstruction are ML [MTH90, MT91], Haskell [HJW+92], and FX [GJSO92].

Chapter 15

Abstract Types

The human heart has hidden treasures,
In secret kept, in silence sealed.

— *Evening Solace, Charlotte Bronte*

15.1 Data Abstraction

A cornerstone of modern programming methodology is the principle of **data abstraction**, which states that programmers should be able to use data structures without understanding the details of how they are implemented. Data abstraction is based on establishing a **contract**, also known as an **application programming interface (API)**, or just **interface**, that specifies the abstract behavior of all operations that manipulate a data structure without describing the representation of the data structure or the algorithms used in the operations.

The contract serves as an **abstraction barrier** that separates the concerns of the two parties that participate in a data abstraction. On one side of the barrier is the **implementer**, who is responsible for implementing the operations so that they satisfy the contract. On the other side of the barrier is the **client**, who is blissfully unaware of the hidden implementation details and uses the operations based purely on their advertised specifications in the contract. This arrangement gives the implementer the flexibility to change the implementation at any time as long as the contract is still satisfied. Such changes should not require the client to modify any code.[1] This separation of concerns is especially

[1] However, the client may need to recompile existing code in order to use a modified implementation.

useful when large programs are being developed by multiple programmers, many of whom may never communicate except via contracts. But it is even helpful in programs written by a single person who plays the roles of implementer and client at different times in the programming process.

15.1.1 A Point Abstraction

As an extremely simple example of data abstraction, consider an abstraction for points on a two-dimensional grid. The point abstraction is defined by the following contract, which specifies an operation for creating a point from its two coordinates and operations for extracting each coordinate:

- `(make-pt` x y`)`: Create a point whose x coordinate is the integer x and whose y coordinate is the integer y.
- `(pt-x` p`)`: Return the x coordinate of the given point p.
- `(pt-y` p`)`: Return the y coordinate of the given point p.

An implementation of the point abstraction should satisfy the following axioms:

1. For any integers n_1 and n_2, `(pt-x (make-pt` n_1 n_2`))` evaluates to n_1.
2. For any integers n_1 and n_2, `(pt-y (make-pt` n_1 n_2`))` evaluates to n_2.

Even for this simple abstraction, there are a surprising number of possible implementations. For concreteness, below we give two point implementations in the dynamically typed FL language. Our convention will be to package up the operations of a data abstraction into a record, but that is not essential.

```
(define pair-point-impl
  (record
    (make-pt (lambda (x y) (pair x y)))
    (pt-x (lambda (p) (left p)))
    (pt-y (lambda (p) (right p)))))

(define proc-point-impl
  (record
    (make-pt (lambda (x y) (lambda (b) (if b x y))))
    (pt-x (lambda (p) (p #t)))
    (pt-y (lambda (p) (p #f)))))
```

In `pair-point-impl`, the two coordinates are stored in a pair. Alternatively, we could have stored them in the opposite order or glued them together in a different kind of product (e.g., array, record, or list). In `proc-point-impl`, a

point is represented as a first-class procedure that "remembers" the coordinates in its environment and uses a boolean argument to determine which coordinate to return when called. Alternatively, some other key (such as a symbol or string message) could be used to select the coordinate.

As a sample client of the point abstraction, consider the following procedure, which, for a given point implementation, defines a coordinate-swapping `transpose` procedure and a `point->pair` procedure that converts a point to a concrete pair (regardless of its underlying representation) and uses these on the point (1,2).

```
(define test-point-impl
  (lambda (point-impl)
    (with-fields (make-pt pt-x pt-y) point-impl
      (let ((transpose (lambda (p) (make-pt (pt-y p) (pt-x p))))
            (point->pair (lambda (p) (pair (pt-x p) (pt-y p)))))
        (point->pair (transpose (make-pt 1 2)))))))
```

The result of invoking `test-point-impl` on a valid point implementation should be the pair value $\langle 2, 1 \rangle$.

In this example, there is little reason to prefer one of the implementations over the other. The pair implementation might be viewed as being more straightforward, requiring less memory space, or being more efficient because it requires fewer procedure calls. However, judgments about efficiency are often tricky and require a deep understanding of low-level implementation details. In more realistic examples, such as abstractions for data structures like stacks, queues, priority queues, sets, tables, databases, etc., one implementation might be preferred over another because of asymptotically better running times or memory usage for certain operations.

15.1.2 Procedural Abstraction is not Enough

Any language with procedural abstraction can be used to implement data abstraction in the way illustrated in the point example. However, in order for the full benefits of data abstraction to be realized, this approach requires that the client never commit **abstraction violations**. An abstraction violation is the inappropriate use of abstract values or their operations.

In our implementation of points that uses pairs, the client can inspect the representation of an abstract value and use this knowledge to manipulate abstract values concretely. For instance, if points are represented as pairs, then the the client might write `(left p)` rather than `(pt-x p)` to extract the x coordinate of a point `p`, or might create a point "forgery" using `(pair 1 2)` in place of `(make-pt 1 2)`. Although these concrete manipulations will not cause errors,

such abuses of the exposed representation are dangerous because they are not guaranteed to work if the implementation is changed. For example, `(left p)` would lead to a runtime type error if the implementation were changed to use a procedural representation for points, and would give the incorrect value if the implementation was changed to put the y coordinate before the x coordinate in a pair.

Furthermore, many representations involve **representation invariants** that are maintained by the abstract operations but which concrete manipulations may violate. A representation invariant is a set of conceptual or actual predicates that a representation must satisfy to be legal. For instance, a string collection implementation might store the strings in a sorted array. Thus a sorted predicate would be true for this representation. If the client creates a forgery with an unsorted array, all bets are off concerning the behavior of the abstract operations on this forgery.

Without an enforcement of the relationship between abstract values and their operations, it is even possible to interchange values of different abstractions that happen to have the same concrete representation. For instance, if an implementation of a rational number abstraction represents a rational number as a pair of two integers, then a rational number could be dissected with `pt-x` and `pt-y`, assuming that points are also represented as pairs of integers.

Although our examples have been for a dynamically typed language, the same problems occur in a statically typed language with structural type equality. Clearly, attempting to achieve data abstraction using procedural abstraction alone is fraught with peril. There must additionally be some sort of mechanism to guarantee that abstract data is **secure**. We will call a language secure when barriers associated with a data abstraction cannot be violated. Such a security mechanism must effectively hide the representation of abstract data by making it illegal to create or operate on abstract values with anything other than the appropriate abstract operations.

In the remainder of this chapter, we first consider how secure data abstractions can be achieved dynamically using a lock and key mechanism. Then we study various ways to achieve such security statically using types.

▷ **Exercise 15.1** In languages with first-class procedures, one approach to hiding the representations of data structures is to encapsulate them in message-passing objects. For example, the following two point-making procedures encapsulate the pair representation and procedural representation, respectively:

```
(define make-pair-point
  (lambda (x y)
    (let ((point (pair x y)))
      ;; Return a message dispatcher
      (lambda (msg)
        (cond
          ((sym=? msg 'pt-x) (left point))
          ((sym=? msg 'pt-y) (right point))
          (else (error unrecognized-message))
          )))))

(define make-proc-point
  (lambda (x y)
    (let ((point (lambda (b) (if b x y))))
      ;; Return a message dispatcher
      (lambda (msg)
        (cond
          ((sym=? msg 'pt-x) (point true))
          ((sym=? msg 'pt-y) (point false))
          (else (error unrecognized-message))
          )))))
```

How secure is this approach to hiding data abstraction representations? What kinds of abstraction violations are prevented by this technique? What kinds of abstraction violations can still occur? ◁

15.2 Dynamic Locks and Keys

One approach for securely encapsulating a data abstraction representation is to make it inaccessible by "locking" abstract values with a "key" in such a way that only the very same key can unlock a locked value to access the representation. We explore a dynamic lock and key mechanism by extending FL! with the following primitives:

- `(new-key)` generates a unique unforgeable key value.
- `(lock` *key value*`)` creates a new kind of "locked value" that pairs *key* with *value* in such a way that *key* cannot be extracted and *value* can only be extracted by supplying *key*.
- `(unlock` *key locked*`)` returns the value stored in *locked* if *key* matches the key used to create *locked*. Otherwise, signals an error.

We extend FL! rather than FL because the presence of cells and a single-threaded store simplify specifying the semantics of these constructs. Indeed,

`new-key`, `lock`, and `unlock` can all be implemented as user-defined procedures in FL! (Figure 15.1). The `new-key` procedure creates a new cell whose location *is* a unique and unforgeable key; the value in the cell is arbitrary and can be ignored. The `lock` procedure represents a locked value as a procedure that "remembers" the given key and value and only returns the value if it is invoked on the original key (as done in `unlock`). The procedural representation of locked values prevents direct access to the key, and the value can only be extracted by supplying the key, as desired.

```
(define new-key (lambda () (cell 0)))

(define lock
  (lambda (key val)
    (lambda (key1)
      (if (cell=? key key1)
          val
          (error wrong-key)))))

(define unlock
  (lambda (key locked)
    (locked key)))
```

Figure 15.1: Implementation of a dynamic lock and key mechanism in FL!.

Figure 15.2 shows how the lock and key mechanism can be used to securely encapsulate two pair representations of points that differ only in the order of the coordinates. The procedures `up` and `down` use `lock` and `unlock` to mediate between the concrete pair values and the abstract point values. Because all operators for a single implementation use the same key, the operators for `pt-impl1` work together, as do those for `pt-impl2`. For example:

```
((select pt-x pt-impl1) ((select make-pt pt-impl1) 1 2)) ⟶FL! 1
((select pt-y pt-impl2) ((select make-pt pt-impl2) 1 2)) ⟶FL! 2
```

However, because different implementations use different keys, point values created by one of the implementations cannot be dissected by operations of the other. Furthermore, because the operators create and use locked values, neither point implementation can be used with concrete pair operations. For example, all of the following four expressions generate dynamic errors when evaluated:

```
((select pt-x pt-impl1) ((select make-pt pt-impl2) 1 2))
((select pt-y pt-impl2) ((select make-pt pt-impl1) 1 2))
(left ((select make-pt pt-impl1) 1 2))
((select pt-y pt-impl2) (pair 1 2))
```

```
(define pt-impl1
  (let ((key (new-key)))
    (let ((up (lambda (x) (lock key x)))
          (down (lambda (x) (unlock key x))))
      (record
        (make-pt (lambda (x y) (up (pair x y))))
        (pt-x (lambda (p) (left (down p))))
        (pt-y (lambda (p) (right (down p))))))))

(define pt-impl2
  (let ((key (new-key)))
    (let ((up (lambda (x) (lock key x)))
          (down (lambda (x) (unlock key x))))
      (record
        (make-pt (lambda (x y) (up (pair y x))))
        (pt-x (lambda (p) (right (down p))))
        (pt-y (lambda (p) (left (down p))))))))
```

Figure 15.2: Using the lock and key mechanism to hide point representations.

Some syntactic sugar can facilitate the definition of implementation records. We introduce a `cluster` macro that abstracts over the pattern used in the point implementations:

$\mathcal{D}_{\text{exp}}[\![$`(cluster (`$I$ E`)*)`$]\!] =$
`(let ((`I_{key} `(new-key))) ;` I_{key} `fresh`
`(let ((up (lambda (x) (lock` I_{key} `x)))`
`(down (lambda (x) (unlock` I_{key} `x))))`
`(recordrec (`I E`)*)))`

The `up` and `down` procedures implicitly introduced by the desugaring may be used in any of the cluster bindings. Using `recordrec` in place of `record` allows for mutually recursive operations. Here is the definition of `pt-impl1` re-expressed using the `cluster` notation:

```
(define pt-impl1
  (cluster
    (make-pt (lambda (x y) (up (pair x y))))
    (pt-x (lambda (p) (left (down p))))
    (pt-y (lambda (p) (right (down p))))))
```

Note that `cluster` creates a new data abstraction every time it is evaluated. For instance, consider:

```
(define make-wrapper
  (lambda ()
    (cluster
      (wrap (lambda (x) (up x)))
      (unwrap (lambda (x) (down x))))))

(define wrapper1 (make-wrapper))
(define wrapper2 (make-wrapper))
```

Evaluating `((select unwrap wrapper2) ((select wrap wrapper1) 17))` signals a dynamic error because the `wrap` procedure from `wrapper1` and the `unwrap` procedure from `wrapper2` use different keys.

▷ **Exercise 15.2** Consider an integer set abstraction that supports the following operations:

- `(empty)` creates an empty set of integers.
- `(insert` *int intset*`)` returns the set that results from inserting *int* into the integer set *intset*.
- `(member?` *int intset*`)` returns *true* if *int* is a member of the integer set *intset* and *false* otherwise.

a. Define a cluster `list-intset-impl` that represents an integer set as a list of integers without duplicates sorted from low to high.

b. Define a cluster `pred-intset-impl` that represents an integer set as a predicate – a procedure that takes an integer and returns *true* if that integer is in the set represented by the predicate and false otherwise.

c. Extend both `list-intset-impl` and `pred-intset-impl` to handle union, intersection, and difference operations on two integer sets.

d. Some representations have advantages over other for implementing particular operations. Show that `size` (which returns the number of elements in an integer set) is easy to implement for `list-intset-impl` but but impossible to implement for `pred-intset-impl` (without changing the representation). Similarly, show that `complement` (which returns the set of all integers not in the given set) is easy to implement for `pred-intset-impl` but impossible to implement for `list-intset-impl`. ◁

▷ **Exercise 15.3**

a. Extend the SOS for FLK! to directly handle the primitives `new-key`, `lock`, and `unlock`. Assume that the syntactic domains MixedExp and ValueExp are extended with expressions of the form `(*key*` *L*`)` to represent keys and `(*locked*` *L V*`)` to represent locked values.

b. It is helpful to have the following additional primitives as well:

- `(key?` *thing*`)` determines if *key* is a key value.
- `(key=?` *key1 key2*`)` returns true if *key1* is the same key value as *key2* and returns false otherwise. Signals an error if either *key1* or *key2* is not a key value.
- `(locked?` *thing*`)` determines if *thing* is a locked value.

Extend your SOS to handle these primitives.

c. Can you extend the implementation in Figure 15.1 to handle the additional primitives? Explain. ◁

▷ **Exercise 15.4** It is not always desirable to export every binding of a cluster in the resulting record. For example, in the following implementation of a rational number cluster, the `gcd` function (which calculates the greatest common divisor of two numbers) is intended to be an unexported local recursive function used by `make-rat`.

```
(define rat-impl
  (cluster
    (make-rat (lambda (x y)
                (let ((g (gcd x y)))
                  (up (pair (div x g) (div y g))))))
    (numer (lambda (r) (left (down r))))
    (denom (lambda (r) (right (down r))))
    (gcd (lambda (a b)
           (if (= b 0)
               a
               (gcd b (rem a b)))))
    ))
```

In this case, we could make the definition of `gcd` local to `make-rat`, but this strategy does not work if the local value is used in multiple bindings. Alternatively, we can extend the `cluster` syntax to be:

`(cluster (`I_{exp}`*) (`I E`)*)`

where `(`I_{exp}`*)` is an explicit list of **exports** – those bindings we wish to be included in the resulting record. For instance, if we use `(make-rat numer denom)` as the export list in `rat-impl`, then `gcd` would not appear in the resulting record. Modify the desugaring of `cluster` to support explicit export lists. ◁

▷ **Exercise 15.5** A dynamic lock and key mechanism can be added to a statically typed language like FL/X.

a. Extend the type syntax and typing rules of FL/X to handle `new-key`, `lock`, and `unlock`.

b. We can add a `cluster` form to FL/X using the syntax

$$\texttt{(cluster } T_{rep} \texttt{ (}I_1\ T_1\ E_1\texttt{) ... (}I_n\ T_n\ E_n\texttt{))},$$

where T_{rep} is the concrete representation type of the data abstraction and T_n is the type of E_n. Give a typing rule for this explicitly typed `cluster` form.

c. Why is it necessary to include T_{rep} and the T_i in the explicitly typed `cluster` form? Would these be necessary in a `cluster` form for FL/R? ◁

booksectionExistential Types

The dynamic lock and key mechanism enforces data abstraction by signaling a run-time error whenever an abstraction violation is encountered. The main drawback of this approach is its dynamic nature. It would be desirable to have a static mechanism that reports abstraction violations when the program is type checked. As usual, the constraints of computability prevent a static system from detecting exactly those violations that would be caught by a dynamic lock and key mechanism. Nevertheless, by relinquishing some expressive power, it is possible to design type systems that prevent abstraction violations via a static lock and key mechanism known as an **abstract type**. In the next three sections, we shall study three designs for abstract types.

Our first abstract type system is based on extending the explicitly typed language FL/XSP with **existential types**. To motivate existential types, consider the types of the `pair-point-impl` and `proc-point-impl` implementations introduced in Section 15.1:

```
(define-type pair-point-impl-type
  (recordof
    (make-pt (-> (int int) (pairof int int)))
    (pt-x (-> ((pairof int int)) int))
    (pt-y (-> ((pairof int int)) int))))

(define-type proc-point-impl-type
  (recordof
    (make-pt (-> (int int) (-> (bool) int)))
    (pt-x (-> ((-> (bool) int)) int))
    (pt-y (-> ((-> (bool) int)) int))))
```

These two types are the same except for the concrete type used to represent an abstract point value: `(pairof int int)` in the first case and `(-> (bool) int)` in the second. We would like to be able to say that both implementations have the same abstract type. We call values that implement an abstract type a **package**. To represent the type of a package, we use a new type construct, packof$_{exist}$, to introduce an abstract type name, `point`, that stands for the concrete type used in a particular implementation:

```
(define-type pt-eface
  (packof_exist point
    (recordof
      (make-pt (-> (int int) point))
      (pt-x (-> (point) int))
      (pt-y (-> (point) int)))))
```

We informally read the above (packof$_{exist}$ ...) type as "there exists a concrete point representation (call it `point`) such that there are `make-pt`, `pt-x`, and `pt-y` procedures with the specified types that manipulate this representation." Such a type is called an **existential type** because it posits the existence of an abstract type and indicates how it is used without saying anything about its concrete representation.[2] In the following discussion, we will often refer to this particular existential type, so we have given it the name `pt-eface`, where `eface` is short for **e**xistential inter**face**.

A summary of existential types is presented in Figure 15.3. The form of an existential type is (packof$_{exist}$ I T). The existential variable I is a binding occurrence of a type variable whose scope is T. The particular name of this variable is irrelevant; as is indicated by the [*exists*=] type equality rule, it can be consistently renamed without changing the essence of the type. So the type

```
(packof_exist q
  (recordof
    (make-pt (-> (int int) q))
    (pt-x (-> (q) int))
    (pt-y (-> (q) int))))
```

is equivalent to the existential type using `point` above.

Values of existential type, which we shall call **existential packages**, are created by the form (pack$_{exist}$ I_{abs} T_{rep} E_{impl}). The type identifier I_{abs} is a type name that is used to hide the concrete representation type T_{rep} within the type of the implementation expression E_{impl}. For example, Figure 15.4 shows two existential packages that implement the type contract specified by `pt-eface`. In the first package, the abstract name `point` stands for the type of pair of integers, while in the second package, it stands for the type of a procedure that maps a boolean to an integer. As in the dynamic `cluster` form studied in Section 15.2, the pack$_{exist}$ form implicitly introduces `up` and `down` procedures that convert between the concrete and abstract values.

[2] In the literature, such types are often written with ∃ or `exists` just as ∀ and `forall` are used for universal polymorphism. For example, a more standard syntax for the `pt-eface` type is: ∃ `point` . {`make-pt: int*int` → `point, pt-x: point` → `int, pt-y: point` → `int`}.

Syntax

$E ::= \ldots \mid$ `(pack`$_{exist}$ I_{abs} T_{rep} E_{impl}`)` [Existential Introduction]
$\quad \mid$ `(unpack`$_{exist}$ E_{pkg} I_{ty} I_{impl} E_{body}`)` [Existential Elimination]

$T ::= \ldots \mid$ `(packof`$_{exist}$ I_{abs} T_{impl}`)` [Existential Type]

Type Rules

$$\frac{A[\texttt{up:(-> (}T_{rep}\texttt{) }I_{abs}\texttt{), down:(-> (}I_{abs}\texttt{) }T_{rep}\texttt{)}] \vdash E_{impl} : T_{impl}}{A \vdash \texttt{(pack}_{exist}\ I_{abs}\ T_{rep}\ E_{impl}\texttt{)} : \texttt{(packof}_{exist}\ I_{abs}\ T_{impl}\texttt{)}} \quad [epack]$$

where $I_{abs} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E_{impl}]\!]\}$ *[import restriction]*

$$\frac{\begin{array}{c}A \vdash E_{pkg} : \texttt{(packof}_{exist}\ I_{abs}\ T_{impl}\texttt{)} \\ A[I_{impl} : [I_{ty}/I_{abs}]T_{impl}] \vdash E_{body} : T_{body}\end{array}}{A \vdash \texttt{(unpack}_{exist}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{body}\texttt{)} : T_{body}} \quad [eunpack]$$

where $I_{ty} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E_{body}]\!]\}$ *[import restriction]*
$I_{ty} \notin (FTV\ T_{body})$ *[export restriction]*

Type Equality

$$\texttt{(packof}_{exist}\ I\ T\texttt{)} = \texttt{(packof}_{exist}\ I'\ [I'/I]T\texttt{)} \quad [exists=]$$

Type Erasure

$\lceil \texttt{(pack}_{exist}\ I_{abs}\ T_{rep}\ E_{impl}\texttt{)} \rceil$
$= \texttt{(let ((up (lambda (x) x))}$
$\texttt{(down (lambda (x) x)))}$
$\lceil E_{impl} \rceil\texttt{)}$

$\lceil \texttt{(unpack}_{exist}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{body}\texttt{)} \rceil = \texttt{(let ((}I_{impl}\ \lceil E_{pkg} \rceil\texttt{))}\ \lceil E_{body} \rceil\texttt{)}$

Figure 15.3: The essence of existential types in FL/XSP.

```
(define pair-point-epkg pt-eface
  (pack_exist point (pairof int int)
    (record
      (make-pt (lambda ((x int) (y int)) (up (pair x y))))
      (pt-x (lambda ((p point)) (left (down p))))
      (pt-y (lambda ((p point) (right (down p))))))))

(define proc-point-epkg pt-eface
  (pack_exist point (-> (bool) int)
    (record
      (make-pt (lambda ((x int) (y int))
                 (up (lambda ((b bool)) (if b x y)))))
      (pt-x (lambda ((p point)) ((down p) true)))
      (pt-y (lambda ((p point) ((down p) false)))))))
```

Figure 15.4: Two existential packages that implement `pt-eface`.

The [*epack*] type rule in Figure 15.3 specifies how different implementations can have exactly the same existential type. The implementation expression E_{impl} is checked in a type environment where `up` converts from the concrete representation type T_{rep} to the abstract type name I_{abs} and `down` converts from I_{abs} to T_{rep}. These conversions allow the implementer to hide the concrete representation type with an **opaque** type name, so called because the concrete type cannot be "seen" through the name, even though the name is an abbreviation for the concrete type.[3] If type checking of a $\texttt{pack}_{exist}$ expression succeeds, we have a proof that there is at least one representation for I_{abs} (namely T_{rep}) and one implementation using this representation (namely E_{impl}) that satisfies the implementation type T_{impl}. This knowledge is recorded with the type $(\texttt{packof}_{exist}\ I_{abs}\ T_{impl})$, in which any implementation details related to T_{rep} and E_{impl} have been purposely omitted.

The $\texttt{pack}_{exist}$ expression can be viewed as a way to package up an implementation in such a way that representation details are hidden. As indicated by the type erasure for $\texttt{pack}_{exist}$ in Figure 15.3, the dynamic meaning of a $\texttt{pack}_{exist}$ expression is just the implementation expression in a context where `up` and `down` are identity operations. The remaining parts of the expression (I_{abs} and T_{rep}) are just type annotations whose purpose is to specify the existential type.

The existential elimination form, $(\texttt{unpack}_{exist}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{body})$, is the means of using the underlying implementation hidden by an existential package.

[3] `up` and `down` are just one way to distinguish concrete and abstract types in an existential type. Some alternative approaches are explored in Exercise 15.9.

The type erasure of this expression — `(let ((`I_{impl} $\lceil E_{pkg} \rceil$`))` $\lceil E_{body} \rceil$`)` — indicates that the dynamic meaning of this expression is simply to give the name I_{impl} to the implementation in the scope of the body. The type name I_{ty} serves as a local name for the abstract type of the existential package that can be used within E_{body}. The abstract name within the existential type itself is unsuitable for this purpose because (1) it is not lexically apparent to E_{body} and (2) it is a bound name that is subject to renaming.

As an example of `unpack`$_{exist}$, consider the following procedure, which is a typed version of the `test-point-impl` procedure presented in Section 15.1.

```
(define test-point-epkg (-> (pt-eface) (pairof int int))
  (lambda ((point-epkg pt-eface))
    (unpack_exist point-epkg pt point-ops
      (with point-ops
        (let ((transpose (lambda ((p pt))
                           (make-pt (pt-y p) (pt-x p))))
              (point->pair (lambda ((p pt))
                             (pair (pt-x p) (pt-y p)))))
          (point->pair (transpose (make-pt 1 2))))))))
```

The `point-epkg` argument to `test-point-epkg` is any existential package with type `pt-eface`. The `unpack`$_{exist}$ form gives the local name `pt` to the abstract point type and the local name `point-ops` to the implementation record containing the `make-pt`, `pt-x`, and `pt-y` procedures. In the context of local bindings for these procedures (made available by `with`), the local `transpose` and `point->pair` procedures are created. Each of these takes a point as an argument and so must refer to the local abstract type name `pt` for the abstract point type. Finally, `test-point-epkg` returns a pair of the swapped coordinates for the point (1,2).

In the [*eunpack*] type rule, it is assumed that the package expression E_{pkg} has type `(packof`$_{exist}$ I_{abs} T_{impl}`)`. The body expression E_{body} is type checked under the assumption that I_{impl} has as its type a version of T_{impl} in which the bound name I_{abs} has been replaced by the local abstract type name I_{ty}. For instance, in the above `unpack`$_{exist}$ example, where `pt` is the local abstract type name, the `make-pt` procedure has type `(-> (int int) pt)`. The fact that the result type is `pt` rather than `point` is essential for matching up the return type of `transpose` and the declared argument type of `point->pair`.

In the [*epack*] rule, there is an **import restriction** on the abstract type name I_{abs} that prevents it from accidentally capturing a type identifier mentioned in the type of a free variable in E_{impl}. Here is an expression that would unsoundly be declared well-typed without this restriction:

```
(plambda (t)
  (lambda ((z t))
    (pack_exist t int (down z))))
```

The application `(down z)` applies the `down` procedure to a value `z` of arbitrary type `t`. But since `down` has type `(-> (t) int)`, where `t` abstracts over the concrete type `int`, this application would unsoundly be declared well-typed without the import restriction. A similar import restriction is also needed in the [*eunpack*] rule. The import restriction is not a serious issue for programmers because it can be satisfied by automatically α-renaming a program to give distinct names to logically distinct type identifiers.

In contrast, the **export restriction** $I_{ty} \notin (FTV\ T_{body})$ in the [*eunpack*] rule can be a serious impediment. This restriction says that the local abstract type name I_{ty} is not allowed to escape the scope of the `unpack`$_{exist}$ expression by appearing in the type T_{body} of the body expression E_{body}. A consequence is that no value of the abstract type can escape from `unpack`$_{exist}$ in any way.

Without the export restriction, the [*eunpack*] rule would be unsound. Consider the following example of what would go wrong if the restriction were removed:

```
(let ((p (unpack_exist proc-point-epkg t point-ops1
           (with point-ops1 (make-pt 1 2))))
      (f (unpack_exist pair-point-epkg t point-ops2
           (with point-ops2 pt-x))))
  (f p)).
```

The first `unpack`$_{exist}$ makes a procedural point whose type within the `unpack`$_{exist}$ is the local abstract type `t`. This point escapes from the `unpack`$_{exist}$ and is `let`-bound to the name `p`. The type of the point at this time is still `t`, which is an unbound type variable in this context. The second `unpack`$_{exist}$ unpackages a pair point implementation and returns its `pt-x` operation, which is renamed `f`. Since `t` is also used as the local abstract type in the second `unpack`$_{exist}$, the type of `f` is `(-> (t) int)`, where `t` again is actually an unbound type variable. Since `f` has type `(-> (t) int)` and `p` has type `t`, the application `(f p)` would be well-typed. But dynamically an attempt is being made to take the left component of a procedural point, which should be a type error! This example makes clear that while it is powerful to be able to locally name the abstract type within `unpack`$_{exist}$, the local type name has no meaning outside the scope of the `unpack`$_{exist}$ and so cannot be allowed to escape.

The export restriction fundamentally limits the usefulness of existential types in practice. For instance, in the `test-point-epkg` procedure studied above, it would be more natural to return the transposed point directly, but then

the type of the `unpack`$_{exist}$ expression would be the abstract type `pt`, which is forbidden by the export restriction. Instead we must first convert the abstract point to a concrete pair in order to satisfy the export restriction. The restriction also prevents us from writing a `make-transpose` procedure that takes a point package and returns a `transpose` procedure appropriate for that package. The type of `make-transpose` would presumably be something like `(-> (pt-eface) (-> (`I_{abs}`) `I_{abs}`))`, where I_{abs} is the name of the abstract type used by the given point package. But there is no way to refer to that type except within `unpack`$_{exist}$ expressions inside the body of `make-transpose`, and that type cannot escape any such expressions to end up in the result type of `make-transpose`.

In practice, there are a few ways to finesse the export type restriction. One approach is to organize programs in such a way that large regions of the program are within the body of `unpack`$_{exist}$ expressions that open up commonly used data abstractions. Within these large regions, it is possible to freely manipulate values of the abstract type. The problem with this approach is that it can make it more difficult to take advantage of one of the key benefits of existential types: the ability to abstract code over different implementations of the same abstract type and choose implementations at run-time based on dynamic conditions.

In cases where we really want to pass values that mention the abstract type outside the scope of an `unpack`$_{exist}$, we can program around the restriction by packaging up such values together with their abstract type into a new existential type. For example, Figure 15.5 shows how to define an `extend-point-epkg` procedure that can take any package with type `pt-eface` and return a new package that has new operations and values in addition to the old ones. While this technique addresses the problem, it can be cumbersome, especially since all values mentioning the same abstract type must always be put together into the same package (or else later they could not be used with each other). Furthermore, the components of the original package need to be repackaged to get the right abstract type (and satisfy the import restriction).[4]

One paradigm in which the packaging overhead is not too onerous is a simple form of object-oriented programming. Figure 15.6 shows how the pair and procedural point representations can be encapsulated as existential packages whose implementations combine the state and methods of an object. As shown in the figure, in this paradigm, it is possible to express a generic top-level `transpose` method that operates on any value with type `point-object`. For example, the following expression is well-typed:

[4]This is an artifact of using `up`/`down` to convert between abstract and concrete types. Such repackaging is not necessary in some other approaches; see Exercise 15.9.

```
(define-type new-pt-eface
  (packof_exist point
    (recordof
      (make-pt (-> (int int) point))
      (pt-x (-> (point) int))
      (pt-y (-> (point) int))
      (transpose (-> (point) point))
      (point->pair (-> (point) (pairof int int)))
      (origin point)
      )))

(define extend-point-epkg (-> (pt-eface) new-pt-eface)
  (lambda ((point-epkg pt-eface))
    (unpack_exist point-epkg pt point-ops
      (with point-ops
        (pack_exist newpt pt
          (record
            (make-pt (lambda ((x int) (y int)) (up (make-pt x y))))
            (pt-x (lambda ((p newpt)) (pt-x (down p))))
            (pt-y (lambda ((p newpt)) (pt-y (down p))))
            (transpose (lambda ((p newpt))
                         (up (make-pt (pt-y (down p))
                                      (pt-x (down p))))))
            (point->pair (lambda ((p newpt))
                           (pair (pt-x (down p)) (pt-y (down p)))))
            (origin (up (make-pt 0 0)))))))))
```

Figure 15.5: The `extend-point-epkg` procedure shows how values mentioning an abstract type can be passed outside `unpack`$_{exist}$ as long as they are first packaged together with their abstract type.

```
(let ((points (list point-object
                    (make-pair-point 1 2)
                    (make-proc-point 3 4))))
  ((pcall append point-object)
   points
   ((pcall map point-object point-object) transpose points)))
```

For simplicity, the existential type system considered here does not permit parameterized abstract types, but it can be extended to do so. For instance, here is an interface type for immutable stacks that is parameterized over the stack component type `t`:

```
(define-type stack-eface
  (poly (t)
    (packof_exist (stackof t)
      (recordof
        (empty (-> () (stackof t)))
        (empty? (-> ((stackof t)) bool))
        (push (-> (t (stackof t)) (stackof t)))
        (pop (-> ((stackof t)) (stackof t)))
        (top (-> ((stackof t)) t))))))
```

Parameterized existential types are explored in Exercise 15.8.

▷ **Exercise 15.6** This exercise revisits the integer set abstraction introduced in Exercise 15.2.

a. Define an interface type `intset-eface` for integer sets supporting the operations `empty`, `insert`, and `member?`.

b. Define an existential package `list-intset-epkg` implementing `intset-eface` that represents integer sets as integer lists.

c. Define an existential package `pred-intset-epkg` implementing `intset-eface` that represents integer sets as integer predicates.

d. Define a testing procedure `test-intset` that takes any implementation of type `intset-eface`, creates a set `s` containing the integers `1` and `3`, and returns a three-element boolean list whose ith element (1-indexed) indicates whether `s` contains the integer i. ◁

▷ **Exercise 15.7**

a. Illustrate the necessity of the import restriction for the [*eunpack*] rule by giving an expression that would unsoundly be well-typed without the restriction.

b. Alf Aaron Ames claims that the import restriction in the [*epack*] rule and the import and export restrictions in the [*eunpack*] rule are all unnecessary if before

```
(define-type point-object
  (packof_exist point
    (recordof
      (state point)
      (methods (recordof
                 (make-pt (-> (int int) point))
                 (pt-x (-> (point) int))
                 (pt-y (-> (point) int)))))))

(define make-pair-point (-> (int int) point-object)
  (lambda ((x int) (y int))
    (pack_exist point (pairof int int)
      (let ((make-pt (lambda ((x int) (y int)) (up (pair x y)))))
        (record
          (state (make-pt x y))
          (methods (record
                     (make-pt make-pt)
                     (pt-x (lambda ((p point)) (left (down p))))
                     (pt-y (lambda ((p point)) (right (down p))))))))))))

(define make-proc-point (-> (int int) point-object)
  (lambda ((x int) (y int))
    (pack_exist point (-> (bool) int)
      (let ((make-pt (lambda ((x int) (y int))
                       (up (lambda ((b bool)) (if b x y))))))
        (recordof
          (state (make-pt x y))
          (methods (record
                     (make-pt make-pt)
                     (pt-x (lambda ((p point)) ((down p) true)))
                     (pt-y (lambda ((p point)) ((down p) false)))))))))))

(define transpose (-> (point-object) point-object)
  (lambda ((pobj point-object))
    (unpack_exist pobj pt impl
      (with impl
        (with methods
          (pack_exist newpt pt
            (record
              (state (up (make-pt (pt-y state) (pt-x state))))
              (methods
                (record
                  (make-pt (lambda ((x int) (y int)) (up (make-pt x y))))
                  (pt-x (lambda ((p newpt)) (pt-x (down p))))
                  (pt-y (lambda ((p newpt)) (pt-y (down p))))))))))))))
```

Figure 15.6: Encoding two pair object representations using existential types.

type checking the program is α-renamed to make all logically distinct type identifiers unique. Is Alf correct? Use suitably modified versions of the unsoundness examples in this section to support your answer. ◁

▷ **Exercise 15.8**

a. Extend the syntax and typing rules of FL/XSP to handle parameterized existential types like `(stackof t)`, which appears in the `stack-eface` example above.

b. Define an implementation `stack-list-epkg` of immutable stacks that has type `stack-eface` and represents a stack as a list of elements ordered from the top down.

c. Define a procedure `int-stack-test` that tests a stack package by (1) defining a `swap` procedure that swaps the top to elements of an integer stack; (2) defining a `stack->list` procedure that converts an integer stack to an integer list; and (3) returning the result of invoking `stack->list` on the result of calling `swap` on a stack that contains the elements `1` and `2`.

d. Define an interface `mstack-eface` for *mutable* stacks and repeat parts **b** and **c** for mutable stacks. ◁

▷ **Exercise 15.9** The $\texttt{pack}_{exist}$ form uses `up` and `down` procedures to explicitly convert between a concrete representation type and an opaque type name. Here we explore alternative ways to specify abstract vs. concrete types in $\texttt{pack}_{exist}$. These alternatives also work for the other forms of `pack` that we shall study.

a. One alternative to using `up` and `down` is to extend $\texttt{pack}_{exist}$ to have the form $(\texttt{pack}_{exist}\ I_{abs}\ T_{rep}\ T_{impl}\ E_{impl})$, in which the implementation type T_{impl} is explicitly supplied. For example, here is one way to express a pair implementation of points using the modified form of pack_{exist}:

```
(packexist point (pairof int int)
  (recordof (make-pt (-> (int int) point))
            (pt-x (-> (point) int))
            (pt-y (-> (point) int)))
  (record
    (make-pt (lambda ((x int) (y int)) (pair x y)))
    (pt-x (lambda ((p point)) (left p)))
    (pt-y (lambda ((p (pairof int int))) (right p))))))
```

Within E_{impl}, the abstract type `point` and the concrete type `(pairof int int)` are interconvertible.

Give a typing rule for this form of $\texttt{pack}_{exist}$. Your rule should not introduce `up` and `down` procedures. Use examples to justify the design of your rule.

b. An alternative to specifying T_{impl} in $\texttt{pack}_{exist}$ is to require the programmer to use explicit type ascriptions (via FL/XSP's `the`) to cast concrete to abstract types or vice versa. Explain, using examples.

c. Yet another way to convert between concrete and abstract types is to interpret the `define-datatype` form in a creative way. (The module system in Section 15.5 follows this approach.) Each constructor can be viewed as performing a conversion up to an opaque abstract type and each deconstructor can be viewed as performing a conversion down from this type. For example, here is a point-as-pair existential package declared via an alternative syntax for pack$_{exist}$ that replaces I_{abs} and T_{rep} by a `define-datatype` declaration:

```
(pack_exist
  (define-datatype point (pt (pairof int int)))
  (record
    (make-pt (lambda ((x int) (y int)) (pt (pair x y))))
    (pt-x (lambda ((p point)) (match p ((pt (pair x _)) x))))
    (pt-y (lambda ((p point)) (match p ((pt (pair _ y)) y))))))
```

Give a typing rule for this modified form of pack$_{exist}$.

d. Express the examples in Figure 15.5 and Figure 15.6 using the alternative approaches to existential types introduced above. ◁

15.3 Nonce Types

We have seen that the export restriction makes existential types an impractical way to express data abstraction in a typed language. The export restriction is a consequence of the fact that the abstract type name in an existential type and the local abstract type names introduced by unpack$_{exist}$ forms are not connected to each other or to the concrete type in any way. One way to address this problem is by replacing the abstract type names by globally unique type symbols that we call **nonce types**. We shall see that nonce types are in many ways a more flexible approach to abstract types than existential types, but suffer from problems of their own.

As an example, the type $T_{point-npkg}$ of one implementation of a point abstraction might be the nonce package type

```
(packof_nonce #1729
  (recordof
    (make-pt (-> (int int) #1729))
    (pt-x (-> (#1729) int))
    (pt-y (-> (#1729) int)))),
```

where `#1729` is the concrete notation for the globally unique nonce type for this particular implementation. Another point abstraction implementation would have the same packof$_{nonce}$ type, except that a different unique nonce type (say `#6821`) for that implementation would be substituted for each occurrence of

`#1729`. Nonce types are automatically introduced by the type checker and cannot be written down directly by the programmer.

Whereas (`packof`$_{exist}$ I_{abs} T_{impl}) is a binding construct declaring that the name I_{abs} may be used in the scope of T_{impl}, (`packof`$_{nonce}$ ν_{abs} T_{impl}) is not a binding construct. Rather, it effectively pairs the nonce type ν_{abs} with an implementation type in such a way that the two components can be unbundled by an elimination form (`unpack`$_{nonce}$). Like I_{abs}, ν_{abs} is an opaque name that hides a concrete representation type. But unlike I_{abs}, which has no meaning outside the scope of the `packof`$_{exist}$, ν_{abs} names a particular concrete representation throughout the entire program. It serves as a globally unique tag for guaranteeing that the operations of a data abstraction are performed only on the appropriate abstract values, regardless of how the operations and values are packaged and unpackaged. For example, a value of type `#1729` is necessarily created by the `make-pt` operation with type `(-> (int int) #1729)`, and it is safe to operate on this value with `pt-x` and `pt-y` operations having type `(pt-x (-> (#1729) int))`. In contrast, these operations are incompatible with abstract values having nonce type `#6821`.

The essence of the nonce type approach to abstract data types in FL/XSP is presented in Figure 15.7. The syntax for creating and eliminating nonce packages (using `pack`$_{nonce}$ and `unpack`$_{nonce}$) and typing nonce packages (`packof`$_{nonce}$) parallels the syntax for existential packages in order to facilitate comparisons.

For example, here is an expression $E_{pair-point-npkg}$ that describes a pair implementation of a point abstraction as a nonce package:

```
(pack_nonce point (pairof int int)
  (record
    (make-pt (lambda ((x int) (y int)) (up (pair x y))))
    (pt-x (lambda ((p point)) (left (down p))))
    (pt-y (lambda ((p point) (right (down p)))))))).
```

According to the [*npack*] typing rule, $E_{pair-point-npkg}$ could have the `packof`$_{nonce}$ type $T_{point-npkg}$ given earlier. Each application of the [*npack*] rule introduces a fresh nonce type ν (in this case `#1729`) that is not used in any other application of the [*npack*] rule. This nonce type replaces all occurrences of the programmer-specified abstract type name I_{abs} (in this case `point`) in E_{impl}. As in existential types, `up` and `down` procedures are used to mediate between the concrete and abstract types. Note that the Type domain must be extended to include nonce types (Nonce-Type), which are distinct from type identifiers and type reconstruction variables. They are instead a sort of newly generated type constant, similar to Skolem constants used in logic.

The following expression $E_{pair-point-test}$ is a use of the example package that

Syntax

$E ::= \ldots \mid$ $(\texttt{pack}_{nonce}\ \ I_{abs}\ \ T_{rep}\ \ E_{impl})$ [Nonce Package Introduction]
$\quad \mid$ $(\texttt{unpack}_{nonce}\ \ E_{pkg}\ \ I_{ty}\ \ I_{impl}\ \ E_{body})$ [Nonce Package Elimination]

$\nu \quad \in \quad$ Nonce-Type

$T ::= \ldots \mid \nu$ [Nonce Type]
$\quad \mid$ $(\texttt{packof}_{nonce}\ \ \nu\ \ T_{impl})$ [Nonce Package Type]

Type Rules

$$\frac{A[\texttt{up:(->}\ (T_{rep})\ \nu)\texttt{, down:(->}\ (\nu)\ \ T_{rep})] \vdash [\nu/I_{abs}]E_{impl} : T_{impl}}{A \vdash (\texttt{pack}_{nonce}\ \ I_{abs}\ \ T_{rep}\ \ E_{impl}) : (\texttt{packof}_{nonce}\ \ \nu\ \ T_{impl})} \quad [npack]$$

where ν is a fresh nonce type *[freshness condition]*
T_{rep} does not contain any `plambda`-bound identifiers *[rep restriction]*

$$\frac{\begin{array}{c}A \vdash E_{pkg} : (\texttt{packof}_{nonce}\ \ \nu\ \ T_{impl}) \\ A[I_{impl} : T_{impl}] \vdash [\nu/I_{ty}]E_{body} : T_{body}\end{array}}{A \vdash (\texttt{unpack}_{nonce}\ \ E_{pkg}\ \ I_{ty}\ \ I_{impl}\ \ E_{body}) : T_{body}} \quad [nunpack]$$

Type Equality

No new type equality rules.

Type Erasure

Same as for $\texttt{pack}_{exist}/\texttt{unpack}_{exist}$.

Figure 15.7: The essence of nonce types in FL/XSP.

is possible with nonce packages but not with existential packages:

```
(let ((pair-point-npkg E_pair−point−npkg))
  (let ((transpose (unpack_nonce pair-point-npkg t pair-point-ops
                     (with pair-point-ops
                       (lambda ((p t))
                         (make-pt (pt-y p) (pt-x p))))))
        (pt (unpack_nonce pair-point-npkg t pair-point-ops
                          (with pair-point-ops
                            (make-pt 1 2)))))
    (transpose pt))).
```

The [*nunpack*] typing rule can be used to show that $E_{pair-point-test}$ is well-typed. Since the same nonce type `#1729` is used within both occurrences of $\texttt{unpack}_{nonce}$, `transpose` has type `(-> (#1729) #1729)` and `pt` has type `#1729`, so `(transpose pt)` (as well as $E_{pair-point-test}$) has type `#1729`. As shown by [*nunpack*], the type identifier I_{ty} in $\texttt{unpack}_{nonce}$ allows the programmer to locally name the nonce type of E_{pkg}, which cannot be written down directly. There are no import or export restrictions in [*nunpack*]. The substitution $[\nu/I_{ty}]E_{body}$ converts all local type identifiers into nonce types that may safely enter and escape from $\texttt{unpack}_{nonce}$ because they are globally unique type symbols that denote the same implementation in all contexts.

Although [*nunpack*] has no restrictions, there are two restrictions in [*npack*]. The **freshness condition** requires that a different nonce type be used for each occurrence of $\texttt{pack}_{nonce}$ encountered in the type checking process. The restriction requires careful attention in practice. One way to formalize it in the type rules would be to modify the type rules to pass a nonce type counter through the type checking process in a single-threaded fashion and increment the counter whenever [*npack*] is used. In languages that allow separate analysis and compilation of modular units, nonce types could include a unique identifier of the computer on which type-checking was performed along with a timestamp of the time when type-checking took place.

The [*npack*] rule also has a **rep restriction** that prohibits the concrete representation type T_{rep} from containing any `plambda`-bound type identifiers. In the simple form of nonce packages that we are studying, this restriction prevents a single nonce type from being implicitly parameterized over any types that are not known when type checking is performed on the $\texttt{pack}_{nonce}$ expression. For example, consider the following expression, which would unsoundly be well-typed without the restriction:

```
(let ((make-wrapper
        (plambda (t)
          (pack_nonce abs t
            (record
              (wrap (lambda ((x t)) (up x)))
              (unwrap (lambda ((y abs)) (down y))))))))
  (let ((wrap-int (unpack_nonce (pcall make-wrapper int) wint ir
                    (select ir wrap)))
        (unwrap-bool (unpack_nonce (pcall make-wrapper bool) wbool br
                       (select br unwrap))))
    (unwrap-bool (wrap-int 3)))).
```

If `#251` is used as the nonce type in the `pack`$_{nonce}$ expression, then `wrap-int` has type `(-> (int) #251)`, `unwrap-bool` has type `(-> (#251) bool)`, and `(unwrap-bool (wrap-int 3))` has type `bool` even though it dynamically evaluates to the integer 3! The problem is that `#251` should not be a single nonce type but some sort of type constructor that is parameterized over `t`.

The key advantage of nonce packages over existential packages for expressing abstract types is that they have no export restriction. As illustrated by $E_{pair-point-test}$, values of and operations on the abstract type may escape from `unpack`$_{nonce}$ expressions. Programmers do not have to rearrange their programs or adopt an awkward programming style to prevent these from happening.

Despite their advantages over existential packages, nonce packages suffer from two drawbacks as a mechanism for abstract types:

1. *Difficulties with expressing nonce types.* The fact that nonce types cannot conveniently be written down directly by the programmer is problematic, especially in an explicitly typed language. For example, in FL/XSP, the programmer cannot write a top level definition of the form

   `(define pair-point-npkg` T $E_{pair-point-npkg}$`)`

   because there is no way to write down the concrete nonce type needed in T. This is not just an issue of type syntax; the programmer does not know which nonce type the type checker will choose when checking $E_{pair-point-npkg}$.

   One way to address this problem is to embed nonce packages in a language with implicit types, where type reconstruction can infer nonce package types that the programmer cannot express (see Exercise 15.14). This is the approach taken in SML, where the nonce-based `abstype` mechanism allows the local declaration of abstract data types. But in reconstructible languages, it is still sometimes necessary to write down explicit types, and

```
(let ((make-rat-impl
       (lambda ((b bool))
         (pack_nonce rat (pairof int int)
           (record
             (make-rat (lambda ((n int) (d int))
                         (up (if b (pair n d) (pair d n)))))
             (numer (lambda ((r rat))
                      (if b (left (down r)) (right (down r)))))
             (denom (lambda ((r rat))
                      (if b (right (down r)) (left (down r))))))))))
  (let ((leftist-rat (make-rat-impl true))
        (rightist-rat (make-rat-impl false)))
    ((unpack_nonce rightist-rat rty rops (select numer rops))
       ((unpack_nonce leftist-rat lty lops (select make-rat lops)) 1 2))
```

Figure 15.8: A form of abstraction violation that can occur with nonce types.

the inability to express nonce package types reduces expressivity in these cases.

Another alternative is to require that the abstract type name I_{abs} in (`pack`$_{nonce}$ I_{abs} T_{rep} E_{impl}) is a globally unique name that serves as a concrete nonce type. This lets the programmer rather than the type checker choose the abstract type name. In this case, the programmer *can* write down the abstract type name and there is no need for the local abstract type name in `unpack`$_{nonce}$. There are serious modularity problems with this approach, but it makes sense in restricted systems where all nonce packages are created at top level; see Exercise 15.15.

2. *Insufficient abstraction.* Nonce packages are sound in the sense that there are no **representation violations** — a well-typed program cannot encounter a run time type error. However, there is still a form of **abstraction violation** that can occur with nonce packages. An example of this is shown in Figure 15.8. The `make-rat-impl` procedure makes a rational number implementation, which in all cases represents a rational number as a pair of integers. However, it is abstracted over a boolean argument `b` that chooses one of two representations. When `b` is `true`, the numerator is the left element of the pair and the denominator is the right element; we will call this the "leftist representation." When `b` is `false`, a "rightist representation" is used, in which the numerator is on the right and the denominator is on the left.

   In the example, the `numer` procedure of the rightist representation is ap-

plied to a leftist rational with numerator 1 and denominator 2. Since nonce types are determined by static occurrences of $\texttt{pack}_{nonce}$ and there is only one of these in the example, the two dynamic invocations of `make-rat-impl` yield implementations that use the same nonce type for `rat`. Thus, the application is well-typed and at run time will return the value 2.

Thus, the nonce package system allows different implementations of a data abstraction to intermingle as long as they are represented via the same concrete type. Although this might seem reasonable in some cases, we normally expect an abstract type system to enforce the contract chosen by the designer of an abstraction. Enforcing the contract (and not just ensuring compatible representations) enables the abstraction and the clients to rely on important invariants that, among other things, ensure correctness of programs.

▷ **Exercise 15.10**

a. Would the expression in Figure 15.8. be well-typed if all occurrences of $\texttt{pack}_{nonce}$ and $\texttt{unpack}_{nonce}$ were replaced by $\texttt{pack}_{exist}$ and $\texttt{unpack}_{exist}$? Explain.

b. Below are three replacements for the body of the inner `let` in the example in Figure 15.8. For each replacement, indicate whether the whole example expression would be well-typed using (1) nonce packages and (2) existential packages[5]. For each case, discuss whether you think the type system does the "right thing" in that case.

i.
```
((unpack leftist-rat lty1 lops1 (select numer lops2))
   ((unpack leftist-rat lty2 lops2 (select make-rat lops1)) 1 2))
```

ii.
```
(unpack leftist-rat lty lops
  (unpack rightist-rat rty rops
    ((select numer rops) ((select make-rat lops) 1 2))))))
```

iii.
```
(unpack leftist-rat lty1 lops1
  (unpack leftist-rat lty2 lops2
    ((select numer lops2) ((select make-rat lops1) 1 2))))
```
◁

▷ **Exercise 15.11** As noted above, the inability to write down nonce types is incompatible with top level `define` declarations in FL/XSP, which require an explicit type to handle potentially recursive definitions. Design a top level definition mechanism for FL/XSP that enables the declaration of non-recursive global values. Illustrate how your mechanism can be used to give the global name `pair-point-npkg` to $E_{pair-point-npkg}$. ◁

[5]Assume that the occurrence of $\texttt{pack}_{nonce}$ in the figure is changed to $\texttt{pack}_{exist}$ for the existential case.

▷ **Exercise 15.12** If the abstract type name I_{abs} in (`pack`$_{nonce}$ I_{abs} T_{rep} E_{impl}) were required to be a globally unique name, then it could serve as a programmer-specified nonce type.

a. Give typing rules for versions of `pack`$_{nonce}$ and `unpack`$_{nonce}$ that are consistent with this interpretation. In this interpretation, there is no need for the identifier I_{ty} in `unpack`$_{nonce}$, since the unique name I_{abs} would be used instead.

b. Describe how to modify the type checking rules to verify the global uniqueness requirement.

c. Discuss the advantages and disadvantages of this approach to nonce types. Is it a good idea? ◁

▷ **Exercise 15.13** By design, the nonce package types of two syntactically distinct occurrences of `pack`$_{nonce}$ are necessarily different. For example, two nonce packages implementing a point abstraction would both have types of the form

```
(packof_nonce ν_point
  (recordof
    (make-pt (-> (int int) ν_point))
    (pt-x (-> (ν_point) int))
    (pt-y (-> (ν_point) int))))
```

but the nonce type ν_{point} would be different for the two packages. Nevertheless, the similarity in form suggests that it should be possible to abstract over different implementations of the same abstract type.

a. Suppose that we modify the syntax of `packof`$_{nonce}$ to be (`packof`$_{nonce}$ T T_{impl}) but do not change the typing rules [*npack*] and [*nunpack*] in any way. Given this change, write a `make-transpose` procedure that takes any nonce package implementing a point abstraction and returns a coordinate swapping procedure for that implementation. (Hint: use `plambda` to abstract over the nonce type.)

b. Explain why it is necessary to modify the syntax of `packof`$_{nonce}$ in order to write `make-transpose`. ◁

▷ **Exercise 15.14** In this exercise, we consider existential and nonce types in the context of type reconstruction by adding them to FL/R.

a. Nonce packages can be added to FL/R by extending it with the expression (`pack`$_{nonce}$ E_{impl}) and nonce types.

   i. Give an FL/R typing rule for the modified `pack`$_{nonce}$ form.

   ii. Describe how to extend the FL/R reconstruction algorithm to reconstruct the modified `pack`$_{nonce}$ form.

   iii. The `unpack`$_{nonce}$ expression and `packof`$_{nonce}$ type are not necessary in FL/R for most programs. Explain why.

iv. In versions of FL/XSP and FL/R supporting records (without row variables), write an FL/XSP program that cannot be re-expressed in FL/R due to the inability to write down an explicit nonce type.

b. Existential packages can be added to FL/R by extending it with the expressions (pack$_{exist}$ E_{impl}) and (unpack$_{exist}$ E_{pkg} T_{pkg} I_{impl} E_{impl}) and the type (packof$_{exist}$ I_{abs} T_{impl}). In the modified unpack$_{exist}$ form, the type T_{pkg} is the type of the expression E_{pkg}.

i. Give FL/R typing rules for the modified pack$_{exist}$ and unpack$_{exist}$ forms.

ii. Describe how to extend the FL/R reconstruction algorithm to reconstruct the modified pack$_{exist}$ and unpack$_{exist}$ forms.

iii. Explain why it is necessary for the reconstruction system to be explicitly given the type T_{pkg} of the existential package expression E_{pkg}. Why can't it reconstruct this package type?

c. What changes would need to be made above to handle existential and nonce packages with parameterized types? ◁

▷ **Exercise 15.15** Many languages support abstract types that can only be declared globally. Here we explore an abstract type mechanism introduced by a top level `define-cluster` form. For simplicity, we assume that programs have the form:

```
(program
  (define-cluster I_impl_1 I_abs_1 T_rep_1 E_impl_1)
  ⋮
  (define-cluster I_impl_k I_abs_k T_rep_k E_impl_k)
  E_body)
```

a. One interpretation of `define-cluster` is given by the following desugaring for the above `program` form:

```
(let ((I_impl_1 (pack_exist I_abs_1 T_rep_1 E_impl_1))
      ⋮
      (I_impl_k (pack_exist I_abs_k T_rep_k E_impl_k)))
  (unpack_exist I_impl_1 I_abs_1 I_impl_1
    ⋮
    (unpack_exist I_impl_k I_abs_k I_impl_k
      E_body)))
```

Would the interpretation be any different if all occurrences of pack$_{exist}$ and unpack$_{exist}$ were replaced by pack$_{nonce}$ and unpack$_{nonce}$, respectively?

b. Give a direct typing rule for the `program` form with `define-cluster` declarations that gives the same static semantics as the above desugaring.

c. An alternative desugaring for the `program` form is:

```
(let ((I_impl_1 (pack_exist I_abs_1 T_rep_1 E_impl_1)))
  (unpack_exist I_impl_1 I_abs_1 I_impl_1
    ⋮
    (let ((I_impl_k (pack_exist I_abs_k T_rep_k E_impl_k)))
      (unpack_exist I_impl_k I_abs_k I_impl_k
        E_body))))
```

What advantage does this desugaring have over the previous one? Give an example where this desugaring would be preferred.

d. Give a direct typing rule for mutually recursive top level `define-cluster` declarations.

e. Give a simple example program where mutually recursive clusters are useful.

f. Suppose that we want to be able to locally define a collection of clusters anywhere in a program via the following form:

```
(let-clusters ((I_clust I_abs T_rep E_impl)*) E_body)
```

Discuss the design issues involved in specifying the semantics of `let-clusters`. ◁

15.4 Dependent Types

As we saw with existential packages, the inability to express "the type exported by *this* package" makes many programs awkward to write. Nonce types provide a way to express this idea but suffer from two key drawbacks: (1) nonce types are thorny to express: either they are chosen by the system, and are inconvenient or impossible for the programmer to write down explicitly; or they are chosen by the programmer, in which case their global uniqueness requirement is at odds with modularity; and (2) they allow abstract types from different instances of the same syntactic package expression to be confused.

There is another option: use a structured name to select a type out of a package just as components are selected out of a product. In particular, we introduce a new type form `(dtype` E_{pkg}`)` to mean "the type exported by the package denoted by E_{pkg}."[6] It is an error if E_{pkg} does not denote a package. As with nonce packages, such a package may be viewed as a pair containing a type and a value. The difference is that the programmer has a convenient way to express the type component outside the scope of an `unpack` expression. The type defined by a package is sometimes referred to as the package's **carrier**.

[6] In a practical system in which packages can export multiple abstractions, we could write `(dselect` I E`)` just as we select named values from records.

A type that contains a value expression is called a **dependent type** because the type it represents *depends* in some sense on the value. The reader may be justifiably concerned about this unholy commingling of types and values, especially with respect to static type checking. As we shall see, dependent types raise some nettlesome issues that the language designer must address in order to reap the expressiveness benefits.

15.4.1 A Dependent Package System

This section explores a simple package system that uses dependent types to express abstract types (see Figure 15.9). We shall use the term **dependent package** to refer to a package based on dependent types. The `pack`$_{depend}$ and `packof`$_{depend}$ forms work in a way similar to the previous package systems. For example, we can define a pair point implementation (`pair-point-dpkg`) and a procedural point implementation (`proc-point-dpkg`) exactly as in Section 15.2 except that we replace all occurrences of `pack`$_{exist}$ by `pack`$_{depend}$. Both of the new packages will have the following interface type:

```
(define-type point-dface
  (packof_depend point
    (recordof
       (make-pt (-> ((x int) (y int)) point))
       (pt-x (-> ((p point)) int))
       (pt-y (-> ((p point)) int)))))
```

Notice that procedure types have been extended to include the names of the formal parameters: the arrow type constructor `->` is now a binding form in which the formal names are available in the return type. We shall see below how this is used.

As with existential packages, `pack`$_{depend}$ and `unpack`$_{depend}$ have an import restriction that prevents the local name of the abstraction from capturing an existing type name. As before, this restriction can be eliminated by automatically α-renaming programs to make all logically distinct type identifiers unique. The `unpack`$_{depend}$ form has an additional restriction that we will discuss in more detail later.

As with nonce packages (but not existential packages), dependent packages have no export restriction, so abstract values may exit the scope of an `unpack`$_{depend}$ expression. But unlike the nonce type system, free references to the abstract type name exported by `unpack`$_{depend}$ are replaced by a user-writable type: a dependent type that records the program code that generated the package exporting the type. For example, what is the type $T_{pair-point}$ in the following definition?

Syntax

$E ::= \ldots \mid$ `(pack`$_{depend}$ I_{abs} T_{rep} E_{impl}`)` [Dependent Package Introduction]
$\mid$ `(unpack`$_{depend}$ E_{pkg} I_{ty} I_{impl} E_{body}`)` [Dependent Package Elimination]

$T ::= \ldots \mid$ `(-> ((`I T`)*)` T`)` [Dependent Arrow Type]
$\mid$ `(packof`$_{depend}$ I_{abs} T_{impl}`)` [Dependent Package Type]
$\mid$ `(dtype` E_{pkg}`)` [Dependent Type Selection]

Type Rules

$$\frac{A[\texttt{up:(-> ((x } T_{rep}\texttt{))} \ I_{abs}\texttt{), down:(-> ((x } I_{abs}\texttt{))} \ T_{rep}\texttt{)}] \vdash E : T}{A \vdash \texttt{(pack}_{depend}\ I_{abs}\ T_{rep}\ E\texttt{)} : \texttt{(packof}_{depend}\ I_{abs}\ T\texttt{)}} \quad [dpack]$$

where $I_{abs} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E]\!]\}$ *[import restriction]*

$$\frac{\begin{array}{c}A \vdash E_{pkg} : \texttt{(packof}_{depend}\ I_{abs}\ T_{impl}\texttt{)} \\ A[I_{impl} : [\texttt{(dtype } E_{pkg}\texttt{)}/I_{abs}]\ T_{impl}] \vdash [\texttt{(dtype } E_{pkg}\texttt{)}/I_{ty}]\ E_{bdy} : T_{bdy}\end{array}}{A \vdash \texttt{(unpack}_{depend}\ E_{pkg}\ I_{ty}\ I_{impl}\ E_{bdy}\texttt{)} : T_{bdy}} \quad [dunpack]$$

where $I_{abs} \notin \{(FTV\ A(I)) \mid I \in FreeIds[\![E_{body}]\!]\}$ *[import restriction]*
E_{pkg} must be pure *[purity restriction]*

$$\frac{A[I_1 : T_1\texttt{,}\ \ldots\texttt{,}\ I_n : T_n] \vdash E : T}{A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ E\texttt{)} : \texttt{(-> ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ T\texttt{)}} \quad [\lambda]$$

$$\frac{\begin{array}{c}A \vdash E_{rator} : \texttt{(-> ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ T_{body}\texttt{)} \\ \forall_{i=1}^{n} .\ A \vdash E_i : T_i\end{array}}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)} : [_{i=1}^{n} E_i/I_i]\,T_{body}} \quad [apply]$$

where $I_i \in FreeIds[\![T_{body}]\!]$ implies E_i is pure. *[purity restriction]*

$$\frac{\begin{array}{c}\forall_{i=1}^{n} .\ A \vdash E_i : T_i \\ A[I_1 : T_1\texttt{,}\ \ldots\texttt{,}\ I_n : T_n] \vdash E_{body} : T_{body}\end{array}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : [_{i=1}^{n} E_i/I_i]\,T_{body}} \quad [let]$$

where $I_i \in FreeIds[\![T_{body}]\!]$ implies E_i is pure *[purity restriction]*

Rules for `letrec` and `with` are similar and left as exercises.

Type Erasure Same as for `pack`$_{exist}$/`unpack`$_{exist}$ and `pack`$_{nonce}$/`unpack`$_{nonce}$.

Type Equality

$$\frac{\forall_{i=1}^{n} .\ I_i' \notin FreeIds[\![T_{body}]\!]}{\begin{array}{l}\texttt{(-> ((}I_1\ T_1\texttt{)}\ldots\texttt{(}I_n\ T_n\texttt{))}\ T_{body}\texttt{)} \\ \equiv \texttt{(-> ((}I_1'\ T_1\texttt{)}\ldots\texttt{(}I_n'\ T_n\texttt{))}[_{i=1}^{n} I_i/I_i']\,T_{body}\texttt{)}\end{array}} \quad [\texttt{->}=]$$

$$\texttt{(packof}_{depend}\ I\ T\texttt{)} = \texttt{(packof}_{depend}\ I'\ [I'/I]\,T\texttt{)} \quad [dpackof=]$$

$$\frac{E_1 =_{depends} E_2}{\texttt{(dtype } E_1\texttt{)} = \texttt{(dtype } E_2\texttt{)}} \quad [dtype=]$$

where $=_{depends}$ is discussed in the text.

Figure 15.9: The essence of dependent types in FL/XSP.

```
(define pair-point T_pair-point
  (unpack_depend pair-point-dpkg point point-ops
    (with point-ops (make-pt 1 2)))
```

The return type of the `make-pt` procedure in `pair-point-dpkg` is `point`. According to the [*dunpack*] rule, `(dtype pair-point-dpkg)` is substituted for this type. So the invocation of `make-pt`, the `with` expression, and the $\texttt{unpack}_{depend}$ expression all have the type $T_{pair-point}$ = `(dtype pair-point-dpkg)`.

The expressive power of dependent types is illustrated by the `make-transpose` procedure:

```
(define make-transpose T_make-transpose
  (lambda ((point-dpkg point-dface))
    (unpack_depend point-dpkg pt point-ops
      (with point-ops
        (lambda ((p pt))
          (make-pt (pt-y p) (pt-x p)))))))
```

What is the type $T_{make-transpose}$ of this procedure? It is a procedure that takes a package that implements the `point-dface` interface and returns a procedure that takes a point implemented by the given package and returns a point from the same package with swapped coordinates:

```
(-> ((point-dpkg point-dface))
  (-> ((p (dtype point-dpkg))) (dtype point-dpkg)))
```

It should now be clear why the `->` type constructor is a binding form in a dependent type system: the return type can depend on the value of a parameter (such as `point-dpkg` above), so we need a way to refer to the parameter. Not all parameter names are actually used in this fashion. For instance, the parameter name `p` is ignored. We shall refer to values of the dependent arrow type as **dependent procedures**.

What happens when we apply a dependent procedure? Consider the application $E_{transpose-test}$:

```
((make-transpose pair-point-dpkg) pair-point-dpkg)
```

By the [*apply*] rule, when we apply `make-transpose` to a package satisfying `point-dface`, we substitute the actual argument expression for the formal parameter `point-dpkg` in the type

```
(-> ((p (dtype point-dpkg))) (dtype point-dpkg))
```

to give the type

```
(-> ((p (dtype pair-point-dpkg))) (dtype pair-point-dpkg))
```

Since `pair-point` has type `(dtype pair-point-dpkg)`, $E_{transpose-test}$ is a well-typed expression denoting a value with type `(dtype pair-point-dpkg)`.

Similarly, free references to `let`-bound variables are substituted away in the result type of a `let`. Now that types refer to values, anytime a value escapes the scope of a variable binding, that variable is replaced with its definition expression if it occurs free in the value's type. The rules for `letrec` and `with` are similar to the [*apply*] and [*let*] rules and are left as exercises.

It is instructive to revisit the rational number example from Figure 15.8 in the context of dependent types. Consider the following two expressions:

```
((unpack_depend rightist-rat rty rops (select numer rops))
   ((unpack_depend leftist-rat lty lops (select make-rat lops))
    1 2))

((unpack_depend leftist-rat lty2 lops2 (select numer lops2))
   ((unpack_depend leftist-rat lty1 lops1 (select make-rat lops1))
    1 2))
```

With dependent types, the first expression is ill-typed because an attempt is made to apply a procedure of type `(-> ((r (dtype rightist-rat))) int)` to a value of type `(dtype leftist-rat)`. However, the second expression is well-typed since the procedure parameter and the argument point both have type `(dtype leftist-rat)`. So dependent types are able to catch the abstraction violation in the first expression while permitting operations and values of the same abstract type to interoperate outside of `unpack`$_{depend}$ in the second expression. In contrast, neither expression is well-typed with existential packages (due to the export restriction) and both expressions are well-typed with nonce types (which cannot distinguish different instantiations of a `pack`$_{nonce}$ expression).

15.4.2 Design Issues with Dependent Types

Dependent types are clearly very powerful. However, care must be taken to ensure that a dependent type system is sound. Moreover, programmers typically expect that a statically typed language will respect the **phase distinction**: the well-typedness of their programs will be verified in a first (terminating) type-checking phase that runs to completion before the second (possibly non-terminating) run time computation phase begins. We shall see that in some designs for dependent types these phases are interleaved and type checking may not terminate.

There are several design dimensions in systems with dependent types. One dimension involves how types are bundled up into and extracted from packages.

In the system we have studied so far, dependent packages have a single type that is extracted via `dtype`, but more generally, a dependent package can have several type components. These are typically named and extracted in a record-like fashion. We will see an example of this in the module system in Section 15.5.

Another design dimension involves the details of performing substitutions in the typing rules. Some alternatives to the rules presented in Figure 15.9 are explored in the exercises for this section.

Perhaps the most important design dimension is type equality on dependent types: when is the type `(dtype` E_1`)` considered equal to `(dtype` E_2`)`? There are a range of choices here that have significant impact on the properties of the language. We spend the rest of this section discussing some of the options.

One option is to treat types as first-class run time entities and dependent packages as pairs of a type and a value. Such pairs are known as **strong sums**[7] because the type component serves as a tag that can be used for dynamic dispatch. In this interpretation, `(dtype` E`)` extracts the type component of the pair, which is convertible with the package's representation type. Since type checking and evaluation are inextricably intertwined in this design, there is no phase distinction. Furthermore, abstraction is surrendered by making representation types transparent. The PEBBLE language [BL84] took this approach and used a lock and key mechanism (similar to that described in Section 15.2) to support data abstraction.

Another option is to consider `(dtype` E_1`)` to be the same as `(dtype` E_2`)` if the expressions E_1 and E_2 are "equal" for a suitable notion of equality. This is the approach taken in the [*dtype*=] rule of Figure 15.9, which is parameterized over a notion of equality ($=_{depends}$) that is not defined in the figure. There are two broad approaches to defining $=_{depends}$:

- *Value equality:* At one end of the spectrum, we can interpret two expressions to be the same under $=_{depends}$ if they denote the same package in the usual dynamic semantics of expressions. In the general case, this implies that type checking may require expression evaluation. As with strong sums, type checking in this approach may not terminate or may need to be done at run time. Even worse, determining if two package values are the same in general requires comparing procedures for equality, which is uncomputable! In practice, some computable conservative approximation for procedure equality must be used. Such an approximation must necessarily distinguish procedures that are denotationally equivalent. A common technique is to associate a unique identifier with each procedure value and to say that two procedures are equal only if they have the same identifier.

[7] In contrast, existential packages are sometimes called **weak sums**.

- *Static equality:* In order to preserve static type checking (with no run time requirements and a guarantee that type checking terminates), we desire a definition of $\equiv_{depends}$ that is statically computable. The easiest solution is to say that two `dtype`s are equal if their expressions are textually the same (more precisely, if they have equal abstract syntax trees). This is obviously statically computable, and this simple solution is the one that we adopt here.

  There are other choices for $\equiv_{depends}$ besides textual equality. We could, for example, allow the expressions in equivalent `dtype`s to admit α-renaming. We could allow certain substitutions to take place (say if expressions are equivalent after their `let`s are substituted away). As long as the equivalence is statically computable and ensures that expressions that denote different values are not equal, the system is sound. We refer to any such system as a **static dependent type (SDT)** system.

For our system to be sound, we need to guarantee that a value that uses a type exported from one package cannot masquerade as a value of some other type, e.g., a point from the `proc-point-dpkg` cannot be passed to an operation from `pair-point-dpkg`.

One requirement is that programs must be α-renamed on input. Otherwise, it would be possible for textually identical expressions to mean different things in different contexts. Our substitutions when a value exits a binding construct are not enough. Consider the following code:

```
(let ((trans (make-transpose pair-point-dpkg))
      (pair-point-dpkg proc-point-dpkg))
  (trans (unpack_depend pair-point-dpkg point pt-ops
           (with pt-ops
             (make-pt 1 2)))))
```

If the new binding of `pair-point-dpkg` were not α-renamed, then it would be confused with the `pair-point-dpkg` that occurs free in the type of `trans`, which would be unsound.

A second requirement is that in a dependent type `(dtype` E_{pkg}`)`, the expression E_{pkg} be pure — i.e., it must not vary with state. This is true whether a language uses value or static equality. In a language with mutation, the same syntactic expression might have different meanings at different times. For example, consider the following expression:

```
(let ((c (cell pair-point-pkg)))
  (let ((p (unpack (^ c) pt ops
             ((select make-pt ops) 1 2))))
    (begin
      (:= c proc-point-pkg)
      ((unpack (^ c) pt ops
         (select pt-x ops)) p))))
```

When cell `c` contains `pair-point-pkg`, the pair point `p` is created and has type `(dtype (^ c))`. Then `c` is modified to contain `proc-point-pkg`, at which time the procedural point operation `pt-x` (with type `(-> ((dtype (^c))) int)` is applied to `p`. It should be an abstraction violation to apply the procedural point operation `pt-x` to a pair point, but the type system encounters no error, because the argument type of `pt-x` and the type of `p` are both `(dtype (^c))`. The type system does not track the fact that `(^c)` refers to different packages at different times.

We address this problem by instituting a **purity restriction** on E_{pkg} in the [*dunpack*] rule, which introduces all dependent types. Of course, it is undecidable to know when an expression is pure. A simple conservative approximation is to require that E_{pkg} be a "syntactic value," a notion introduced in Section 8.2.5 and used in polymorphic types (Section 13.2) and in the type reconstruction system of FL/R (Section 14.2). However, we will see in Section 15.5 that this approximation prohibits many expressions we would like to write. A better alternative is to use an effect system (see Chapter 16) to conservatively approximate pure expressions.

▷ **Exercise 15.16**

a. Write typing rules for `letrec` and `with` in a language with dependent types.

b. Dependent types permit code to be abstracted over particular implementations of a data abstraction. The typing rules of this section require that such abstractions be curried by the programmer because of the scoping of parameter names in procedure types. The `make-transpose` procedure studied above is an example of such currying. In its type,

```
(-> ((point-dpkg point-dface))
  (-> ((p (dtype point-dpkg)))
    (dtype point-dpkg))),
```

the argument type of the transposition procedure refers to `point-dpkg`.

Suppose that we want to modify the typing rules for a dependently typed language to implicitly curry multiple parameters — i.e., to allow the types of later parameters to refer to the names of earlier parameters. For example, in the modified system, an uncurried form of `make-transpose` could have the type

```
(define-type uncurried-make-transpose-type
  (-> ((point-dpkg point-dface) (p (dtype point-dpkg)))
    (dtype point-dpkg)))
```

Curiously, the [λ] rule does not need to change to support implicitly curried parameters.[8] The [*apply*] rule, however, must change. Write a new [*apply*] rule that supports procedure formals that refer to previous parameters. E.g., a procedure of type `uncurried-make-transpose-type` must be applied to two arguments where the type of the second argument depends on the value of the first. ◁

▷ **Exercise 15.17** Del Sharkmon suggests the following alternative [*dunpack'*] type rule that does dependent type substitutions on the way out of $\texttt{unpack}_{depend}$ rather than on the way in.

$$\frac{\begin{array}{c} A \vdash E_{pkg} \ : \ (\texttt{packof}_{depend} \ \ I_{abs} \ \ T_{impl}) \\ A[I_{impl} : [I_{ty}/I_{abs}]T_{impl}] \vdash E_{bdy} \ : \ T_{bdy} \end{array}}{A \vdash (\texttt{unpack}_{depend} \ \ E_{pkg} \ \ I_{ty} \ \ I_{impl} \ \ E_{bdy}) \ : \ [(\texttt{dtype} \ \ E_{pkg})/I_{ty}]T_{bdy}} \qquad [dunpack']$$

a. Using dependent packages, redo Exercise 15.10(b) using (1) the original [*dunpack*] rule and (2) Del's [*dunpack'*] rule. Which rule do you think is better and why?

b. Del claims that [*dunpack'*] is better than [*dunpack*] in some situations where E_{pkg} contains side effects. Write an expression that is well-typed with [*dunpack'*] but not [*dunpack*].

c. For any expression that is well-typed with [*dunpack'*] but not [*dunpack*], it is possible to make the expression well-typed by naming E_{pkg} with a `let`. Show this in the context of your expression from the previous part. ◁

▷ **Exercise 15.18** Ben Bitdiddle looks at the typing rules in Figure 15.9 and your solution to Exercises 15.16 and 15.17 and complains that all the substitutions make him dizzy. He suggests leaving them all out except for those in the [*apply*] rule. Under what assumptions is his idea sound? Write a type safe program that type checks under the given rules but does not type check under Ben's. ◁

15.5 Modules

15.5.1 An Overview of Modules and Linking

It is desirable to decompose a program, especially a large one, into modular components that can be separately written, compiled, tested, and debugged.

[8] In a system with kinds, the [λ] rule *would* change because we would need the scope to be manifest to verify that dependent types are well-formed.

Such components are typically called **modules** but are also known as packages, structures, units, and classes.[9] Ideally, each individual module is described by an **interface** that specifies the components required by the module from the rest of the program (the **imports**) and the components supplied by the module to the rest of the program (the **exports**). Interfaces often list the names and types of imported and exported values along with informal English descriptions of these values. Such interfaces make it possible for programmers to implement a module without having to know the implementation details of other modules. They also make it possible for a compiler to check for type consistency within a single module.

In most module systems, modules are record-like entities that have both type and value components. In this respect, they are more elaborate versions of the packages we have studied. Indeed, modules are often used as a means of expressing abstract types in addition to being a mechanism for decomposing a program into parts, which is why we study them here. The key difference between the modules discussed here and the packages we studied earlier is that there is an expectation that modules can be separately written and compiled and later combined to form a whole program.

The process of combining modules to form a whole program is called **linking**. The specification for how to combine the modules to form a program is written in a **linking language**. Linking is typically performed in a distinct **link time** phase that is performed after all the individual modules are compiled (compile time) but before the entire program is executed (run time).

A crude form of linking involves hard-wiring the file names for imported modules within the source code for a given module. In more flexible approaches, a module is parameterized over names for the imported modules and the linking language specifies the actual modules to be used for the parameters. Ideally, the linking language should check that the interface types of the actual module parameters are consistent with those of the formal module parameters. In this case, the linking language is effectively a simple typed programming language.

Often, a linking language simply lists the modules to be combined. For example, the object files of a C program are linked by supplying a list of file names to the compiler. A linking language can be made more powerful by adding other programming language features that allow more computation to be performed during the linking process. But the desire to make linking languages more expressive is often in tension with the desire to guarantee that (1) the linking process terminates and (2) mere mortals can reliably understand and use

[9] In many languages, such as C, files serve as de facto modules, but in general the relationship between source files and program modules can be more complex.

the sophisticated types that often accompany more expressive linking languages.

An extreme design point is to make the linking language the same as the base language used to express the modules themselves. Such **first-class module systems** are powerful because arbitrary modules can be created at run time and the decision of which module to import can be based on dynamic conditions. These systems blur the distinction between link time and run time, and for any Turing-complete base language, the linking process may not terminate. For these reasons, linking is usually specified in a different language from the base language that is suitably restricted to guarantee termination and designed to link programs in a separate phase. In such **second-class module systems**, modules are not first-class values that can be manipulated in the base language. The module systems of the CLU, SML and OCAML languages are examples of expressive second-class module systems.

15.5.2 A First-Class Module System

We conclude this chapter by presenting a first-class module system based on static dependent types and incorporating extensions for sum-of-product data type definitions, pattern matching, higher-order abstractions (type constructors), and multiple abstract type and value definitions in a single module. Our module system illustrates how the simple ideas presented earlier can be combined into a more realistic system, and also how delicate a balance must be struck to make the system both useful and correct.

We add the module features to FL/R to yield the language FL/RM. A language where types are reconstructed is far more convenient for programming than an explicitly typed language, where the explicit types can be tedious and challenging to write. However, as we shall see below, certain types (the types of modules) *must* be declared because they cannot be reconstructed. This is not a big drawback since explicit module types are important in software engineering for documenting module interfaces.

15.5.2.1 The Structure of FL/RM

The syntax and semantics of FL/RM are presented in Figures 15.10–15.13. Figure 15.10 presents the new expression and type syntax that FL/RM adds to FL/R. A module is a record-like entity whose abstract type components are declared via `define-datatype` (discussed below) and whose value components are declared via `define`. The type of a `module` is a `moduleof` type that records the abstract types and the types of each of the named value components. The named type and value components of the module denoted by E_{mod} are made

available to a client expression E_{body} via the `(open` E_{mod} E_{body}`)` form, which is the module analog of the `with` form for typed records. Programmers load separately compiled modules from an external storage system via the `load` construct. Loading is described in more detail in Section 15.5.2.5.

Dependent types have the form `(dselect` I_{tc} E_{mod}`)`, which selects the abstract type constructor named I_{tc} from the module denoted by E_{mod}. As in Section 15.4, dependent procedure types of the form `(->` `((`I T`)*)` T`)` must be supported. Parameter names may be omitted from non-dependent procedure types (i.e., where the parameter names don't appear in the return type); it is assumed that the system treats these as dependent procedure types with fresh parameter names. The type syntax also includes type constructor applications of the form `(`TC T`*)`, where a type constructor TC is either an identifier or the result of extracting a type constructor from a module. In both cases, the type constructor is presumed to be either a name the programmer declares via `define-datatype` or a predefined type constructor name like `listof`.

Conventional type reconstruction cannot, in general, infer module types. For this reason, optional declarations have been added to the syntax for `lambda` and `letrec` expressions and `define` declarations. (We use the convention that syntax enclosed by square brackets is optional.) Whenever an identifier introduced by `lambda`, `letrec`, or `define` denotes a module (or a value whose type includes a module type), that identifier *must* have its type supplied. If the type is omitted, the program will not type check.

15.5.2.2 Datatypes and Pattern Matching

The `define-datatype` form is a typed version of the `define-data` sum-of-products declaration introduced in Section 10.4. It declares a parameterized abstract type constructor along with a collection of constructor procedures and their associated deconstructors. For example,

```
(define-datatype (treeof t)
  (leaf)
  (node t (treeof t) (treeof t)))
```

declares a binary tree type constructor, `treeof`, that is parameterized over the node value type `t`. It also declares two constructor/deconstructor pairs with the types shown in Figure 15.14. The constructor procedure types are quantified over the type constructor parameter `t`. The deconstructor procedure types are additionally quantified over the return type `r` of the deconstructor. The types of the constructor and deconstructor procedures associated with a `define-datatype` declaration are formalized by the $\oplus$ operator (see Figure 15.12), which extends

Syntax

E ::= `...` | `(lambda (`LF`*) `E`)`
| `(letrec ((`I `[`T`] `E`)*) `E`)`
| `(match `E_{disc} `(`P E`)*)`
| `(module `DD`* `D`*)` [Module introduction]
| `(open `E_{mod} E_{body}`)` [Module elimination]
| `(load `S`)` [Load compiled code]

LF ::= I | `(`I T`)` [Lambda Formals]
P ::= L | I | `_` | `(`I P`*)` [Patterns]
DD ::= `(define-datatype `AB `(`I_{cnstr} T_{comp}`*)*)` [Datatype Definition]
D ::= `(define `I `[`T`] `E`)` [Value Definition]
AB ::= `(`I_{tc} I_{param}`*)` [Abstract Type]
UID ::= System dependent [Unique File Identifier]

T ::= `...` | `(moduleof (`AB`*) (`I TS`)*)` [Module Type]
| `(`TC T`*)` [Type Constructor Application]
| `(-> ((`I T`)*) `T`)` [Dependent Proc Type]
| `(-> (`T`*) `T`)` [Non-dependent Proc Type]
| `(dselect `I_{tc} E_{mod}`)` [Dependent Type]
TC ::= I_{tc} | `(dselect `I_{tc} E_{mod}`)` [Type Constructor]
TS ::= T | `(generic (`I`*) `T`)` [Type Schema]

Sugar

The usual FL/R desugaring function $\mathcal{D}$ is extended as follows:

$$\mathcal{D}[\![\texttt{(match } E_{disc} \texttt{ (}P\ E\texttt{)*)}]\!] = \mathcal{D}[\![\mathcal{D}_{\text{match}}[\![\texttt{(match } E_{disc} \texttt{ (}P\ E\texttt{)*)}]\!]]\!]$$

where $\mathcal{D}_{\text{match}}$ is the pattern matching desugarer presented in Figure 10.30 with the modification to $\texttt{equal}_L$ described in the text.

Figure 15.10: Syntax for the module system of FL/RM.

Type Rules

$$\frac{\forall_{j=1}^{m} \ . \ (A \oplus [DD_1, \ldots, DD_n])[I_1 : T_1, \ldots, I_m : T_m] \vdash E_j \ : \ T_j}{\begin{array}{l} A \vdash \texttt{(module } DD_1 \ \ldots \ DD_n \\ \qquad \texttt{(define } I_1 \ [T_1] \ E_1\texttt{)} \ldots \texttt{(define } I_m \ [T_m] \ E_m\texttt{))} \\ \quad : \texttt{(moduleof ((}I_{tc_1} \ I_{p_{1,1}} \ldots I_{p_{1,a_1}}\texttt{)} \ldots \texttt{(}I_{tc_n} \ I_{p_{n,1}} \ldots I_{p_{n,a_n}}\texttt{))} \\ \qquad\qquad \texttt{(}I_1 \ (mgen \ T_1)\texttt{)} \ldots \texttt{(}I_m \ (mgen \ T_m)\texttt{)))} \end{array}} \quad [module]$$

where $DD_i = \texttt{(define-datatype (}I_{tc_i} \ I_{p_{i,1}} \ldots I_{p_{i,a_i}}\texttt{) ...)}$
$(mgen \ T) = \texttt{(generic (}J^*\texttt{) } T\texttt{)}$
$J^* = FTV \ (T) - FTE(A) - \{I_{tc_1}, \ldots, I_{tc_n}\}$

$$\frac{\begin{array}{c} A \vdash E_m \ : \ \texttt{(moduleof ((}I_{tc_1} \ldots\texttt{)} \ldots \texttt{(}I_{tc_n} \ldots\texttt{)) (}I_1 \ TS_1\texttt{)} \ldots \texttt{(}I_m \ TS_m\texttt{))} \\ A[I_1 : sub(TS_1), \ldots, I_m : sub(TS_m)] \vdash sub(E_b) \ : \ T_b \end{array}}{A \vdash \texttt{(open } E_m \ E_b\texttt{)} \ : \ T_b} \quad [open]$$

where $sub(X) = [_{i=1}^{n} \texttt{(dselect } I_{tc_i} \ E_m\texttt{)}/I_{tc_i}]\text{X}$ *[Dependent type introduction]*
E_m is pure *[Purity restriction]*

$$\frac{\vdash \texttt{contents [}S\texttt{]} \ : \ T}{A \vdash \texttt{(load } S\texttt{)} \ : \ T} \quad [load]$$

The [*proc*], [*apply*], [*let*], and [*letrec*] rules are similar to those in Section 15.4 and are left as exercises.

Type Equality

$$\frac{\forall_{i=1}^{n} \ . \ T_i \ = T_i\,'}{\begin{array}{c} \texttt{(moduleof (}AB_1 \ \ldots AB_k\texttt{) ((}I_1 \ T_1\texttt{) } \ldots\texttt{(}I_n \ T_n\texttt{)))} \\ = \texttt{(moduleof (}AB_1 \ \ldots AB_k\texttt{) ((}I_1 \ T_1\,'\texttt{) } \ldots\texttt{(}I_n \ T_n\,'\texttt{)))} \end{array}} \quad [moduleof{=}]$$

(This is more restrictive than necessary; see discussion in text.)

$$\frac{E_{m_1} =_{depend} E_{m_2}}{\texttt{(dselect } I \ E_{m_1}\texttt{)} = \texttt{(dselect } I \ E_{m_2}\texttt{)}} \quad [dselect{=}]$$

where $=_{depend}$ is textual equality and all programs are appropriately α-renamed.

$$\frac{\begin{array}{c} \{I_1, \ldots, I_n\} = \{I_1\,', \ldots, I_n\,'\} \\ T = T' \end{array}}{\texttt{(generic (}I_1 \ldots I_n\texttt{) } T\texttt{)} = \texttt{(generic (}I_1\,' \ldots I_n\,'\texttt{) } T'\texttt{)}} \quad [generic{=}]$$

$$T = \texttt{(generic () } T\texttt{)} \quad [type\text{-}generic{=}]$$

The [->=] rule from Figure 15.9 is used for dependent arrow types.

Figure 15.11: Static semantics for the module system of FL/RM.

The notation $A \oplus DD$ denotes a type environment such that

$$(A \oplus DD)(I) = \begin{cases} \texttt{(generic (}I_{p_1}\ \ldots\ I_{p_m}\texttt{)} \\ \quad \texttt{(->\ (}T_1\ \ldots\ T_n\texttt{)\ (}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)))}, \\ \quad \textbf{if } I = I_{con} \textbf{ and } DD = \texttt{(define-datatype (}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)} \\ \qquad\qquad \ldots\ \texttt{(}I_{con}\ T_1\ \ldots\ T_n\texttt{)}\ \ldots\texttt{)} \\ \\ \texttt{(generic (}I_{ret}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)\ ;\ }I_{ret}\texttt{ is fresh} \\ \quad \texttt{(->\ ((}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)\ \ \ ; datum} \\ \qquad \texttt{(->\ (}T_1\ \ldots\ T_n\texttt{)}\ I_{ret}\texttt{)\ \ ; success continuation} \\ \qquad \texttt{(->\ ()}\ I_{ret}\texttt{))\ \ \ ; failure continuation} \\ \qquad I_{ret}\texttt{))}, \\ \quad \textbf{if } I = I_{con}{\bowtie}^{\sim} \textbf{ and } DD = \texttt{(define-datatype (}I_{tc}\ I_{p_1}\ \ldots\ I_{p_m}\texttt{)} \\ \qquad\qquad \ldots\ \texttt{(}I_{con}\ T_1\ \ldots\ T_n\texttt{)}\ \ldots\texttt{)} \\ \\ A(I), \text{otherwise} \end{cases}$$

$A \oplus [DD_1, \ldots, DD_n]$ is an abbreviation for $((A \oplus DD_1) \oplus \ldots) \oplus DD_n$.

Figure 15.12: Notation for extending type environments with datatypes.

Type Erasure

$\lceil$`(define-datatype (`$I_{tc}\ I_{p_1}\ \ldots\ I_{p_k}$`)`
$\quad$`(`$I_{cnstr_1}\ T_{1,1}\ \ldots T_{1,m_1}$`) ... (`$I_{cnstr_n}\ T_{n,1}\ \ldots T_{n,m_n}$`))`$\rceil$
$=$`(define-data `I_{tc} `(`I_{cnstr_1} `x`$_1$ `... x`$_{m_1}$`) ... (`I_{cnstr_n} `x`$_1$ `... x`$_{m_n}$`))`

$\lceil$`(module `$DD_1\ \ldots\ DD_k$ `(define `$I_1\ [T_1]\ E_1$`) ...(define `$I_n\ [T_n]\ E_n$`))`$\rceil$
$=$ `(let ((`$I_{dd_1}\ E_{dd_1}$`) ... (`$I_{dd_m}\ E_{dd_m}$`))`
`(recordrec (`$I_1\ \lceil E_1 \rceil$`) ...(`$I_n\ \lceil E_n \rceil$`)))`
where $\mathcal{D}_{\text{def}}[\![\lceil DD_1 \rceil]\!] \ @\ \cdots\ @\ \mathcal{D}_{\text{def}}[\![\lceil DD_k \rceil]\!]$
$= [$`(define `$I_{dd_1}\ E_{dd_1}$`)...(define `$I_{dd_m}\ E_{dd_m}$`)`$]$
using the $\mathcal{D}_{\text{def}}$ function from Figure 10.24.

$\lceil$`(open `$E_{mod}\ E_{body}$`)`$\rceil$ $=$ `(with-fields (`$I_1\ \ldots\ I_n$`) `$\lceil E_{mod} \rceil\ \lceil E_{body} \rceil$`)`
where E_{mod} : `(moduleof (`AB^*`) (`$I_1\ T_1$`) ... (`$I_n\ T_n$`))`

Operational Semantics

`(load `S`)` $\Rightarrow$ `contents[`S`]` `[load]`

Figure 15.13: Dynamic semantics for the module system of FL/RM.

```
leaf : (generic (t) (-> () (treeof t)))
leaf~ : (generic (r t)
          (-> ((treeof t) (-> () r) (-> () r))
              r)

node : (generic (t) (-> (t (treeof t) (treeof t)) (treeof t)))
node~ : (generic (r t)
          (-> ((treeof t)
               (-> (t (treeof t) (treeof t)) r)
               (-> () r))
              r)
```

Figure 15.14: The types of the constructor and deconstructor procedures introduced by the `treeof` datatype declaration.

a type environment with these types.

Although deconstructors may be used explicitly in programs, they are usually used implicitly via the `match` construct introduced in Section 10.5. In the module language, the `match` construct can be desugared exactly as in Figure 10.30, except that $\texttt{equal}_L$ must be an equality operation appropriate for the type of the literal L rather than the generic `equal?`. For instance, $\texttt{equal}_{17}$ is `=`, $\texttt{equal}_{true}$ is `bool=?`, and $\texttt{equal}_{'foo}$ is `sym=?`.

For simplicity, unparameterized datatypes are required to be written as applications of nullary type constructors. For instance, a geometric shape type could be declared as

```
(define-datatype (shape)
  (square int)
  (rectangle int int)
  (triangle int int int))
```

in which case the figure type would be `(shape)` (the application of a nullary type constructor) and not `shape` (which is a nullary type constructor, not a type). It is left as an exercise to extend the language to support declarations of types in addition to type constructors.

15.5.2.3 Example: A Parameterized Module

As a non-trivial example of a module, consider the parameterized table module expression $E_{makeTableModule}$ in Figure 15.15, which implements an immutable but updatable table as a linked list of key/value pairs. The module declares a `tableof` type constructor parameterized over the value type `t`. Additionally, we

want to abstract our table implementation over the type of the key, which must admit equality testing via a `key=?` procedure. We achieve this simply by using `lambda` to parameterize the table module over a key module `key-mod`. Since the type of `key-mod` cannot be inferred, it is explicitly provided. When we call this procedure on a particular key module, we get back a table module that is polymorphic in the type of value stored in the table.

Most languages would require that we write the table module in terms of a key module that must be supplied at either compile time or link time. There would be a separate language for specifying the relationship between the table and key modules. For example, in SML we would use a **functor** (a linking language function) to abstract the table module over the key module. But because FL/RM has first class modules, the relationship can be expressed via ordinary procedural abstraction.

15.5.2.4 Semantics of `module` and `open`

The static and dynamic semantics of the module constructs is defined in Figures 15.11–15.13. From the type erasure for `module` in Figure 15.13, we see that at run time a module is just a record of values denoted by recursively scoped definition expressions that are evaluated in the scope of the constructor and deconstructor procedures introduced by the `define-datatype` declarations. This scoping information is also apparent in the [*module*] type rule, where the value bindings in the module are analyzed with respect to a type environment that not only includes the types of all constructor and deconstructor procedures declared in the `define-datatype` declarations (via $\oplus$) but also includes the types of the defined expressions. In addition to types for the value bindings, the `moduleof` type for a `module` expression includes abstract types of the form `(`I_{tc} I_{p_1} ... I_{p_k}`)` declared by the `define-datatype` declarations. Only the abstract type constructor name I_{tc} is a binding occurrence; the type parameters I_{p_1} ... I_{p_k} are provided only to indicate arity information (the number of type arguments for the type constructor).

As indicated by the definition of $\oplus$ in Figure 15.12, constructors create values with the abstract type and deconstructors decompose values with the abstract type. Thus, they play the role of `up` and `down` in the typing rule for $\texttt{pack}_{depend}$ in Figure 15.9 (see Exercise 15.9). As indicated by the [*module*] type rule and the `module` type erasure rule, the constructor and deconstructor procedures are *not* exported by a module unless the programmer includes explicit value definitions for them. Thus, the programmer has complete control over how abstract values are constructed and deconstructed. If the constructors for an abstract type are not exported, clients cannot create forgeries that possibly violate representation

```
(lambda ((key-mod (moduleof ((key))
                    (key=? (-> ((key) (key)) bool)))))
  (open key-mod
    (module
      (define-datatype (tableof t)
        (empty)
        (non-empty (pairof (pairof (key) t)
                           (tableof t))))
      (define make-table empty)
      (define lookup
        (lambda (k tbl succ fail)
          (match tbl
            ((empty) (fail))
            ((non-empty (pair (pair ak x)) rest)
             (if (key=? k ak)
                 (succ x)
                 (lookup rest k succ fail))))))
      (define insert
        (lambda (newk newval tbl)
          (lookup tbl k
            (lambda (x) (error alreadyInTable))
            (lambda () (non-empty (pair newk newval) tbl)))))
      (define delete
        (lambda (k tbl)
          (match tbl
            ((empty) tbl)
            ((non-empty (pair (pair ak x)) rest)
             (if (key=? k ak)
                 rest
                 (non-empty (pair (pair ak x)
                                  (delete rest k))))))))
       )))
```

Figure 15.15: An expression $E_{makeTableModule}$ denoting a procedure that takes a key module and returns a table module.

invariants. If the deconstructor for an abstract type is not exported, then clients cannot directly manipulate the concrete representation of an abstract value.

For example, in the table module in Figure 15.15, the `empty` constructor is exported by the module under a different name (`make-table`), but the `non-empty` constructor and the `empty~` and `non-empty~` deconstructors are *not* exported. These are used only to define the `lookup`, `insert`, `delete` operations. So it is impossible for a client to create a non-empty table or manipulate the bindings of a table except through these operations.

For simplicity, we consider two `moduleof` types to be equal when their abstract types are exactly the same (including order and parameter names) and their bindings have equal types in the same order. This is overly restrictive. The reader is encouraged to develop a more lenient type equality rule as well as subtyping rules for `moduleof`.

The type erasure rule for `(open` E_{mod} E_{body}`)` indicates that it dynamically makes the value bindings of the module denoted by E_{mod} available in the body expression E_{body}. Note that the type erasure rule needs to "know" the type of E_{mod} in order to determine the field names needed by `with-fields`. The [*open*] typing rule is similar to the [*dunpack*] rule for dependent packages in that it substitutes a dependent type for all occurrences of abstract type constructors in the body expression. As in [*dunpack*], the expression on which a dependent type depends must be pure. The easiest way to guarantee this is to require E_{mod} in `(dselect` I_{tc} E_{mod}`)` to be a syntactic value. However, we will see shortly that this solution has fundamental drawbacks in the presence of parameterized modules.

An example of the result of applying the [*module*] and [*open*] type rules is the type $T_{makeTableModule}$ (Figure 15.16) of the expression $E_{makeTableModule}$ studied earlier. Each of the procedures exported by the module has a type schema that parameterizes over unification variables introduced by type reconstruction. Note how all instances of the `(key)` type have been replaced by `((dselect key key-mod))` due to the substitution in the [*open*] rule.

```
(-> ((key-mod (moduleof ((key))
                (key=? (-> ((key) (key)) bool)))))
  (moduleof ((tableof t))
    (make-table (generic (a) (-> () (tableof a))))
    (lookup (generic (b c)
              (-> (((dselect key key-mod)) (tableof b)
                   (-> (b) c) (-> () c))
                  c)))
    (insert (generic (d)
              (-> (((dselect key key-mod)) d (tableof d))
                  (tableof d))))
    (delete (generic (e)
              (-> (((dselect key key-mod)) (tableof e))
                  (tableof e))))
    ))
```

Figure 15.16: The type $T_{makeTableModule}$ of the expression $E_{makeTableModule}$.

15.5.2.5 Loading Modules

The `load` construct supports the development and construction of large programs by allowing separately developed program modules to refer to one another. In our simple system, `(load` S`)` causes the desugared expression named by the unique name S to replace `(load` S`)`. The loaded expression is called `contents[`S`]` in our rules.

Because module dependencies must be acyclic, it is not possible to have modules that directly load each other. Nevertheless, modules with recursive dependencies can be parameterized over their dependencies and expressed within FL/RM; see Exercise 15.26.

Unifying the programming and linking languages via `load` and first-class modules is very powerful. As illustrated above, the creation, instantiation, and linking of parameterized modules is easily accomplished via `lambda` and application. It is also possible to choose which modules to load at run time using `if`, as in the following procedure:

```
(lambda (matrix)
  (open (if (sparse? matrix)
            (load "sparse-matrix-module-v3.22.cmp")
            (load "dense-matrix-module.cmp-v4.5"))
    ... code that manipulates matrix ...))
```

The ability to use arbitrary computation when linking program components permits idioms that are not expressible in most linking languages. Some down-

sides are that linking is not a separate phase from computation and it may not terminate.

The simple module facility described here still has many important shortcomings.

- Module types can quickly become very awkward to write. Often, we don't need all the functionality of a module, only particular features. The table module above can use any type with equality as a key. Subtyping can supply the necessary machinery, but it is still useful for languages to provide special syntax for specifying that a type parameter must support certain operations. CLU's `where` mechanism was designed to solve this problem.

- Explicitly abstracting over modules is a tedious operation, and results in overly complex code when, for example, two modules must share a common definition. SML's sharing specification was designed to address this issue.

- Having programmers essentially encode all version information in a manifest string constant is very inconvenient. It is possible to have the programmer specify just a name, like `"make-table-module.cmp"`, have the compiler use the most recent version of the file, and have the compiler and runtime system ensure that the code available during type checking is in fact the source for the object code loaded at run time. FX used the desugaring process to introduce a unique stamp from the file system for this purpose. Whenever a module is modified, any other modules that `load` that module name must be recompiled. Exercise **??** explores more sophisticated approaches to the value store.

- In the presence of parameterized modules, there is a fundamental problem with using the crude syntactic value test to conservatively approximate which module expressions do not have side effects. To see this, suppose that we replace the body of the `open` subexpression in Figure **??** by just the `insert` application. In this case, the type of both the `insert` and `open` subexpressions would be `((dselect tableof tbl-mod) bool)`, but the inner `let` would have type

  ```
  ((dselect tableof (mk-tbl-mod int-key-mod)) bool)
  ```

  and the outer let would have a type like

  ```
  ((dselect tableof ((load "make-table-module-v1.3.cmp")
                     (load "int-key-module-v2.0.cmp")))
   bool)
  ```

The problem with the last two types is that both module expressions in the `dselect`s are applications and therefore not syntactic values (*expansive* in the literature). Even though both expressions are in fact referentially transparent, the conservative syntactic test of non-expansiveness fails and causes type checking to fail. This sort of failure will occur whenever an attempt is made to export a type containing an abstract type from a parameterized module outside the scope of an application of that module.

Thus, the syntactic value test for expression purity imposes a kind of export restriction on abstract values, thereby reducing the power of the module system. This problem can be mitigated somewhat by using `let` to introduce names for applications, as in the `let` binding of `tbl-mod`, without which even the type of the `insert` expression would contain the application `(mk-tbl-mod int-key-mod)`. But `let` only locally increases the scope in which subexpressions are well-typed and cannot remove the effective export restriction. What we really need is a better way to determine the purity of an expression, which is the subject of the next chapter.

▷ **Exercise 15.19** In the table implementation in Figure 15.15, the `lookup` procedure takes success and failure continuations, and is polymorphic in the return type of the continuations. Alternatively, `lookup` could be modified to return either the value stored under the key or some entity indicating the value was not found. Define a new datatype to express this return type, and modify the table implementation to use it. ◁

▷ **Exercise 15.20** Sam Antix notices that the `load` syntax requires the value's name to be a manifest constant. He suggests that `load` should be a primitive procedure that takes a string argument, i.e., one could apply `load` to any expression that returns a string. Is this a good idea? Why or why not? ◁

▷ **Exercise 15.21** The abstract type names (and their parameters) introduced by `define-datatype` and used in `moduleof` types are binding occurrences. Extend the definition of FTV and type substitution to properly handle these type names. ◁

▷ **Exercise 15.22** Is the following FL/RM expression well-typed? If so, give the type reconstructed for `test` and the type of the whole expression. If not, explain.

```
(let ((mk-tbl-mod (load "make-table-module-v1.3.cmp")))
      (int-key-mod (load "int-key-module-v2.0.cmp"))
  (let ((tbl-mod (mk-tbl-mod int-key-mod)))
    (open tbl-mod
      (let ((test (lambda (k v)
                    (lookup k
                            (insert k v (make-table))
                            (lambda (x) x)
                            (error shouldntHappen)))))
        (pair (test 1 true) (test 2 3))))))
```
◁

▷ **Exercise 15.23** It would be convenient if the module system were extended to support the declaration of types in addition to type constructors. For example, after the extension, an alternative way to define the geometric shape type discussed in the text would be

```
(define-datatype shape
  (square int)
  (rectangle int int)
  (triangle int int int))
```

and `(square 3)` would have the type `shape`.

a. Extend the type syntax, typing rules, and the definition of $\oplus$ so that `define-datatype` can declare types in addition to type constructors.

b. An alternative strategy is to transform all declarations and uses of user-defined types to declarations and uses of nullary type constructors. Define a program transformation that implements this strategy. ◁

▷ **Exercise 15.24** Write a type equality rule for `moduleof` type expressions that (1) permits the type components to be in any order; (2) permits the value components to be in any order; and (3) ignores the names (but not the number!) of the type parameters for each type constructor. ◁

▷ **Exercise 15.25** The module system described above uses static dependent types. Write [*proc*], [*let*], [*letrec*], and [*apply*] typing rules for this language, being careful to carry out all necessary substitutions. You may want to refer to Figure 15.9 and Exercise 15.16. ◁

▷ **Exercise 15.26** Consider the following three expressions:

E_A =
```
(module
  (f (lambda (x)
       (if (= x 0)
           0
           ((open (load "B.cmp") g) (- x 1))))))
```

E_B =
```
(open (load "A.cmp")
  (module
    (g (lambda (x)
         (if (= x 0)
             1
             (+ ((open (load "A.cmp") f) (- x 1))
                (g (- x 1))))))))
```

E_C =
```
(let ((amod (load "A.cmp"))
      (bmod (load "B.cmp")))
  (+ (open amod (f 3))
     (open bmod (g 3))))
```

a. Suppose that we want to compile E_A to the file `"A.cmp"`, E_B to the file `"B.cmp"`, and E_C to the file `"C.cmp"`. Explain why there is no compilation order that can be chosen that will allow us to eventually execute the code in E_C that will use E_A for `"A.cmp"` and E_B for `"B.cmp"`. Consider the case where the file system contains pre-existing files named `"A.cmp"` and `"B.cmp"`.

b. It is possible to change E_A and E_B into parameterized modules that do not directly load modules from particular files, but instead load modules from a parameter that is a (thunk of) a module. Based on this idea, rewrite E_A, E_B, and E_C in such a way that all three files can be compiled and executing E_C will return the desired result. ◁

▷ **Exercise 15.27** Modify the type reconstruction algorithm from Chapter **??** to handle the `module`, `open`, `load`, `lambda`, and `letrec` constructs. ◁

Reading

- CLU[L+79]
- SML and revised[AM87, MTH90, MTHM97]
- Mesa[MMS78]
- Benjamin Pierce's book[Pie02]
- John Mitchell's books[Mit96, Mit03]

- [CW85] discusses polymorphism, bounded quantification, existential types, compares existential types to data abstraction in Ada.
- Mitchell and Plotkin's existential types [MP84]. Impredicative Strong Existential Equivalent to Type:Type[HH86].
- MacQueen on Modules[Mac84, Mac88]. Dependent types to express modular structure[Mac86]
- Harper on modules[Har86, HMM90]
- [CHD01][CHP99]
- [Ler95]

[Ada] [Parnas?] [ML sharing]

For more information on existential types, see John Mitchell's textbook, [Mit96]. Luca Cardelli's QUEST language [Car89] employed first-class existential types.

Static dependent types are due to Mark Sheldon and David Gifford [SG90].

For a somewhat different of view in which a type *is* its operation set, see the programming language RUSSELL [BDD80].

The programming language Pebble [BL84] included strong existential types (also known as strong sums) and dependent types that could contain any value. Type checking in Pebble could fail to terminate if values in dependent types looped.

Putting type declarations into a language with type reconstruction, as we did with our final module system design, can lead to some surprising results. For example, it is easy to make type checking undecidable. To see how inferable and non-inferable types can be combined in a decidable type system, see James O'Toole's work in [OG89].

[Recent work on dependent types: Cayenne, Hongwei's work.]

Chapter 16

Effects Describe Program Behavior

Nothing exists from whose nature some effect does not follow.

— Ethics, I, proposition 36, Benedict Spinoza

16.1 Types, Effects, and Regions - What, How, and Where

We have seen that types are a powerful tool for reasoning about the ideas presented in the FL, Naming, and Data Chapters (Chapter **??**), yet types do not help us reason in detail about the ideas introduced in the State, Control, and Concurrency Chapters (Chapter **??**). A formal system called an **effect system** allows us to reason about many of the state, control, and storage issues that arise in practical programs.

In this chapter, we introduce effect systems and explore their applications. An effect system produces a concise description of the observable actions of an expression, and this description is called the **effect** of the expression. Example effects include writing into a region of the store or jumping to a non-local label. An effect is a **dual** to a type. Just as a type describes **what** an expression computes, an effect describes **how** an expression computes.

As we shall see, effects describe a wide variety of properties about a program that are usefull to programmers, compiler writers, and language designers. Effect systems provide three benefits to the programmer: improved documentation, safety, and execution efficieny. Documentation and saftey improvements

include the ability to better understand the behavior of code, including the ability to detemine how modules developed by others may modify state, modify files, perform non-local tranfers, or be the target of non-local jumps. Efficiency improvements include parallel expression scheduling, remote procedure call scheduling, and storage management. For example, when expressions do not share store dependencies, they can be reordered or executed in parallel, subject to their input values being available.

To make effects precise we introduce the idea of **regions** that describe **where** objects reside. In our effect system every object resides in a single region, and this region is described by the type of the object. We can think of regions as colors (red, blue, green, etc.) or as distinct memory banks (Bank 1, Bank 2, Bank 3, etc.), or even machines (mit.edu, cmu.edu, etc.). However, regions are logical locations, and may or may not correspond to physical locations in a given implementation. For example, control points can be assigned to regions that represent locations in code as opposed to regions in a store.

When two objects are in distinct regions mutations to one of the objects will not cause changes to the other object. This is a consequence of our invariant that an object is only in a single region. Thus regions can be used to prove that object references do not alias one another. *Aliasing* occurs when two references refer to the same object. Aliasing can inhibit important compiler optimizations such as caching the values of mutable objects in registers.

To produce an accurate accounting of effects we include three key innovations in our type system. First, the type of every mutable object includes the object's region. Second, we will account for the effect of a procedure in the type of the procedure as a **latent effect** that is realized when the procedure is called. Latent effects communicate the effects of a procedure from the point of the procedure's definition to its points of use. Third, we introduce the idea of effect and region polymorphism to permit procedures to have effects that depend on their input parameters.

Although in this chapter we discuss an interwoven system for effects in regions, it is possible to have effects without regions and regions without effects. In the absence of regions our effect system would be coarse, and would simply report a limited repitore of broad effects. In the absence of an effect system, a region system alone can not deduce when a particular region is accessed or when it becomes inaccessible. Although decoupling effects and regions is possible, we will show that there is no advantage to doing so because we can hide the complexity of a simulatenous effect and region system from programmers.

Ultimately, programmers must find an effect system easy to use and it must produce valuable results. In this chapter we will make an effect system easy to use by making it invisible to programmers. We will make it invisible by

performing effect and region reconstruction without programmer declarations or assistance. Early experiments with effect systems showed that programmers had a difficult time composing appropriate effect declarations, and thus the existance of a sound effect reconstruction algorithm is necessary for the practical application of a novel effect system you might think of creating!

Since the types of procedures include effects, effect reconstruction naturally depends upon type reconstruction and vice-versa. We will use FL/R as our base langauge in this chapter, and demonstrate how to reconstruct effects fully automatically in this language context.

In this chapter we introduce two classes of example effects. The first class, store effects (read, write, and create), describe the creation or observation of store based state. The second class, control effects (comefrom, goto), describe the creation of control points (labels) and the transfer of control to control points (gotos). As described above, both store and control effects are subscripted by a region that delinates the scope of the effect. A store region describes a set of cells (usually one), and a control region describes a set of control points (usually one). We will use store and control effects for concreteness, but new effects are readily introduced in the effect system framework, and abstract effects that encapsulate base effects are also possible.

We will introduce store effects with a few short examples, and then provide a complete set of effect system rules. We begin with the standard operations on cells:

E ::=	`...` | `(cell` *E*`)`	[Allocate and initialize a cell]
	| `(:=` *E* *E*`)`	[Cell set]
	| `(^` *E*`)`	[Cell read]

The `:=` and `^` procedures are respectively implemented with the `cell-ref` and `cell-set!` primitives that we previously defined in Chapter 8.

Our effect system will produce a summary of how we use these procedures in an expression. The simple effect system we discuss here does not keep track of the ordering or number of times a particular effect is used. However, we will keep track of what region of the store is subject to an effect. We will use "`!`" as the "has effect" relation for expressions as a complement to the "`:`" relation for "has type."

First, we create a mutable cell that contains an integer. The expression that creates the cell is assigned an `init` (initialize) effect in a new store region named `?r-1`:

```
(cell 1) : (cellof int ?r-1) ! (init ?r-1)
```

Next, we create a boolean cell, set it to true, and read out the contents of the cell. Note that the effects on this boolean cell are in a new store region called

?r-2. Whenever possible, we will use a new region for each object we create:

```
(let ((x (cell #f)))
  (begin
    (:= x #t)
    (^ x))) : bool ! (maxeff (init ?r-2) (write ?r-2) (read ?r-2))
```

Higher order procedures, such as the apply-twice procedure below, can be polymorphic both in type and effect. In this example, the application of apply-twice has the store effects (read ?r-3) and (write ?r-3):

```
(let ((apply-twice (lambda (f x) (f (f x))))
      (add-one (lambda (c)
                 (begin (:= c (+ (^ c) 1))
                        c)))
      (counter (cell 0)))
  (begin (apply-twice add-one counter)
         (^ counter)))))  :  int
                          ! (maxeff (init ?r-3)
                                    (write ?r-3)
                                    (read ?r-3))
```

The type schema of apply-twice in this example is

```
apply-twice : (generic (tf ft)   ; tf is input and output type of f
                                 ; ft is latent effect of f
                 (-> ((-> (tf) ft tf)  ; f
                      tf)              ; x
                     ft          ; latent effect of apply-twice
                     tf))        ; result
```

Note in this instance that effect polymorphism carries the effect of the procedure provided to apply-twice to apply-twice itself.

Procedure types in standard environment now have latent effects. Here are the entries in the standard type envirnoment for the free variables in the above example:

```
+    : (-> (int int) pure int)
cell : (generic (t r) (-> (t) (init r) (cellof t r)))
:=   : (generic (t r) (-> ((cellof t r) t) (write r) unit))
^    : (generic (t r) (-> ((cellof t r)) (read r) t))
```

When we generalize over a region in a type schema we are indicating that any region can be assigned. For example, every time that cell is used we assign a new region variable to the newly created cell. Thus we try to maximize the number of distinct regions used in a program to provide a fine grained accounting

of storage (or other assets). However, keep in mind that we assign a single region for each static occurance of `cell`, and all of the dynamically created cells from a single static occurance of `cell` will wind up in the same region. In addition, when cells are used interchangably, their types will be unified, forcing them to be in the same region.

Of course, many expressions will not have any effects:

```
(+ 1 2) : int ! pure
```

When an expression has no effects we say that the expression is `pure`, and conversely when it has effects we call it **impure**. The `pure` effect is a shorthand for the effect `(maxeff)`. A pure expression is guaranteed to be **referentially transparent**. An expression is referentially transparent when different syntactic occurrences of the expression are guaranteed to have the same value assuming identical bindings for all of the free variables in the expression. We have already discussed the idea of referential transparency in the chapter on state (Chapter 8). Programming languages do not guarantee referential transparency when expressions observe mutable state with expressions that have effects. Thus when an expression is pure, it will be referentially transparent becuase it can not observe mutable state.

Advanced properties of effect systems are beyond the scope of this book, including effect algebras that can associate execution times or storage costs with expressions. These advanced effect algebras require different approaches to type and effect reconstruction than the one we discuss below. The interested reader can consult the bibliography at the end of this chapter for reserach papers on these topics.

In the rest of this chapter, we introduce rules for assigning effects to expressions (Section 16.2), we will discuss how effects can be used to analyze program behavior (Section 16.3), and how effects can be reconstructed as an integral part of a type and effect system (Section 16.4).

16.2 An Effect System for FL/R

Formally, an *effect system* is a set of rules for assigning an effect and a type to an expression. Our effect system needs to assign types to expressions to permit us to analyze the behavior of user defined procedures. Thus, we first extend the syntax of procedure types to include latent effects:

`(->` `(`T*`)` F T`)`

where the T* are the types of the procedure's parameters, F describes the procedure's latent effect, and T is the output type of the procedure.

Figure 16.1 shows the grammar for FL/R language with types that include latent effects. An effect system inherits the names of effects from the latent effects of the primitive operators and procedures that are available in the standard top-level environment. We combine effects with the effect union operator `maxeff` that is associative (A), commutative (C), unitary (U), and has an identity (I) (`pure` is the identity element). Thus, effect combination is an ACUI algebra.

Our algebra of effects is important to consider because the equality of latent effects that occur in procedure types must be considered with respect to our ACUI effect algebra. Because effect combination is commutative, the order in which effects occur is not preserved when effects are combined. Thus procedures that perform equivent operations in different orders will have the same effect according to our algebra of effects.

E ::= L | `(if` E E E`)` | `(primop` O E*`)` | `(let ((`I E`)*)` E`)`
| `(letrec ((`I E`)*)` E`)` | `(lambda (`I*`)` E`)` | `(`E E*`)`

T ::= I | `(`I T`)` | `(->` `(`T*`)` F T`)` | `(cellof` T R`)`

F ::= `pure` | `(maxeff` F*`)` | `(`I R`)`

R ::= I

TS ::= `(generic (`I*`)` T`)`

Figure 16.1: Grammar for FL/R with latent effects.

We will now introduce a set of rules for assigning types and effects to FL/R expressions. The rules show us how to deduce the type T and effect F of an expression E given a type environment A:

$$A \vdash E \ : \ T \,!\, F$$

The standard environment A includes a library of standard procedures (such as `^`, `:=`, etc.), and the types of these procedures include latent effects that describe their actions.

The effect system shown in Figure 16.2 consists of rules that simultaneously compute the type and effect of an expression. FL/R's [*lambda*] and application [*app*] typing rules are extended to permit latent effects to move in and out of procedure types. The [*lambda*] rule moves the effect of a procedure's body into the procedure type of the `lambda` expression, and the [*app*] rule moves the

latent effect of a procedure type into the effect of a procedure application. Latent effects and these rules are the mechanism that communicates effect information from the point of procedure definition to the point of procedure call. Other rules such as [*if*] simply combine the effects of all subexpressions of E to compute a conservative approximation of the effects of E.

When our typing rules require that two types T_1 and T_2 be equal, any latent effects that are in identical positions in T_1 and T_2 must be equivalent according to our effect algebra. This occurs when T_1 and T_2 are procedure types since procedure types include latent effects. Recall that effect equivalence is considered with respect to the ACUI algebra for effects, and thus the order of effects does not matter. For example the effects `(maxeff (read ?r-1) (write ?r-2))` and `(maxeff (write ?r-2) (read ?r-1))` are identical.

However, even with our algebra of equivalence over effects, it is a simple matter to construct a program that does not "effect check." We say a program does not effect check when two effects are compared during type checking and the effects do not match. For example

```
(if #t ^ (lambda (c) 1))
```

is not well typed in a strict sense since the cell reference operator `^` and the `lambda` must have identical types but their types do not contain identical latent effects. Note that we do not insist that the consequent and alternative of an `if` have the same effect. Only effects in the types of the consequent and alternative must be the same, and this will only occur when `if` is returning a type that contains a procedure type.

We can make the latent effects in two procedure types equivalent by permitting expressions to take on more effects than they may actually cause to ensure that programs always effect check. Thus our effect system for FL/R includes a subeffecting rule called [*does*] that permits effect expansion. For example, the [*does*] rule permits our example

```
(if #t ^ (lambda (c) 1))
 : (-> ((cellof int ?r-1)) (read ?r-1) int)  ! pure
```

to be well typed by expanding the latent effect of `(lambda (c) 1)` to be `(read ?r-1)` to match the latent effect of the cell reference operator `^`.

Henceforth when we refer to the **effect of an expression**, we will mean the smallest effect that can be proven by our rules. This is because [*does*] permits an expression to take on many possible effects. Effects form a lattice under `maxeff` and thus the notion of a smallest effect is well defined. Later in this chapter when we discuss effect reconstruction (Section 16.4), we will show how to compute the smallest effect allowed by the rules.

Our typing rules for `let` have two forms, [*impure-let*] and [*pure-let*]. As we discussed in our introduction to FL/R, only `let` bindings that do not have side effects can be generalized. In the literature, such let expressions are called "non-expansive." For simplicity we will assume that any expression that is not a `lambda` that includes an application is expansive. It would seem logical to use our own effect system to determine which `let` expressions are pure and thus can be generalized. We leave this extension to the interested reader.

In our typing rules we use FV to denote a function that returns the free type, effect, or region variables in a description expression (type or effect), and FDV to denote a function that returns all of the free type, effect, or region variables in a type environment.

16.3 Using Effects to Analyze Program Behavior

Our exploration of the application of effects will consider how effects can be made to disappear with effect masking, how effects can be used to describe the actions of applets, how effects can be used to describe control transfers, and how static storage allocation can use effects.

16.3.1 Effect Masking Hides Invisible Effects

Effect masking is an important tool for encapsulation. It allows effects to be erased from an expression when the effects cannot be observed from outside of the expression. For example, let's reconsider the effect of the expression we introduced above:

```
(let ((apply-twice (lambda (f x) (f (f x))))
      (add-one (lambda (c) (begin (:= c (+ (^ c) 1)) c)))
      (counter (cell 0)))
  (begin
    (apply-twice add-one counter)
    (^ counter)))))
        : int ! (maxeff (init ?r-3) (write ?r-3) (read ?r-3))
```

Since region `?r-3` is not in the type of this expression and is not in the types of the free variables of this expression (`:=, +, ∧, cell`), we know that region `?r-3` is invisible outside of the expression. It is impossible for any context for this expression to determine if the expression has performed any side effects to `?r-3`. Thus, effects on `?r-3` can be erased, leaving this expression with no effect.

$$A[I\!:\!T] \vdash I \;:\; T \;\; \texttt{!}\; \texttt{pure} \qquad [id]$$

$$[\ldots, I\!:\!(\texttt{generic}\ (I_1\ \ldots\ I_n)\ \ T_{body}), \ldots] \vdash I \;:\; ([T_i/I_i]_{i=1}^{n})\, T_{body} \;\texttt{!}\; \texttt{pure} \qquad [genvar]$$

$$\frac{A[I_1\!:\!T_1\ \ldots\ I_n\!:\!T_n] \vdash E \;:\; T_r \;\; \texttt{!}\; F}{A \vdash (\texttt{lambda}\ (I_1\ \ldots\ I_n)\ E) \;:\; (\texttt{->}\ (T_1\ \ldots\ T_n)\ F\ T_r) \;\; \texttt{!}\; \texttt{pure}} \qquad [lambda]$$

$$\frac{A \vdash E_o \;:\; (\texttt{->}\ (T_1 \ldots T_n)\ F_p\ T_r) \;\; \texttt{!}\; F_o \qquad \forall_{i=1}^{n} \,.\, A \vdash E_i \;:\; T_i \;\; \texttt{!}\; F_i}{A \vdash (E_o\ E_1 \ldots E_n) \;:\; T_r \;\; \texttt{!}\; (\texttt{maxeff}\ F_o\ F_1 \ldots F_n\ F_p)} \qquad [app]$$

$$\frac{A \vdash E_1 \;:\; \texttt{bool} \;\; \texttt{!}\; F_1 \qquad A \vdash E_2 \;:\; T \;\; \texttt{!}\; F_2 \qquad A \vdash E_3 \;:\; T \;\; \texttt{!}\; F_3}{A \vdash (\texttt{if}\ E_1\ E_2\ E_3) : \quad T \,!\, (\texttt{maxeff}\ F_1\ F_2\ F_3)} \qquad [if]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n} \,.\, A \vdash E_i \;:\; T_i \;\; \texttt{!}\; \texttt{pure} \\ A[I_1\!:\!\texttt{Gen}(T_1,\ A),\ \ldots\ I_n\!:\!\texttt{Gen}(T_n,\ A)] \vdash E_b \;:\; T_b \;\; \texttt{!}\; F_b\end{array}}{A \vdash (\texttt{let}\ ((I_1\ E_1)\ \ldots\ \ (I_n\ E_n))\ E_b) \;:\; T_b \;\; \texttt{!}\; F_b} \qquad [pure\text{-}let]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n} \,.\, A \vdash E_i \;:\; T_i \;\; \texttt{!}\; F_i \\ A[I_1\!:\!T_1,\ \ldots\ I_n\!:\!T_n] \vdash E_b \;:\; T_b \;\; \texttt{!}\; F_b\end{array}}{A \vdash (\texttt{let}\ ((I_1\ E_1)\ \ldots\ \ (I_n\ E_n))\ E_b) \;:\; T_b \;\; \texttt{!}\; F_b} \qquad [impure\text{-}let]$$

$$\frac{A \vdash E \;:\; T \;\; \texttt{!}\; F' \qquad F' \;\sqsubseteq\; F}{A \vdash E \;:\; T \;\; \texttt{!}\; F} \qquad [does]$$

$$\frac{\begin{array}{c}A_{standard} \vdash O \;:\; (\texttt{->}\ (T_1\ \ldots\ \ T_n)\ F\ T) \;\; \texttt{!}\; \texttt{pure} \\ \forall_{i=1}^{n} \,.\, A \vdash E_i \;:\; T_i \,!\, F_i\end{array}}{A \vdash (\texttt{primop}\ O\ E_1\ \ldots\ \ E_n) \;:\; T \;\; \texttt{!}\; (\texttt{maxeff}\ F_1\ \ldots\ \ F_n\ F)} \qquad [primop]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n} \,.\, A[I_1\!:\!T_1,\ \ldots\ I_n\!:\!T_n] \vdash E_i \;:\; T_i \;\; \texttt{!}\; \texttt{pure} \\ A[I_1\!:\!\texttt{Gen}(T_1,\ A),\ \ldots\ I_n\!:\!\texttt{Gen}(T_n,\ A)] : E_b\!:\ T_b \,!\, F_b\end{array}}{A \vdash (\texttt{letrec}\ ((I_1\ E_1)\ \ldots\ (I_n\ E_n))\ E_b) \;:\; T_b \;\; \texttt{!}\; F_b} \qquad [letrec]$$

$$Gen(T, A) = (\texttt{generic}\ (I_1\ \ldots\ I_n)\ T), \text{where } \{I_i\} = FV(T) - FDV(A)$$

Figure 16.2: FL/R Type and Effect Rules

The general rule for effect erasure is

$$\frac{\begin{array}{rcl} A & \vdash & E : T \,!\, F \\ F' & = & F - \{F_1 \ldots F_n\} \\ \textit{where for all } R_j & \in & FV(F_i),\ R_j \notin FV(T)\ [\textit{Export restriction}] \\ \textit{and for all } V_i & \in & \textit{FreeIds}[\![E]\!],\ R_j \notin FV(A[V_i])\ [\textit{Import Restriction}] \end{array}}{A \vdash E : T \,!\, F'}$$

The effect erasure rule will detect that certain expressions, while internally impure, are in fact externally pure, and thus are referentially transparent. Thus it permits impure expressions to be included in functional programs. Thus even in functional programs selected program expressions can take advantage of local side effects for efficiency without losing their referential transparency. It also allows effects that denote control transfers (as from `cwcc`) to be masked, indicating that an expression may perform internal control transfers that are not observable outside of the expression. (See Section 16.3.3 for more on control effects.)

16.3.2 Effects Describe the Actions of Applets

One application of effects is to provide applet security by labeling trusted primitive operations with latent effects that describe their actions. For example, all procedures that write on the executing computer's disk could carry a `write-disk` latent effect. Other latent effects could be assigned to display and networking procedures. These effects create a verifiable, succinct summary of the actions of an imported applet. The effects of an applet could be presented to a security checker — such as a user dialog box — that would accept or reject applets on the basis of their effects. In such a system, the vocabulary of effects is defined by the client machine and its effect system, and not by the imported applet. Thus the client security system is able to verify the potential actions of an applet in client defined terms.

An essential part of using types and effects for mobile code security is the ability of the recepient of code to rapidly verify the code's purported type and effect. This is because the type and effect of an applet effectively document its output and observable behavior, and the well-typedness of an applet guarantees that the applet will not perform illegal run-time operations. In a proof carrying code framework, a producer of code provides a series of assetions about code that can be rapidly verified by a code consumer. In proof carrying code, these assertions document saftey properties of the code because the underlying language (e.g. assembly language) may be inherently unsafe. In our framework we too can provide assertions with exported code. When an applet is provided to

a consumer we can include a parse tree of the applet with explicit types and effects attached to each expression. With such assertions for every expression it is a simple matter to run our type and effect rules in linear time over the code to verify the purported type and effect of the provided applet.

16.3.3 Effects Describe Control Transfers

Effects can be used to analyze non-local control transfers such as the behavior produced by call-with-current-continuation (`cwcc`). `cwcc` is a procedure that creates a local control point, continues execution, but permits the executed code to invoke the created control point and exit from the body of the `cwcc`. For example, consider the following FL/R example:

```
(lambda (x y)
  (+ 1 (cwcc (lambda (exit)
               (if (= y 0) (exit 0) x)))))
```

In this example, if `y` is 0 the outer `lambda` will return 1, and if `y` is not 0 the outer `lambda` will return `x+1`.

`cwcc` can be understood by consdering the procedure P that `cwcc` takes as its only input. P receives as its single parameter a continuation procedure C that, when called with value V, will cause the computation to return from `cwcc` with V as its value (see Section 9.4 for more). If C is never called, the value returned by P is returned by `cwcc`. Whew! Now read that one more time following along with the example above.

The type schema of `cwcc` is rather complicated:

```
(generic (t r t2 f)
  (-> ((-> ((-> (t) (goto r) t2)) f  t))
      (maxeff f (comefrom r))
      t))
```

This type schema shows that `cwcc` takes a procedure P with type

```
(-> ((-> (t) (goto r) t2)) f t)
```

that has a latent effect `f` and returns a value of type `t`. Procedure P will receive the current continuation, C, as an argument. C has type `(-> (t) (goto r) t2)`. `cwcc` will return with a value when the value is either provided as an input to C or is the return value from P. Since either of these choices will cause `cwcc` to return with the provided value, both of these options must insist upon the same type for the value as can be seen in the above type schema.

`cwcc` creates a `comefrom` effect to indicate that control may be transfered

back to the return point of `cwcc`. If the continuation C is called after `cwcc` returns, the `cwcc` will return again! Thus the continuation procedure C has a latent `goto` effect that is specific to the continuation's region. Other effects of P (represented by the variable `f`) are assigned to `cwcc`. `(goto r)` will only show up in `f` if P actually invokes C. The type `t2` is generic to allow C to be called in any context because C never returns.

Returning to our example

```
(lambda (x y)
  (+ 1 (cwcc (lambda (exit)
               (if (= y 0) (exit 0) x)))))
```

the expression `(exit 0)` will have a `(goto ?r-4)` effect (where `?r-4` is a new region), and the call to `cwcc` will have the effect `(comefrom ?r-4)`. The `comefrom` effect is used because `cwcc` has established the `exit` control point that permits control to materialize at a later time at the return from the `cwcc`. The effect name `comefrom` is a play on the name `goto`. Finally, in this example, effect masking can be used to drop the `(goto ?r-4)` and `(comefrom ?r-4)` from the `cwcc` expression and thus the latent effect of the `lambda`.

As we have just seen in our small example, effect masking works for all effects including the control effects `comefrom` and `goto`. When a control effect in region R is masked from expression E, it means that any context for expression E will not be subject to unexpected control transfers with respect to the continuation in R. Effect masking of control effects is powerful because it allows module implementors to use control transfers internally, while allowing clients of the modules to insist that these internal control transfers do not alter the clients' control flow. A client can guarantee this invariant by ensuring that it does not call module procedures with control effects.

16.3.4 Effects Can Be Used to Deallocate Storage

In implementations of FL/R class languages, a cell is typically reclaimed by a garbage collector (see Chapter **??**) because it is difficult to statically determine when a cell can no longer be reached. With regions, it is possible to do limited forms of static allocation and deallocation of memory. Assuming we are considering deallocating a region R that occurs in the type of an expression, the rule

for storage deallocation is

$$\frac{\begin{array}{c} A \vdash E \ : \ T \,!\, F \\ R \notin FV(F) \\ R \notin FV(T) \\ \textit{for all } R_i \ \texttt{(comefrom } R_i\texttt{)} \notin F \\ \textit{for all } I_i \in \textit{FreeIds}[\![E]\!],\ R \notin FV(A[I_i])\ ,\ \textit{for all } R_i\ (\texttt{comefrom}\ R_i),\ \notin A[I_i] \end{array}}{\textit{Deallocate all storage allocated in } R \textit{ after } E}$$

We rely upon our effect system to make this rule sound. Consider the following expression:

```
(let ((my-cell (cell 47))
      (your-cell (cell 48)))
       (lambda () (^ my-cell))).
```

We cannot deallocate `my-cell` after this expression as the latent effect of the procedure returned will contain the region for `my-cell`. This latent effect prevents us from deallocating storage that can be accessed by a procedure. However, we are free to deallocate `your-cell` after this expression returns because it meets the test of our deallocation rule above and thus will no longer be accessible.

Effects can also be used to manage the deallocation of `lambda` storage by associating a region with every procedure type. We leave the details as an exercise for the reader. Note that there are no allocation effects for `lambda` because FL/R does not provide comparison operators on procedures. Thus it is not possible in FL/R to distinguish procedure instances, and thus procedure allocation is not an obervable effect.

16.4 Reconstructing Types and Effects

Our treatment of type reconstruction in Chapter **??** introduced type schemas to permit an identifier to have different types in different contexts. For example, the identity function `(lambda (x) x)` can be used on any type of input. When it is `let` bound to an identifier, it has the type schema `(generic (t) (-> (t) t))`. The job of a type schema is to describe all of the possible types of an identifier by identifying type variables that can be generalized.

Effect and region reconstruction requires us to further elaborate a type schema with effect and region variables. These type schemas also carry along a set of constraints on the effects they describe. We call a type schema that includes a constraint set an **algebraic type schema** [JG91]. A **constraint set** (C) is a set of assertions (A) between effects

C ::= `(`A`*)`
A ::= `(>=` F_1 F_2`)`

where (`>=` F_1 F_2) means that effect F_1 contains all of the effects of effect F_2. The general form of an algebraic type schema is:

TS ::= `(generic (`I^*`)` T C`)`

For example, the algebraic type schema for a procedure that implements the cell assignment operation (`:=`) is:

```
(generic (t e r)
        (-> ((cellof t r)) e unit)     ; type
        ((>= e (write r)))             ; effect constraints
```

There are three parts to this algebraic type schema. The variables

```
(t e r)
```

describe the type, effect, and region variables that can be generalized in the type schema. The procedure type

```
(-> ((cellof t r)) e unit)
```

describes the cell assignment operation and notes that its application will have effect `e`. The constraint

```
((>= e (write r)))
```

describes the constraints upon the effect variable `e`. In this case, the assignment operation can have any effect as long as it is larger than `(write r)`.

An integer cell incrementing procedure would have the following algebraic type schema:

```
(generic  (e r)
  (-> ((cellof int r)) e unit)           ; type
  ((>= e (read r)) (>= e (write r))))    ; effect constraints
```

The constraint set in the above type schema constrains the latent effect of the increment procedure to include read and write effects for the region of the cell being incremented.

Type schemas that include effects and constraints are central to Algorithm Z, our algorithm for type and effect reconstruction (Figure **??**). Algorithm Z is similar to Algorithm R from Chapter **??**, except that it simultaneously computes the type and effect of an expression. The unification algorithm U is inherited unchanged from Algorithm R. A key intuitive insight into Algorithm Z is that constraints are used to keep track of the effects that expressions must have, and the constraints on an expression are solved after the type and effect of the expression is computed. Thus types and effects returned by Z can include effect variables that are subject to effect constraints returned by Z.

The type and effect reconstruction algorithm Z takes as input an expression E, a type environment A_C, and a starting substitution S. Algorithm Z outputs the type T of the expression, the effect F of the expression, the final substitution S', and the constraint set C on effect variables in T, F, and S'.

$$(Z[\![E]\!]\ A_C\ S) = \langle T, F, S', C\rangle$$

Note that in the type environment A_C type schemas include effect constraints as descrbed above.

An expression E is well typed if Z does not fail and if C is solvable. A constraint set C is solvable if there is a concrete assignment of effects M to the effect variables in C that satisfies all of the constraints in C. The substitution of concrete effects to effect variables that satisfies C is called a model. We write that substitution M is a model of C as $M \models C$. In our case, the [*does*] rule allows us to increase the effect of any expression and thus we will always be able to find a model for C.

Algorithm Z in Figures 16.4–16.5 is defined such that its results and a corresponding model M result in a provable type and effect

$$(M\ (S'A_C)) \vdash\ E : (M\ (S'\ T))\ !\ (M\ (S'\ \ F))$$

(S X) or (M X) means the result of respectively applying the substitution S or M to X, where X can be a type, effect, or a type environment. In the case of a type environment the substitution is applied to all of the identifiers bound in the environment.

An integral part of Algorithm Z is the *Zgen* algorithm for creating algebraic type schemas. The *Zgen* algorithm is identical to the *Rgen* algorithm from our type reconstruction algorithm R, with the key addition of detecting generic effect and region variables and accounting for them by carrying along a copy of a constraint set that can later be instantiated. When a type schema is instantiated, the constraint set carried in the type schema is updated to replace the generic effect and region variables and is returned from Z.

$$\texttt{Zgen}(T,\ A_C,\ S,\ C) = (\texttt{generic}\ (I_1 \ldots I_n)\ T\ C)$$
$$\{I_1 \ldots I_n\} = FV((S\ T)) + FV((S\ C)) - FDV((S\ A_C))$$

In our definition of `Zgen`, we have used FV to denote a function that returns the free type, effect or region variables in a type or constraint expression, and FDV to denote a function that returns all of the free type, effect, or region variables in a type environment .

An example helps to clarify the role of constraints kept in type schemas. Consider the type schema of the integer cell incrementing procedure `plus-one`

```
(generic (e r)
        (-> ((cellof int r)) e unit)
        ((>= e (read r)) (>= e (write r))))
```

and assume that `c` has type

```
c: (cellof int ?r-5)
```

Then

```
(plus-one c) : unit ! ?e-1
```

The constraints created by `(plus-one c)` will be

```
((>= ?e-1 (read ?r-5)) (>= ?e1 (write ?r-5)))
```

and thus an application of `plus-one` will have `read` and `write` effects. Note that the effect of the `plus-one` procedure is a variable. Our algorithm for creating procedure types always uses a unification variable to represent the effect of a procedure in a procedure type. The effects of the procedure's body are placed into the constraint set to bound the newly introduced variable. The advantage of this approach is that two procedure types always have compatible effect components because they can be unified together. The consequence of unifying the latent effects of two procedures is that both of the procedure bodies will have their effect bounds combined.

An expression E is well typed if Z does not fail and if the resulting constraint set C is solvable. A minimal solution for a constraint set C that assigns concrete effects to effect variables can be found using Algorithm Solve shown in Figure 16.3. Solve is used to solve the constraint set C after the final substitution produced by Z is applied to the constraints. Solve will always succeed because of the [*does*] rule, and thus every expression that is well typed without effects will have both a type and a conservative effect in our type and effect system. Another way to see this is that if two types contain different effects that must be made equal, the [*does*] rule allows us to choose their least upper bound as a common effect. This is precisely what Algorithm Solve does.

▷ **Exercise 16.1** Complete Algorithm Z to handle *impure* let. ◁

▷ **Exercise 16.2** Imagine that `lambda` is extended in FL/R to create a procedure in a region. Thus every procedure type will have a region that identifies where the procedure is located.

a. Give a revised type grammar for FL/R.

b. Give a revised typing rule for `lambda`.

c. Give the revised portion of Algorithm Z for `lambda`.

```
Set all effect variables I_j := pure;
Changed := true;
While Changed
    Changed := false;
    For every constraint C_i
        ; Constraint C_i is of the form (>= I_j E_j) on variable I_j with effect E_j
        ; E_j is evaluated with respect to current effect variable assignments
        If I_j ≠ (maxeff E_j I_j) then begin
            Changed := true;
            I_j := (maxeff E_j I_j);
            End If;
    End For;
End While;
```

Figure 16.3: Algorithm Solve.

d. Procedures can be *stack allocated* when they are *not* returned out of the context where they are created or stored in a cell. Give a rule using the regions in procedure types that identifes procedure regions that can be stack allocated. ◁

▷ **Exercise 16.3** Costs

Sam Antics has a new idea for a type system that is intended to help programmers estimate the running time of their programs. His idea is to develop a set of static rules that will assign every expression a *cost* as well as a type. The cost of an expression is a conservative estimate of how long the expression will take to evaluate.

Sam has developed a new language, called Discount, that uses his cost model. Discount is a call-by-value, statically typed functional language with type reconstruction. Discount is based on FL/R, and inherits its types, with one major difference: a function type in Discount includes the *latent cost* of the function, that is, the cost incurred when the function is called on some arguments.

For example, the Discount type `(-> (int int) 4 int)` is the type of a function that takes two `int`s as arguments, returns an `int` as its result, and has cost at most 4 every time it is called.

The grammar of Discount is shown in Figure 16.1 except that procedure types are altered to include latent costs instead of latent effects:

C ::= `loop` | I | `(sum` C*`)` | `(max` C*`)` | `0` | `1` | `2` | ...

T ::= `int` | `bool` | I | `(->` (T*`)` C T_{body}`)`

Sam has formalized his system by defining type/cost rules for Discount. The rules allow judgments of the form

$$A \vdash E : T \$ C,$$

$(Z[\![\texttt{\#u}]\!]\ A\ S) = \langle \texttt{unit}, \texttt{pure}, S, \texttt{()} \rangle$

$(Z[\![I]\!]\ A[\ldots, I\texttt{:}T, \ldots]\ S) = \langle T, \texttt{pure}, S, \texttt{()} \rangle$

$(Z[\![I]\!]\ A[\ldots, I\texttt{:(generic (}I_1 \texttt{..} I_n\texttt{)}\ T\ C\texttt{)}, \ldots]\ S) =$
$\langle [?D_i/I_i]T, \texttt{pure}, S, [?D_i/I_i]C \rangle$

All $?D_i$ are fresh and represet type, effect, or region variables

$(Z[\![I]\!]\ A\ S) =$ `fail`, where I unbound **in** A

$(Z[\![\texttt{(if}\ E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!]\ A\ S) =$
let $\langle T_{test}, F_{test}, S_{test}, C_{test} \rangle$ **be** $(Z[\![E_{test}]\!]\ A\ S)$ **in**
let ${S_{test}}'$ **be** $U(T_{test}, \texttt{bool}, S_{test})$ **in**
let $\langle T_{con}, F_{con}, S_{con}, C_{con} \rangle$ **be** $(Z[\![E_{con}]\!]\ A\ {S_{test}}')$ **in**
let $\langle T_{alt}, F_{alt}, S_{alt}, C_{alt} \rangle$ **be** $(Z[\![E_{alt}]\!]\ A\ S_{con})$ **in**
let ${S_{alt}}'$ **be**
$U(T_{con}, T_{alt}, S_{alt})$ **in**
$\langle T_{alt}, \texttt{(maxeff}\ F_{test}\ F_{con}\ F_{alt}\texttt{)}, {S_{alt}}', C_{test}\texttt{+}C_{con}\texttt{+}C_{alt} \rangle$

$(Z[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E\texttt{)}]\!]\ A\ S) =$
let $\langle T, F, S, C \rangle$ **be** $(Z[\![E]\!]\ A[I_1\texttt{:?v}_1\ \ldots\ I_n\texttt{:?v}_n]\ S)$ **in**
$\langle \texttt{(->}\ \texttt{(?v}_1\ \ldots\ \texttt{?v}_n\texttt{)}\ \texttt{?e}_n\ T\texttt{)}, \texttt{pure}, S, C\texttt{+((>=}\ \texttt{?e}\ F\texttt{))} \rangle$

$(Z[\![\texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)}]\!]\ A\ S) =$
let $\langle T_{rator}, F_{rator}, S_{rator}, C_{rator} \rangle$ **be** $(Z[\![E_{rator}]\!]\ A\ S)$ **in**
let $\langle T_1, F_1, S_1, C_1 \rangle$ **be** $(Z[\![E_1]\!]\ A\ S_{rator})$ **in**
$\vdots$
let $\langle T_n, F_n, S_n, C_n \rangle$ **be** $(Z[\![E_n]\!]\ A\ S_{n-1}\ C_{n-1})$ **in**
let S_{final} **be** $U(T_{rator}, (->\ (T_1\ \ldots\ T_n)\ \texttt{?e}\ \texttt{?t}))$ **in**
$\langle \texttt{?t}, \texttt{(maxeff}\ F_{rator}\ F_1 \ldots F_n\ \texttt{?e}\texttt{)}, S_{final}, C_{rator}\texttt{+}C_1\texttt{+}\ldots\texttt{+}C_n \rangle$

$(Z[\![\texttt{(let ((}I_1\ E_1\texttt{)..(}I_n\ E_n\texttt{))}\ E\texttt{)}]\!]\ A\ S) =$ `; Pure LET`
let $\langle T_1, F_1, S_1, C_1 \rangle$ **be** $(Z[\![E_1]\!]\ A\ S)$ **in**
$\vdots$
let $\langle T_n, F_n, S_n, C_n \rangle$ **be** $(Z[\![E_n]\!]\ A\ S_{n-l})$ **in**
let $\langle T, F, S, C \rangle$ **be**
$(Z[\![E]\!]$
$A\texttt{[}I_1\texttt{:Zgen(}T_1\texttt{,}\ A\texttt{,}\ S_n\texttt{,}\ C_n\texttt{)}, \ldots, I_n\texttt{:Zgen(}T_n\texttt{,}\ A\texttt{,}\ S_n\texttt{,}\ C_n\texttt{)]}$
$S_n)$ **in**
$\langle T, \texttt{(maxeff}\ F\ F_1 \ldots F_n, S, C\texttt{+}C_1\texttt{+}\ldots\texttt{+}C_n \rangle$

Figure 16.4: Algorithm Z reconstructs Types, Regions, and Effects, Part I

$$
\begin{array}{l}
(Z[\![\texttt{(letrec ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{)) }E\texttt{)}]\!]\ A\ S) = \\
\quad \textbf{let}\ A_1\ \textbf{be}\ A\texttt{[}I_1\texttt{:?t}_1\texttt{,...,}A_n\texttt{:?t}_n\texttt{]}\ \textbf{in} \\
\quad\ \textbf{let}\ \langle T_1, F_1, S_1, C_1\rangle\ \textbf{be}\ (Z[\![E_1]\!]\ A_1\ S)\ \textbf{in} \\
\qquad\qquad \vdots \\
\quad\ \textbf{let}\ \langle T_n, F_n, S_n, C_n\rangle\ \textbf{be}\ (Z[\![E_n]\!]\ A_1\ S_{n-1}\ C_{n-1})\ \textbf{in} \\
\quad\ \textbf{let}\ S_b\ \textbf{be}\ U(\texttt{(?t}_1\ \ldots\texttt{?t}_n\texttt{)}, \texttt{(}T_1\ \ldots\ T_n\texttt{)}, S_n)\ \textbf{in} \\
\quad\ \textbf{let}\ \langle T, F, S, C\rangle\ \textbf{be} \\
\qquad (Z[\![E]\!] \\
\qquad\ A\texttt{[}I_1\texttt{:Zgen(}T_1\texttt{, }A\texttt{, }S_n\texttt{, }C_n\texttt{), ..., }I_n\texttt{:Zgen(}T_n\texttt{, }A\texttt{, }S_n\texttt{, }C_n\texttt{)]} \\
\qquad\ S_b)\ \textbf{in} \\
\qquad \langle T, \texttt{(maxeff }F\ F_1 \ldots F_n, S, C\texttt{+}C_1\texttt{+...+}C_n\rangle
\end{array}
$$

Figure 16.5: Algorithm Z reconstructs Types, Regions, and Effects, Part II

which is pronounced, "in the type environment A, expression E has type T and cost C."

For example, here are Sam's type/cost rules for literals and(non-generic) identifiers:

$$A \vdash U : \texttt{int \$ 1}$$
$$A \vdash B : \texttt{bool \$ 1}$$
$$A[I : T] \vdash I : T\ \texttt{\$ 1}$$

That is, Sam assigns both literals and identifiers a cost of 1. In addition:

- The cost of a `lambda` expression is 2.
- The cost of an `if` expression is 1 plus the cost of the predicate expression plus the maximum of the costs of the consequent and alternate.
- The cost of an N argument application is the sum of the cost of the operator, the cost of each argument, the latent cost of the operator, and N.
- The cost of an N argument primop application is the sum of the cost of each argument, the latent cost of the primop, and N. The latent cost of the primop is determined by a *signature* Σ, a function from primop names to types. For example,
$$\Sigma(\texttt{+}) = \texttt{(-> (int int) 1 int)}.$$

Here are some example judgments that hold in Sam's system:

$$A \vdash \texttt{(primop + 2 1) : int \$ 5}$$
$$A \vdash \texttt{(primop + (primop + 1 2) 4) : int \$ 9}$$
$$A \vdash \texttt{(primop + 2 ((lambda (y) (primop + y 1)) 3)) : int \$ 13}$$

`Loop` is the cost assigned to expressions that may diverge. For example, the expression

```
(letrec ((my-loop (lambda () (my-loop))))
  (my-loop))
```

is assigned cost `loop` in Discount. Because it is undecidable whether an arbitrary expression will diverge, we cannot have a decidable type/cost system in which *exactly* the diverging expressions have cost `loop`. We will settle for a system that makes a *conservative approximation*: every program that diverges will be assigned cost `loop`, but some programs that do not diverge will also be assigned `loop`.

Because Discount has non-numeric costs, like `loop` and cost identifiers (which we won't discuss), it is not so simple to define what we mean by statements like "the cost is the sum of the costs of the arguments...." That is the purpose of the costs `(sum` C_1 C_2`)` and `(max` C_1 C_2`)`. Part of Sam's system ensures that `sum` and `max` satisfy sensible cost equivalent axioms, such as the following:

`(sum` U_1 U_2`)`	=	$U_1 + U_2$
`(sum loop` U`)`	=	`loop`
`(sum` U `loop)`	=	`loop`
`(sum loop loop)`	=	`loop`
`(max` U_1 U_2`)`	=	the max of U_1 and U_2
`(max loop` U`)`	=	`loop`
`(max` U `loop)`	=	`loop`
`(max loop loop)`	=	`loop`

You do not have to understand the details of how cost equivalences are proved in order to solve this problem.

a. Give a type/cost rule for `lambda`.

b. Give a type/cost rule for application.

c. Give a type/cost rule for `if`.

◁

Reading

The first paper on effect systems outlined the need for a new kind of static analysis [LG88], and this early effect system was later extended to include regions in the FX-89 programming language [**?**]. Region and and effect inference were developed next [JG91]. A wide variety of effect systems have been developed, from systems for cost accounting [DJG92, RG94], to control effects [JG89], to region based memory management [**?**]. The FX-91 programming language [GJSO92] included all of these features.

Chapter 17

Compilation

Bless thee, Bottom! bless thee! thou art translated.

— A Midsummer-Night's Dream, II, i, 124, William Shakespeare

17.1 Why do we study compilation?

Compilation is the process of translating a high-level program into low-level machine instructions that can be directly executed by a computer. Our goal in this chapter is to use compilation to further our understanding of advanced programming language features, including the practical implications of language design choices. To be a good designer or user of programming languages, one must know not only how a computer carries out the instructions of a program (including how data are represented) but also the techniques by which a high-level program is converted into something that runs on an actual computer. In this chapter, we will show the relationship between the semantic tools developed earlier in the book and the practice of translating high-level language features to executable code.

Our approach to compilation is rather different than the approach taken in most compiler texts. We assume that the input program is syntactically correct and already parsed, thus ignoring issues of lexical analysis and parsing that are central to real compilers. We also assume that type and effect checking are performed by the reconstruction techniques we have already studied. Our focus will be a series of source-to-source program transformations that implement complex high-level naming, state, and control features by making them explicit in an FL-like intermediate compilation language. In this approach, traditional compilation notions like symbol tables, activation records, stacks, and

code linearization can be understood from the perspective of a simple uniform framework that does not require special-purpose compilation machinery. The result of compilation will be a program in a restricted subset of the intermediate language that is similar in structure to low-level machine code. We thus avoid details of code generation that are critical in a real compiler. Throughout the compilation process, efficiency, though important, will take a back seat to clarity, modularity, expressiveness, and demonstrable correctness.

Although not popular in compiler texbooks, the notion of compilation by source-to-source transformation has a rich history. Beginning with Steele's RABBIT compiler (1978), there has been a long line of research compilers based on this approach. (See the reading section at the end of this chapter for more details.) In homage to RABBIT, we will call our compiler TORTOISE.

We study compilation for the following reasons:

- We can review many of the language features presented earlier in this book in a new light. By showing how they can be transformed into low-level machine code, we arrive at a more concrete understanding of these features.

- We will see how type systems, effect systems, and formal semantics can be applied to the job of compiling a high-level programming language down to a low-level machine architecture.

- We present some simple ways to implement language features by translation. These techniques can be useful in everyday programming, especially if your programming language doesn't support the features that you need.

- We will see how complex translations can be composed out of many simpler passes. Although in practice these passes might be merged, we will discuss them separately for conceptual clarity. Some of these passes have already been mentioned in previous chapters and exercises (e.g., desugaring, assignment conversion, closure conversion, CPS conversion). Here, we study these passes in more depth, introduce some new ones, and show how they fit together to make a compiler.

- We will see that dialects of FL can be powerful intermediate languages for compilation. Many low-level machine details find a surprisingly convenient expression in FL-like languages. Some advantages of structuring our treatment of compilation as a series of source-to-source transformations on one such language are as follows:

  - There is no need to describe a host of disparate intermediate languages.
  - A single intermediate language encourages modularity of translation phases and experimentation with the ordering of phases.
  - The result of every transform phase is executable source code. This makes it easy to read and test the transformation results using a single existing interpreter or compiler for the intermediate language.

- We will see that the inefficiencies that crop up in the compiler are a good motivation for studying static semantics. These inefficiencies are solved by a combination of two methods:

  - Developing smarter translation techniques that take advantage of information known at compile time.
  - Restricting source languages to make them more amenable to static analysis techniques.

  For example, we'll see that dynamically typed languages imply a run-time overhead that can be reduced by clever techniques or eliminated by restricting the language to be statically typable.

These overall goals will be explored in the rest of this chapter. We begin with an overview of the transformation-based architecture of TORTOISE and the languages used in this architecture (Section 17.2). We then discuss the details of each transformation in turn (Sections 17.3–17.12). We conclude by describing the run-time environment for garbage collection (Section 17.13).

17.2 Tortoise Architecture and Languages

17.2.1 Overview of Tortoise

The TORTOISE compiler is organized into ten transformations that incrementally massage a source language program into code resembling register machine code (Figure 17.1). The input and output of each transformation are programs written either in a dialect of FL/R named $\text{FL/R}_{\text{TORTOISE}}$ or an FL-like intermediate language named SILK. In Figure 17.1, one of the SILK dialects have been given a special name: SILK_{tgt} is a subset of SILK that corresponds to low-level machine code. We present $\text{FL/R}_{\text{TORTOISE}}$ and SILK later in this section. The SILK_{tgt} dialect is described in Section 17.12.

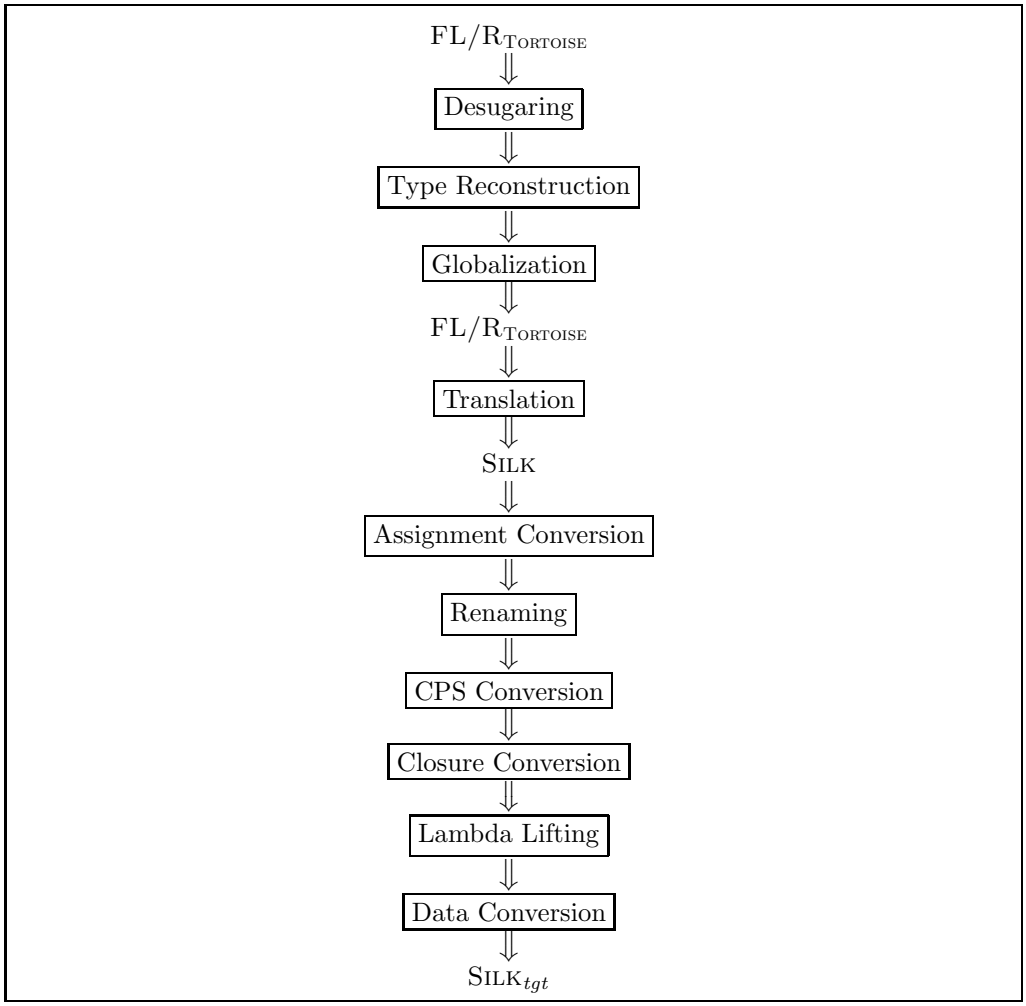


Figure 17.1: Organization of the TORTOISE compiler. After desugaring, type reconstruction, and globalization, a FL/R$_{\text{TORTOISE}}$ source program is translated into the SILK intermediate language, and the SILK program is gradually transformed into a form that resembles register machine code.

Each compiler transformation expects its input program to satisfy certain pre-conditions and produces output code that satisfies certain post-conditions. These conditions will be stated explicitly in the formal specification of each transformation. They will help us understand the purpose of each transformation, and why the compiler is sound. A compiler is **sound** when it produces low-level code that faithfully implements the formal semantics of the compiler's source language. We will not formally prove the soundness of any of the transformations because such such proofs can be very complex. Indeed, soundness proofs for some of these transformations have been the basis for Ph.D. dissertations! However, we will informally argue that the transformations are sound.

TORTOISE implements each transform as a separate pass for clarity of presentation and to allow for experimentation. Although we will apply the transformations in a particular order in this chapter, other orders are possible. Our descriptions of the transformations will explore some alternative implementations and point out how different design choices affect the efficiency and semantics of the resulting code. We generally opt for simplicity over efficiency in our presentation.

17.2.2 The Compiler Source Language: $\text{FL/R}_{\textbf{Tortoise}}$

The source language of the TORTOISE compiler is a slight variant of the FL/R language presented in Chapter 14. Recall that FL/R is a stateful, call-by-value, statically scoped, function-oriented, and statically typed language with type reconstruction that supports mutable cells, pairs, and homogeneous immutable lists. The syntax of TORTOISE language is presented in Figure 17.2. It differs from FL/R in three ways:

- It replaces FL/R's general `letrec` construct with a more specialized `funrec` construct, in which recursively named entities must be manifest abstractions rather than arbitrary expressions. As noted earlier (see Section 7.1), this is a restriction adopted in real languages (such as SML) to avoid thorny issues involving call-by-value recursion. In the context of compilation, we shall see that this restriction simplifies certain transformations.

- Like the FLAVAR! language studied in Section 8.3, it includes mutable variables (changed via `set!`) in addition to mutable cells. These will allow us to show how mutable variables can be automatically converted into mutable cells in the assignment conversion transformation.

- It treats `begin` as a sugar form rather than a kernel form, and uses two new sugar forms: a `let*` construct that facilitates the expression of nested

lets, and a recur form for declaring recursive functions that are immediately called. The other syntactic sugar forms (scand, scor, and list) are inherted from FL.

Figure 17.3 presents a contrived but compact FL/R$_{\text{TORTOISE}}$ program that illustrates many features of the language, such as numbers, booleans, lists, locally defined recursive functions, higher-order functions, tail and non-tail procedure calls, and side effects. We will use it as a running example throughout the rest of this chapter. The revmap procedure takes a procedure f and a list lst and returns a new list that is the reversal of the list obtained by applying f to each element of lst. The accumulation of the new list ans is performed by a local tail recursive loop procedure that is defined using the recur sugar, which abbreviates the declaration and invocation of a recursive procedure. The loop procedure performs an iteration in a single state variable xs denoting the unprocessed sublist of lst. Although ans could easily be made to be a second argument to loop, here it is defined externally to loop and updated via set! to illustrate side effects. The example program takes two integer arguments, a and b, and returns a list of the two booleans $((7 \cdot a) > b)$ and $(a > b)$. For example, on the inputs 6 and 17, the program returns the list $[\mathit{true}, \mathit{false}]$.

Note that all primitive names (such as *, >, and cons) may be used as free identifiers in a FL/R$_{\text{TORTOISE}}$ program, where they denote global procedures performing the associated primitive. Thus (primop * E_1 E_2) may be written as (* E_1 E_2) in almost any context. The "almost any" qualifier is required because these names can be assigned and locally rebound like any other names. For example, the program

```
(flr (x y)
  (let ((- +))
    (begin (set! / *) (- (/ x x) (/ y y)))))
```

calculates the sum of the squares of x and y.

17.2.3 The Compiler Intermediate Language: SILK

For the intermediate language of our transformation-based compiler, we use language that we call SILK = Simple Intermediate Language Kernel. Like FL/R$_{\text{TORTOISE}}$, SILK is a stateful, call-by-value, statically scoped, function-oriented language, and it is also a statically typed language with implicit types. However, SILK has a more expressive type system than FL/R$_{\text{TORTOISE}}$ and, unlike FL/R$_{\text{TORTOISE}}$, does not support type reconstruction.

Kernel Grammar

$P \in \text{Program}_{FL/R}$ $I \in \text{Identifier}_{FL/R}$ = usual identifiers
$E \in \text{Exp}_{FL/R}$ $B \in \text{Boollit}_{FL/R} = \{\texttt{\#t}, \texttt{\#f}\}$
$AB \in \text{Abstraction}_{FL/R}$ $N \in \text{Intlit}_{FL/R} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$L \in \text{Lit}_{FL/R}$ $O \in \text{Primop}_{FL/R}$

P ::= `(flr (`I_{fml}`*)` E_{body}`)`
E ::= L | I | AB | `(`E_{rator} E_{rand}`*)` | `(primop` O_{op} E_{arg}`*)`
| `(if` E_{test} E_{then} E_{else}`)` | `(set!` I_{var} E_{rhs}`)` | `(error` I`)`
| `(let ((`I_{name} E_{defn}`)*)` E_{body}`)` | `(funrec ((`I_{name} AB_{defn}`)*)` E_{body}`)`
AB ::= `(lambda (`I_{fml}`*)` E_{body}`)`
L ::= `#u` | B | N

$O_{FL/R}$ ::= `+` | `-` | `*` | `/` | `%` [Arithmetic ops]
| `<=` | `<` | `=` | `!=` | `>` | `>=` [Relational ops]
| `not` | `band` | `bor` [Logical ops]
| `cell` | `^` | `:=` [Mutable cell ops]
| `pair` | `fst` | `snd` [Pair ops]
| `cons` | `car` | `cdr` | `null` | `null?` [List ops]

Syntactic Sugar

`(begin)` $\xrightarrow{desugar}$ `#u`

`(begin` E`)` $\xrightarrow{desugar}$ E

`(begin` E_1 E_{rest}`*)` $\xrightarrow{desugar}$ `(let ((`I E_1`)) (begin` E_{rest}`*))`, where I is fresh

`(let* () `E_{body}`)` $\xrightarrow{desugar}$ E_{body}

`(let* ((`I_1 E_1`)` IE`*)` E_{body}`)` $\xrightarrow{desugar}$ `(let ((`I_1 E_1`)) (let* (`IE`*)` E_{body}`))`
where IE ranges over bindings of the form `(`I E`)`.

`(recur` I_{fcn} `((`I_1 E_1`)` ... `(`I_n E_n`))` E_{body}`)`
$\xrightarrow{desugar}$ `(funrec ((`I_{fcn} `(lambda (`I_1 ...I_n`)` E_{body}`)))`
`(`I_{fcn} E_1 ... E_n`))`

`(scand)` $\xrightarrow{desugar}$ `#t`

`(scand` E_1 E_{rest}`*)` $\xrightarrow{desugar}$ `(if` E_1 `(scand` E_{rest}`*) #f)`

`(scor)` $\xrightarrow{desugar}$ `#f`

`(scor` E_1 E_{rest}`*)` $\xrightarrow{desugar}$ `(if` E_1 `#t (scor` E_{rest}`*))`

`(list)` $\xrightarrow{desugar}$ `(primop null)`

`(list` E_1 E_{rest}`*)` $\xrightarrow{desugar}$ `(primop cons` E_1 `(list` E_{rest}`*))`

Standard Identifiers

I_{std} ::= O

Figure 17.2: Syntax for FL/R_{TORTOISE}, the source language of the TORTOISE compiler.

```
(flr (a b)
  (let ((revmap
         (lambda (f lst)
           (let ((ans (null)))
             (recur loop ((xs lst))
               (if (null? xs)
                   ans
                   (begin
                     (set! ans (cons (f (car xs)) ans))
                     (loop (cdr xs)))))))))
    (revmap (lambda (x) (> x b))
            (list a (* a 7)))))
```

Figure 17.3: Running example.

17.2.3.1 The Syntax of SILK

The syntax of SILK is specified in Figure 17.4. SILK is similar to many of the stateful variants of FL that we have studied, especially FLAVAR!. Some notable features of SILK are:

- Multi-argument abstractions and applications are hardwired into the kernel rather than being treated as syntactic sugar. As in FL/R$_{\text{TORTOISE}}$, the abstraction keyword is `lambda`. Unlike in FL/R$_{\text{TORTOISE}}$, SILK applications have an explicit `call` keyword.

- Multi-binding `let` expressions are considered kernel expressions rather than sugar for applications of manifest abstractions.

- It has mutable variables (changed via `set!`) and mutable tuples (which are created via `mprod` and whose component slots are accessed via `mget` and changed via `mset!`). We treat `mget` and `mset!` as "indexed primitives" `(mget` S_{index}`)` and `(mset!` S_{index}`)` in which the primitive operator includes the index S_{index} of the manipulated component slot. If we wrote `(primop mget` E_{index} E_{mp}`)`, this would imply that the index could be calculated by an arbitrary expression E_{index} when in fact it must be a positive integer literal S_{index}. So we instead write `(primop (mget` S_{index}`)` E_{mp}`)`. Treating `mget` and `mset!` as primitives rather than as special constructs simplifies the definition of several transformations.

- Other data include integers, booleans, and immutable lists. Unlike FL/R$_{\text{TORTOISE}}$, SILK does not include cells and pairs; these are modeled via mutable tuples.

Kernel Grammar

$P \in \text{Program}_{Silk}$ $I \in \text{Identifier} = \text{usual identifiers}$
$E \in \text{Exp}_{Silk}$ $B \in \text{Boollit} = \{\texttt{\#t}, \texttt{\#f}\}$
$BV \in \text{BindingValue}_{Silk}$ $N \in \text{Intlit} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$DV \in \text{DataValue}_{Silk}$ $S \in \text{Poslit} = \{\texttt{1}, \texttt{2}, \texttt{3}, \ldots\}$
$AB \in \text{Abstraction}_{Silk}$ $O \in \text{Primop}_{Silk}$
$L \in \text{Lit}_{Silk}$

P ::= `(silk (`I_{fml}`*) `E_{body}`)`
E ::= L | I | AB | `(if `E_{test} E_{then} E_{else}`)` | `(set! `I_{var} E_{rhs}`)`
| `(call `E_{rator} E_{rand}`*)` | `(primop `O_{op} E_{arg}`*)`
| `(let ((`I_{name} E_{defn}`)*) `E_{body}`)` | `(cycrec ((`I_{name} BV_{defn}`)*) `E_{body}`)`
| | `(error `I`)`
AB ::= `(lambda (`I_{fml}`*) `E_{body}`)`
BV ::= L | AB | `(primop mprod `DV`*)`
DV ::= L |I
L ::= `#u` | B | N

O ::= `+` | `-` | `*` | `/` | `%` [Arithmetic ops]
| `<=` | `<` | `=` | `!=` | `>` | `>=` [Relational ops]
| `not` | `band` | `bor` [Logical ops]
| `mprod` | `(mget `S`)` | `(mset! `S`)` [Mutable tuples]
| `cons` | `car` | `cdr` | `null` | `null?` [List ops]
| *... add data conversion bit ops here ...*

Syntactic Sugar

`(@mget `S E_{mp}`)` $\xrightarrow{desugar}$ `(primop (mget `S`) `E_{mp}`)`
`(@mset! `S E_{mp} E_{new}`)` $\xrightarrow{desugar}$ `(primop (mset! `S`) `E_{mp} E_{new}`)`
`(@`O E_1 $\ldots$ E_n`)` $\xrightarrow{desugar}$ `(primop `O E_1 $\ldots$ E_n`)`, where $O \notin \{\texttt{mget}, \texttt{mset!}\}$
`(let* () `E_{body}`)` $\xrightarrow{desugar}$ E_{body}
`(let* ((`I_1 E_1`) `IE`*) `E_{body}`)` $\xrightarrow{desugar}$ `(let ((`I_1 E_1`)) (let* (`IE`*) `E_{body}`))`
where IE ranges over bindings of the form `(`I E`)`.

Figure 17.4: Syntax for SILK, the TORTOISE compiler intermediate language.

- Unlike FL/R$_{\text{TORTOISE}}$, SILK does not have any globally bound standard identifiers for procedures like `+`, `<`, and `cons`. This means that all well-typed SILK programs are closed (i.e., have no free variables).

- Recursion is handled via a `cycrec` form. Syntactically, `cycrec` is similar to FL's `letrec` form, except that the definitions appearing in a binding are restricted to be simple syntactic values in the BindingValue domain: literals, abstractions, and mutable tuple creations. The components in a mutable tuple creations are required to be syntactic values in the DataValue domain: either literals or identifiers. Both BindingValue and DataValue are restricted subsets of Exp. As we shall see in Section 17.2.3.2, these syntactic restrictions allow `cycrec` to specify cyclic data structures and avoid some thorny semantic issues in the more general `letrec` construct. In contrast, FL/R$_{\text{TORTOISE}}$'s `funrec` can only specify mutually recursive procedures, not cyclic data structures.

To improve the readability of SILK programs, we will use the syntactic sugar specified in Figure 17.4. The `@` notation is a more concise way of writing primitive applications. E.g., `(@+ 1 2)` abbreviates `(primop + 1 2)` and `(@mget 1 t)` abbreviates `(primop (mget 1) t)`. As in FL/R$_{\text{TORTOISE}}$, `let*` abbreviates a sequence of nested single-binding `let` expressions. Throughout the rest of this chapter, we will "resugar" expressions using these abbreviations in all code examples to make them more readable.

The readability of SILK programs is further enhanced if we assume that the syntactic simplifications in Figure 17.5 are performed when SILK ASTs are constructed. These simplifications automatically remove some of the "silly" inefficiencies that can be introduced by transformations. In transformation-based compilers, such simplifications are typically performed via a separate simplifying transformation, which may be called several times in the compilation process. However, building the simplifications into the AST constructors is an easy way to guarantee that the inefficient forms are never constructed in the first place. The conciseness and readability of the SILK examples in this chapter is due in large part to these simplifications. Putting all the simplifications in one place means that individual transformations do not need to implement any simplifications, so this also simplifies the specification of transformations.

The [*empty-let*] and [*empty-cycrec*] rules remove trival instances of `let` and `cycrec`. The [*implicit-let*] rule treats an application of a manifest `lambda` as a `let` expression. The [*eta-lambda*] rule performs **eta reduction** on an abstraction. The requirement that E_{rator} be a variable or abstraction is a simple syntactic constraint guaranteeing that E_{rator} is pure. If E_{rator} is impure, the

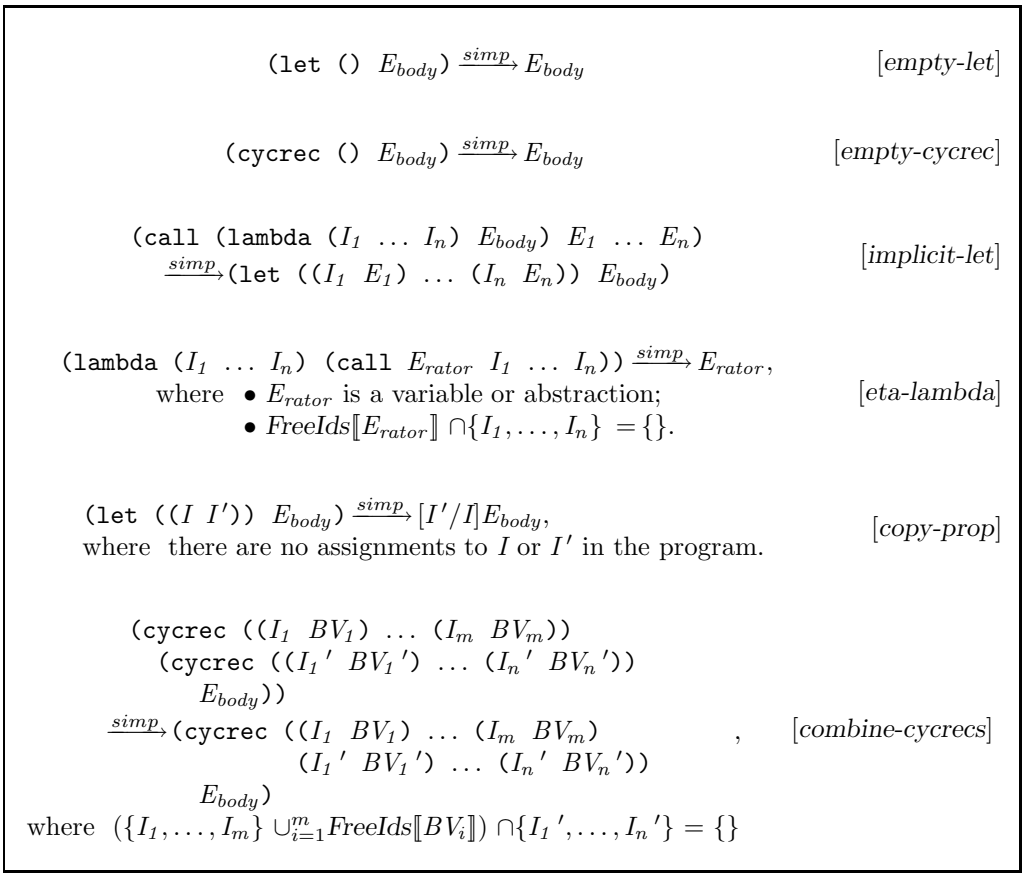


Figure 17.5: Simplifications performed when constructing SILK ASTs.

simplification is unsound because it could change the order of side effects in (and thus the meaning of) the program. For example, it is safe to simplify `(lambda (a b) (call f a b))` to `f`, but is it is not safe to simplify

```
(lambda (a b) (call (begin (set! c (@+ c 1)) f) a b))
```

to

```
(begin (set! c (@+ c 1)) f)
```

Of course, an `lambda` cannot be eliminated by [*eta-lambda*] if E_{rator} mentions one of its formal parameters, as in

```
(lambda (a) (call (lambda (b) (@+ a b)) a)).
```

The [*copy-prop*] rule performs a **copy propagation** simplification that is an important optimization in traditional compilers. This simplification removes a `let` that simply introduces one variable to rename the value of another. Recall that $[I'/I]E$ denotes the *capture-free* substitution of I' for I in E, renaming bound variables as necessary to prevent variable capture. In the presence of assignments involving I or I', the simplification can be unsound (see Exercise 17.1), so these are outlawed. The [*combine-cycrec*] rule combines nested `cycrec` expressions into a single `cycrec` in cases where no variable capture would occur.

The [*empty-let*], [*empty-cycrec*], and [*implicit-let*] simplifications are easy to perform in any context. The [*eta-lambda*] and [*combine-cycrec*] rules require information about the free identifiers of subexpressions. These can be efficiently performed in practice if each AST node is annotated with its free identifiers. The [*copy-prop*] rule requires global information about assignments. For simplicity, we assume the TORTOISE compiler does not perform [*copy-prop*] simplifications until after the assignment conversion stage, when it is guaranteed that there are no assignments in the entire program.

▷ **Exercise 17.1** Consider the following SILK program skeleton:

```
(silk (a)
  (let ((f E_fun))
    (let ((b a)) E_body)))
```

For each of the following scenarios, develop an example in which applying the [*copy-prop*] simplification rule to `(let ((a b)) E_body)` is unsound:

a. E_{body} contains an assignment to `a` (but not `b`).

b. E_{body} contains an assignment to `b` (but not `a`).

c. E_{body} contains no assignments to `a` or `b`, but E_{fun} contains an assignment to `a`. ◁

17.2.3.2 The Dynamic Semantics of SILK

SILK is a statically scoped, call-by-value language. Since SILK is a stateful language, the order of expression evaluation matters: the subexpressions of a `call`, arguments of a `primop`, and definition expressions of a `let` are evaluated in left-to-right.

We have studied the semantics for all SILK constructs previously except for `cycrec`. Intuitively, the `cycrec` form is used to specify recursive functions and cyclic data structures. For example, Figure 17.6 depicts the cyclic structure denoted by a sample `cycrec` expression. As we shall see, the cyclic data aspect of `cycrec` comes into play during the closure conversion transformation, where abstractions are transformed into tuples.

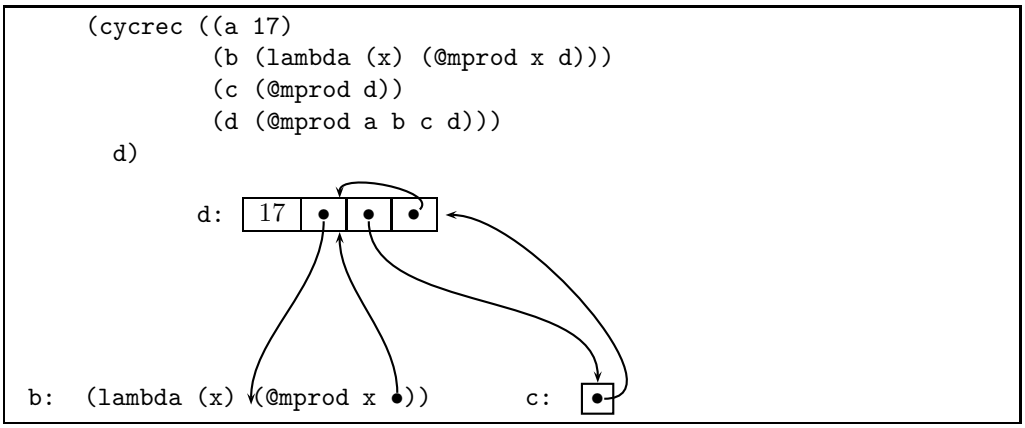


Figure 17.6: A sample `cycrec` expression and the cyclic structure it denotes.

Informally, `(cycrec ((`I_1 BV_1`) ... (`I_n BV_n`)) `E_{body}`)` is evaluated in three stages:

1. First, all the binding value expressions BV_1, ..., BV_n are evaluated. Literals and abstractions are evaluated normally; as in a `letrec`, the `cycrec`-bound variables are in scope within any abstractions. However, mutable tuple creations must be handled specially since their components may reference binding values that have not been determined yet. So only a "skeleton" for each mutable tuple is created. Such a skeleton has the number of slots specified by the creation form, but each slot is initially empty (i.e., is an unassigned location). For example, in Figure 17.6, the first stage binds `c` to a mutable tuple skeleton with one slot and `d` to a skeleton with four slots.

2. Next, the slots of each mutable tuple skeleton are filled in. Recall that mutable tuple binding values have the form `(@mprod` DV_1 ... DV_k`)`. Each data value DV_i is evaluated and is stored in the ith slot of the mutable tuple. Some data values may include references to variables declared by the `cycrec` being processed, but that's OK since these denote values already determined during the first stage. For the `cycrec` in Figure 17.6, the second stage fills in the single slot of the skeleton in `c` with a reference to the `d` skeleton, and fills in the four slots of `d` with (1) the number 17, (2) the procedure named by `b`, (3) the skeleton named by `c`, and (4) the skeleton named by `d`. At the end of the second stage, all binding values have been completely fleshed out.

3. Finally, the body expression E_{body} is evaluated in a scope where the `cycrec`-bound variables denote the values determined in the second stage.

These three stages are formalized in the denotational semantics for `cycrec` presented in Figure 17.7. In the $\mathcal{E}$ clause for `cycrec`, the first stage is modeled by the creation of the triple $\langle e_{fix}, s_{fix}, \langle in_{1_{fix}}, \ldots, in_{n_{fix}} \rangle \rangle$, where:

- e_{fix} is the initial environment e extended with bindings of the `cycrec`-bound variables I_1, ..., I_n to locations holding the binding values for BV_1, ..., BV_n. In the $\mathcal{BV}$ clause for `mprod`, the returned binding value is a mutable tuple skeleton, since the locations returned by *fresh-locs* are initially not filled in.

- s_{fix} is an extension to the initial store s in which locations for the skeletons and environment bindings have been allocated.

- $\langle in_{1_{fix}}, \ldots, in_{n_{fix}} \rangle$ is a tuple of initializer functions associated with each binding value. For mutable tuple creations, these describe how a skeleton should be filled in during the second stage. For literals and abstractions, the associated initializer does nothing.

It is necessary to calculate this triple in the context of a fixed point computation so that abstractions appearing in a binding value are evaluated relative to the fixed point environment e_{fix} containing bindings for each of the `cycrec`-bound variables.

The second stage of evaluation is modeled by the expression

$$\left(init\ [in_{1_{fix}}, \ldots, in_{n_{fix}}]\ \ e_{fix}\ \ s_{fix}\right),$$

which fills in each of the mutable tuple skeletons by invoking the binding value initializers on the extended environment and current store. The third stage of

$$
\begin{array}{l}
c \in \mathit{Computation} = \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store}) \\
\delta \in \mathit{Denotable} = \mathit{Location} \\
\sigma \in \mathit{Storable} = \mathit{Value} \\
mt \in \mathit{MProd} = \mathit{Location}^* \\
p \in \mathit{Procedure} = \mathit{Denotable}^* \to \mathit{Computation} \\
v \in \mathit{Value} = \mathit{Procedure} + \mathit{MProd} + \ldots \\
in \in \mathit{Initializer} = \mathit{Environment} \to \mathit{Store} \to \mathit{Store} \\
vi \in \mathit{VI} = \mathit{Value} \times \mathit{Initializer}
\end{array}
$$

$$
\begin{array}{l}
\mathit{extend}^* : \mathrm{Identifier}^* \to \mathit{Denotable}^* \to \mathit{Environment} \to \mathit{Environment} \\
\mathit{extend}^*\ [\,]_{\mathrm{Identifier}}\ [\,]_{\mathit{Denotable}}\ e = e \\
\mathit{extend}^*\ (I_1\ .\ I_{rest}{}^*)\ (\delta_1\ .\ \delta_{rest}{}^*)\ e = \mathit{extend}^*\ I_{rest}{}^*\ \delta_{rest}{}^*\ ([\delta : e]I)
\end{array}
$$

$$
\begin{array}{l}
\mathit{assign}^* : \mathit{Location}^* \to \mathit{Storable}^* \to \mathit{Store} \to \mathit{Store} \\
\mathit{assign}^*\ [\,]_{\mathit{Location}}\ [\,]_{\mathit{Storable}}\ s = s \\
\mathit{assign}^*\ (l_1\ .\ l_{rest}{}^*)\ (\sigma_1\ .\ \sigma_{rest}{}^*)\ s = \mathit{assign}^*\ l_{rest}{}^*\ \sigma_{rest}{}^*\ (\mathit{assign}\ l\ \sigma\ s)
\end{array}
$$

$$
\begin{array}{l}
\mathit{init} : \mathit{Initializer}^* \to \mathit{Environment} \to \mathit{Store} \to \mathit{Store} \\
\mathit{init}\ [\,]_{\mathit{Initializer}}\ e\ s = s \\
\mathit{init}\ (in_1\ .\ in_{rest}{}^*)\ e\ s = \mathit{init}\ in_{rest}{}^*\ e\ (in\ e\ s)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{BV} : \mathrm{BindingValue} \to \mathit{Environment} \to \mathit{Store} \to (\mathit{Value} \times \mathit{Store} \times \mathit{Initializer}) \\
\mathcal{BV}[\![L]\!]\ e\ s = \langle \mathcal{L}[\![L]\!], s, \lambda e's'\,.\,s' \rangle \\
\mathcal{BV}[\![\texttt{(lambda (}I^*\texttt{)}\ E_{body}\texttt{)}]\!]\ e\ s = \\
\quad \langle (\mathit{Procedure} \mapsto \mathit{Value}\ (\lambda\delta^*\,.\,\mathcal{E}[\![E_{body}]\!](\mathit{extend}^*\ I^*\ \delta^*\ e))), s, \lambda e's'\,.\,s' \rangle \\
\mathcal{BV}[\![\texttt{(primop mprod }DV_1\ \ldots\ DV_n\texttt{)}]\!]\ e\ s = \\
\quad \textbf{let}\ \langle l^*, s' \rangle\ \textbf{be}\ (\mathit{fresh\text{-}locs}\ n\ s) \\
\quad\ \textbf{in}\ \langle (\mathit{MProd} \mapsto \mathit{Value}\ l^*),\ s', \\
\qquad \lambda e'\,.\ \mathit{with\text{-}values}\ (\mathcal{E}^*[\![[DV_1, \ldots, DV_n]]\!]\ e')\ (\mathit{assign}^*\ l^*)\ \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{BV}^* : \mathrm{BindingValue}^* \to \mathit{Environment} \to \mathit{Store} \to ((\mathit{Value} \times \mathit{Initializer})^* \times \mathit{Store}) \\
\mathcal{BV}^*\ [\,]_{\mathrm{BindingValue}}\ e\ s = s \\
\mathcal{BV}^*\ (BV_1\ .\ BV_{rest}{}^*)\ e\ s = \\
\quad \textbf{let}\ \langle v_1, s_1, in_1 \rangle\ \textbf{be}\ \mathcal{BV}\ BV_1\ e\ s \\
\quad\ \textbf{in let}\ \langle vi_{rest}, s_n \rangle\ \textbf{be}\ \mathcal{BV}^*\ BV_{rest}\ e\ s_1\ \textbf{in}\ \langle \langle v_1, in_1 \rangle\ .\ vi_{rest}, s_n \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(cycrec ((}I_1\ BV_1\texttt{) ...(}I_n\ BV_n\texttt{))}\ E_{body}\texttt{)}]\!] = \\
\quad \lambda es\,.\ \textbf{let}\ \langle e_{fix}, s_{fix}, \langle in_{1_{fix}}, \ldots, in_{n_{fix}} \rangle \rangle\ \textbf{be} \\
\qquad\quad \textbf{fix}_{\mathit{Environment} \times \mathit{Store} \times (\mathit{Initializer}^n)} \\
\qquad\qquad (\lambda \langle e_{fix}, s_{fix}, \langle in_{1_{fix}}, \ldots, in_{n_{fix}} \rangle \rangle\,. \\
\qquad\qquad\quad \textbf{let}\ \langle [\langle v_1, in_1 \rangle, \ldots, \langle v_n, in_n \rangle], s'' \rangle\ \textbf{be}\ (\mathcal{BV}^*\ [BV_1, \ldots, BV_n]\ e_{fix}\ s) \\
\qquad\qquad\quad\ \textbf{in let}\ \langle [l_1, \ldots, l_n], s''' \rangle\ \textbf{be}\ (\mathit{fresh\text{-}locs}\ n\ s'') \\
\qquad\qquad\qquad\ \textbf{in}\ \langle [I_1 : l_1] \ldots [I_n : l_n] e, \\
\qquad\qquad\qquad\qquad \mathit{assign}^*\ [l_1, \ldots, l_n]\ [v_1, \ldots, v_n]\ s''', \\
\qquad\qquad\qquad\qquad \langle in_1, \ldots, in_n \rangle \rangle \\
\qquad\quad \textbf{in}\ \mathcal{E}[\![E_{body}]\!]\ e_{fix}\ (\mathit{init}\ [in_{1_{fix}}, \ldots, in_{n_{fix}}]\ e_{fix}\ s_{fix})
\end{array}
$$

Figure 17.7: Denotational semantics of `cycrec`.

evaluation is modeled by evaluating E_{body} relative to the extended environment and the store resulting from the second stage.

Syntactically, `cycrec` is just a restricted form of `letrec`, but semantically it is subtly different. In cases where the binding values are restricted to literals and abstractions, the two forms have the same behavior. But their behavior can differ when binding values include mutable tuples. The `cycrec` form allows the creation of mutually recursive mutable tuples that cannot be expressed via `letrec`. For instance, if we replace the `cycrec` by `letrec` in Figure 17.6, the example would denote an error (or bottom, depending on which `letrec` semantics is used). There is an unresolvable cyclic dependency between the `letrec`-bound name `c` (whose definition expression requires the value of `d`) and the `letrec`-bound name `d` (whose definition expression requires the value of `c`), as well as a cyclic dependency of `d` on itself. Note that `b` is not problematic because the abstraction can be evaluated without immediately requiring the value of the `d` referenced in its body.

The syntactic restrictions of `cycrec` circumvent some of the thorny semantic issues in `letrec`. By construction, BindingValue expressions do not have any side effects (other than allocating mutable tuple skeletons), so issues involving the side effects in `letrec` bindings (see Section **??**) are avoided. Furthermore, the restrictions guarantee that every `cycrec`-bound variable denotes a non-bottom value node in a collection of potentially cyclic abstraction and mutable tuple nodes. They prohibit nonsensical examples like `(cycrec ((a a)) a)`, in which there is no non-trivial value denote by `a`.

▷ **Exercise 17.2** Consider a new form `(mskel` N`)` that creates a mutable tuple skeleton with N unassigned slots. Show that in SILK+{`mskel`}, `cycrec` can be defined as syntactic sugar involving `mskel` and `mset!`. It may be helpful to define some auxiliary functions on BindingValue forms that you use in your desugaring. ◁

17.2.3.3 The Static Semantics of SILK

SILK is an implicitly typed language using the types in Figure 17.8 and type rules in Figure 17.9. SILK types differ from the FL/$\text{R}_{\text{TORTOISE}}$ types as follows:

- they include mutable product types (`mprodof`) in place of cell types (`cellof`) and pair types (`pairof`). The `mprodof` syntax allows an optional `...` at the end, which stands for an unknown number of additional slots of unknown type. The [*mprod*-$\sqsubseteq$] subtyping rule allows any number of `mprod` component types to be "forgotten". It turns out that this will be important for the closure conversion stage.

Types

$T \in$ Type
$\nu \in$ Nonce-Type

$T ::=$ `unit` | `int` | `bool` | `char` | I | ν
| `(listof` T`)` | `(mprodof` T`*` `[...])`
| `(->` `(`T_{arg}`*)` T_{body}`)` | `(tletrec ((`I_{name} T_{defn}`)*)` T_{body}`)`
| `(forall (`I`*)` T`)` | `(exists (`I`*)` T`)`

Subtyping

`(mprodof` T_1 ... T_n`)` $\sqsubseteq$ `(mprodof` T_1 ... T_k `...)`, where $n \geq k$ [*mprod*-$\sqsubseteq$]

Other subtyping rules are as usual.

Figure 17.8: SILK types.

- they include universal types (`forall`, Section 13.2) and existential types (`exists`, Section 15.2).
- they include nonce types (Section 15.3), which are used here to handle elimination of existential types.
- they include recursives types (`tletrec`). These are useful for giving types to the cyclic data structures provided by `cycrec`. For instance, the `cycrec` expression in Figure 17.6 can be given the type

```
(tletrec ((p (mprodof int
                      (forall (t) (-> (t) (mprodof t p)))
                      (mprodof p)
                      p)))
  p)
```

Figure 17.9 presents type rules for implicitly typed constructs that are analogs to many of the rules for the corresponding explicitly typed constructs we have studied earlier. The most interesting rules are for introduction and elimination of universal and existential types, which are much simpler without type annotation syntax like `plambda` and `pcall` (for universals) and `pack` and `unpack` (for existentials). In the implicitly typed setting, the duality between universal and existential types is much clearer. In particular, note the similarity between the [$\forall$-*elim*] rule and the [$\exists$-*intro*] rule.

A a value with universal type can be introduced anywhere and then later be implicitly projected at various types. For example, in

$\vdash$ `#u` : `unit` [*unit*] $\quad \vdash N$: `int` [*int*] $\quad \vdash B$: `bool` [*bool*] $\quad \vdash H$: `char` [*char*]

$\vdash$ `(error` I`)` : T [*error*] $\qquad A \vdash I : A(I)$, where $I \in dom(A)$ [*var*]

$$\frac{A \vdash E_{rhs} : A(I)}{A \vdash \texttt{(set!}\ I\ E_{rhs}\texttt{)} : \texttt{unit}}, \text{ where } I \in dom(A) \qquad [assign]$$

$$\frac{A \vdash E_{test} : \texttt{bool} \;\; ; \;\; A \vdash E_{con} : T \;\; ; \;\; A \vdash E_{alt} : T}{A \vdash \texttt{(if}\ E_{test}\ E_{con}\ E_{alt}\texttt{)} : T} \qquad [if]$$

$$\frac{A[I_1 : T_1, \ \ldots, \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\texttt{->}\textit{-intro}]$$

$$\frac{A \vdash E_{rator} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{result}\texttt{)} \;\; ; \;\; \forall_{i=1}^{n} . \, A \vdash E_i : T_i}{A \vdash \texttt{(call}\ E_{rator}\ E_1\ \ldots\ E_n\texttt{)} : T_{result}} \qquad [\texttt{->}\textit{-elim}]$$

$$\frac{\forall_{i=1}^{n} . \, A \vdash E_i : T_i \;\; ; \;\; A[I_1 : T_1, \ \ldots, \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [let]$$

$$\frac{\forall_{i=1}^{n} . \, A' \vdash BV_i : T_i \;\; ; \;\; A' \vdash E_{body} : T_{body}}{A \vdash \texttt{(cycrec ((}I_1\ BV_1\texttt{)}\ \ldots\ \texttt{(}I_n\ BV_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [cycrec]$$

where $A' = A[I_1 : T_1, \ \ldots, \ I_n : T_n]$

$$\frac{A_{std} \vdash O_{name} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{result}\texttt{)} \;\; ; \;\; \forall_{i=1}^{n} . \;\; A \vdash E_i : T_i}{A \vdash \texttt{(primop}\ O_{name}\ E_1\ \ldots\ E_n\texttt{)} : T_{result}} \qquad [primop]$$

$$\frac{A \vdash E : T}{A \vdash E : \texttt{(forall (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}} \qquad [\forall\textit{-intro}]$$

where E is pure *[purity restriction]*
$\{I_1, \ldots, I_k\} \cap \{(FTV\ \ A(I)) \mid I \in FreeIds[\![E]\!]\} = \{\}$ *[import restriction]*

$$\frac{A \vdash E : \texttt{(forall (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}}{A \vdash E : ([T_i/I_i]_{i=1}^{n})\ \ T} \qquad [\forall\textit{-elim}]$$

$$\frac{A \vdash E : ([T_i/I_i]_{i=1}^{n})\ \ T}{A \vdash E : \texttt{(exists (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}} \qquad [\exists\textit{-intro}]$$

where $\{I_1, \ldots, I_k\} \cap \{(FTV\ \ A(I)) \mid I \in FreeIds[\![E]\!]\} = \{\}$ *[import restriction]*

$$\frac{A \vdash E : \texttt{(exists (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}}{A \vdash E : ([\nu_i/I_i]_{i=1}^{n})\ \ T}, \text{ where } \nu_1, \ldots, \nu_n \text{ are fresh nonce types.} \qquad [\exists\textit{-elim}]$$

$$\frac{A \vdash E : T}{A \vdash E : T'}, \text{ where } T \sqsubseteq T' \qquad [subtype]$$

$$\frac{\{I_1 : \texttt{int}, \ldots, I_n : \texttt{int}\} \vdash E_{body} : T}{\vdash \texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)} : T} \qquad [prog]$$

Figure 17.9: SILK type rules.

```
(let ((id (lambda (x) x)))
  (call (call id id) 3)),
```

`(lambda (x) x)` can be given the type `(forall (t) (-> (t) t))`, and this type can be implicitly projected on `(-> (int) int)` for the first occurrence of `id` and projected on `int` for the second occurrence of `id`.

In SILK, existential types are particularly useful for describing structures that combine procedures with explicit environment components. As we shall see in Section 17.10, such structures are called **closures**. Consider the following expression E_{clo1}:

```
(lambda (b)
  (let ((c1 (@mprod (lambda (env1)
                      (+ (@mget 1 env1) (@mget 2 env1))))
                    (@mprod 4 5))
        (c2 (@mprod (lambda (env2)
                      (if env2 1 0))
                    b)))
    (let ((c (if b c1 c2)))
      (call (@mget 1 c) (@mget 2 c))))).
```

The variables `c1` and `c2` name tuples whose first component is a procedure that expects the second component of the tuple (its "environment") as an argument. The expression E_{clo1} applies the first component of one of these tuples to the second component of the same tuple. Even though the two environments have very different types (the first is a pair of integers; the second is a boolean), E_{clo1} is intuitively a well-typed expression that denotes an integer. This can be shown formally by giving both `c1` and `c2` the following existential type:

T_{clo1} = `(exists (envty) (mprodof (-> (envty) int) envty))`

This type captures the essential similarity between the tuples (both are tuples in which invoking the first component on the second yields an integer) while hiding the inessential details (the types of the two environments are different).

The nonce types that are introduced in the [∃-*elim*] rule serve the role of the user-specified abstract type name I_{ty} in the explicitly typed expression form `(unpack`$_{exist}$ E_{pkg} I_{ty} I_{impl} E_{body}`)`. No export restriction is necessary here because the freshness condition in [∃-*elim*] guarantees that the nonces introduced at different elimination nodes in a type derivation tree will be distinct. This makes it impossible for a nonce introduced by one existential elimination to masquerade as a nonce from another elimination. The subexpression

```
(let ((c (if b c1 c2)))
  (call (@mget 1 c) (@mget 2 c)))
```

of E_{clo1} is well-typed if `c` has type T_{clo1} and the existential is eliminated to yield `(mprodof (-> (`ν_1`) int) `ν_1`)` before the `call` is type-checked. Note that rewriting the subexpression to

```
(call (@mget 1 (if b c1 c2)) (@mget 2 (if b c1 c2)))
```

yields an expression that is not well-typed, since the existential type T_{clo1} would have to be eliminated independently at each `if` expression, and the nonce types introduced for these two eliminations would necessarily be incompatible.

The `exists` type is not powerful enough to describe certain types of closure representations that will be introduced in the TORTOISE compiler. Consider the following expression E_{clo2}, which is a slight variation on E_{clo1}:

```
(lambda (b)
  (let ((c3 (@mprod (lambda (clo3)
                      (+ (@mget 2 clo3) (@mget 3 clo3))))
                    4
                    5)
        (c4 (@mprod (lambda (clo4)
                      (if (@mget 2 clo4) 1 0))
                    b)))
    (let ((c (if b c3 c4)))
      (call (@mget 1 c) c)))).
```

In this expression, the procedure in the first component of each tuple takes the whole tuple as its argument. Again, we expect E_{clo2} to be well-typed with type `int`, but it is challenging to develop a single existential type that abstracts over the differences between `c3` and `c4`. Such a type should presumably look like

```
(tletrec ((cloty (mprodof (-> (cloty) int) <???>))) cloty),
```

but how can can we flesh out the `<???>`?. In `c3`, `<???>` stands for two integer slots in a `mprodof` type, while in in `c4` it stands for a single boolean slot.

To handle this situation, SILK includes mutable product types of the form `(mprodof` T_1 ... T_n `...)`. The first set of ellipses, written "...", is a meta-alanguage abbreviation for all the types between T_1 and T_n. But the set of second ellipses, written "`...`", is an explicit notation in the SILK type syntax that stands for a type variable that is existentially quantified over an unknown number of unknown types. The subtyping rule [*mprod-*$\sqsubseteq$] allows any number of component types at the end of an `mprodof` to be replaced by the ellipses. Types of this form can be introduced into a type derivation via the [*subtype*] rule. For example, since `c3` has the type

```
(tletrec ((cloty (mprodof (-> (cloty) int) int int))) cloty),
```

it also has the following type via [*subtype*] rule:

```
(tletrec ((cloty (mprodof (-> (cloty) int) ...))) cloty).
```

Since `c4` can also be given this type, `c` can also be given this type, and E_{clo2} can be shown to be well-typed with type `int`.

The fact that existential types may be introduced anywhere means that a given SILK expression may have many possible types. For example, `(@mprod 1 #t)` can be given all the following types:

```
(mprodof int bool)
(exists (t) (mprodof t bool))
(exists (t) (mprodof int t))
(exists (t1 t2) (mprodof t1 t2))
(exists (t) t)
```

Similar comments hold for universal types. For example, all of the following types can be assigned to `(@mprod (lambda (x) x) (lambda (y) 3))`:

```
(mprodof (-> (bool) bool) (-> (int) int))
(mprodof (-> (int) int) (-> (bool) int))
(mprodof (forall (t) (-> (t) t)) (-> (char) int))
(mprodof (-> (char) char) (forall (t) (-> (t) int)))
(mprodof (forall (s) (-> (s) s)) (forall (t) (-> (t) int)))
(forall (t) (mprodof (-> (t) t) (-> (t) int)))
(forall (s t) (mprodof (-> (s) s) (-> (t) int)))
```

Indeed, for implicit type systems with full universal and/or existential types, there is not even a notion of "most general" type, so that type reconstruction in such systems is impossible in general [Wel99]. So SILK, unlike FL/R$_{\text{TORTOISE}}$, is *not* a type reconstructable language.

What's the point of considering SILK to be an implicitly typed language if the types cannot be automatically reconstructed?

- The types of the restricted set of SILK programs manipulated by the compiler *can* be automatically determined. Although reconstruction on arbitrary SILK programs is not possible, when a FL/R$_{\text{TORTOISE}}$ program P (which is reconstructable) is initially translated to a SILK program P', it *is* possible to automatically transform the type derivation for P into a type derivation for P'. So the initial SILK program P' is guaranteed to be well-typed. Furthermore, for each of the SILK transformations, it is possible to transform a type derivation of the the input program into a type derivation for the output program. So each transform preserves well-typedness as well as runtime behavior. Note that this approach requires explicitly passing program type derivations through each transform along

with the program.

- The fact that programs are well-typed in each TORTOISE transformation implies important invariants that can be used by the compiler. For example, all well-typed SILK programs are closed, so a transform never needs to handle the case of a global free variable. When the compiler processes the SILK expression `(if` E_1 `0 (@+` E_2 E_3`))`, there is no question that E_1 denotes a boolean value and E_2 and E_3 denote integers. There is no need to handle cases where these expressions might have other types. The compiler uses the fact that each SILK program is implicitly well-typed to avoid generating code for certain run time error checks (see Section 17.12).

Many modern research compilers use so-called **typed intermediate languages (TILs)** that carry explicit type information (possibly including effect, flow, and other analyses information) through all stages of the compiler. In these systems, program transformations transform the types as well as the terms in the programs. In addition to the benefits sketched above, the explicit type information carried by a TIL can be inspected to guide compilation (e.g., determining clever representations for certain types) and can be used to implement run-time operations (such as tag-free garbage collection and checking safety properties of dynamically linked code). It also serves as an important tool for debugging a compiler implementation: if the output of a transformation doesn't type check, the transformation has a bug!

Unfortunately, TILs tend to be very complex. Transforming types in sync with terms can be challenging, and the types in the transformed programs can quickly become so large that they are nearly impossible to read. In the interests of pedagogical simplicity, our SILK intermediate language does not have explicit types, and we only describe how to transform terms and not types. Nevertheless, we maintain the TIL "spirit" by (1) having SILK be an implicitly typed language and (2) imagining that program type derivations are magically transformed by each compiler stage. For more information on TILs, see the reading section.

17.2.4 Purely Structural Transformations

Most of the $\text{FL/R}_{\text{TORTOISE}}$ and SILK program transformations that we shall study can be described by functions that traverse the abstract syntax tree of the program and transform some of the tree nodes but leave most of the tree nodes unchanged. We will say that a transformation is **purely structural** for a given kind of tree node if the result of applying it to that node results in the same kind of node whose children are transformed versions of the chidren of the original node.

We formalize this notion for SILK transformations via the $mapsub_{Silk}$ function defined in Figure 17.10. This function returns a copy of the given SILK expression whose immediate subexpressions have been transformed by a given transformation *tf*. A SILK transformation is purely structural for a given kind of node if its action on that node can be written as an application of $mapsub_{Silk}$.

$tf \in Transform_{Silk} = \text{Exp}_{Silk} \rightarrow \text{Exp}_{Silk}$

$mapsub_{Silk} : \text{Exp}_{Silk} \rightarrow Transform_{Silk} \rightarrow \text{Exp}_{Silk}$

$mapsub_{Silk}[\![L]\!]\ tf = L$

$mapsub_{Silk}[\![I]\!]\ tf = I$

$mapsub_{Silk}[\![\texttt{(error } I_{msg}\texttt{)}]\!]\ tf = \texttt{(error } I_{msg}\texttt{)}$

$mapsub_{Silk}[\![\texttt{(set! } I_{var}\ E_{rhs}\texttt{)}]\!]\ tf = \texttt{(set! } I_{var}\ (tf\ E_{rhs})\texttt{)}$

$mapsub_{Silk}[\![\texttt{(if } E_{test}\ E_{then}\ E_{else}\texttt{)}]\!]\ tf = \texttt{(if } (tf\ E_{test})\ (tf\ E_{then})\ (tf\ E_{else})\texttt{)}$

$mapsub_{Silk}[\![\texttt{(lambda (} I_1\ \ldots\ I_n\texttt{) } E_{body}\texttt{)}]\!]\ tf = \texttt{(lambda (} I_1\ \ldots\ I_n\texttt{) } (tf\ E_{body})\texttt{)}$

$mapsub_{Silk}[\![\texttt{(call } E_{rator}\ E_1\ \ldots\ E_n\texttt{)}]\!]\ tf = \texttt{(call } (tf\ E_{rator})\ (tf\ E_1)\ \ldots\ (tf\ E_n)\texttt{)}$

$mapsub_{Silk}[\![\texttt{(let ((} I_1\ E_1\texttt{) } \ldots\ \texttt{(} I_n\ E_n\texttt{)) } E_{body}\texttt{)}]\!]\ tf$
$= \texttt{(let ((} I_1\ (tf\ E_1)\texttt{) } \ldots\ \texttt{(} I_n\ (tf\ E_n)\texttt{))) } (tf\ E_{body})\texttt{)}$

$mapsub_{Silk}[\![\texttt{(cycrec ((} I_1\ BV_1\texttt{) } \ldots\ \texttt{(} I_n\ BV_n\texttt{)) } E_{body}\texttt{)}]\!]\ tf$
$= \texttt{(cycrec ((} I_1\ (tf\ BV_1)\texttt{) } \ldots\ \texttt{(} I_n\ (tf\ BV_n)\texttt{))) } (tf\ E_{body})\texttt{)}$

$mapsub_{Silk}[\![\texttt{(primop } O\ E_1\ \ldots\ E_n\texttt{)}]\!]\ tf = \texttt{(primop } O\ (tf\ E_1)\ \ldots\ (tf\ E_n)\texttt{)}$

Figure 17.10: The $mapsub_{Silk}$ function simplifies the specification of purely structural transformations.

In the `cycrec` clause for $mapsub_{Silk}$, we take the liberty of applying the transformation *tf* directly to the binding values $BV_1\ \ldots\ BV_n$. Since binding values are a restricted subset of expressions, it is sensible for the input of *tf* to be a binding value, though technically there should be some sort of inclusion function that converts the binding value to an expression. We will omit such inclusion functions for elements of the Abstraction, BindingValue, and DataValue domains throughout our study of transformations in the TORTOISE compiler. More worrisome in the `cycrec` case is the output of $(tf\ BV_i)$. If the result is not in the BindingValue domain, then the `cycrec` form is not syntactically well-formed. So whenever $mapsub_{Silk}$ is applied to `cycrec` forms, we must argue that *tf* maps elements of BindingValue to elements of BindingValue.

As an example of $mapsub_{Silk}$, consider a transformation $\mathcal{IT}$ that rewrites

every occurrence of `(if (primop not` E_1`)` E_2 E_3`)` to `(if` E_1 E_3 E_2`)`. Since SILK expressions are implicitly well-typed, this is a safe transformation. The fact that $\mathcal{IT}$ is purely structural on almost every kind of node is expressed via a single invocation of $mapsub_{Silk}$ in the following definition:

$$\mathcal{IT} : \mathrm{Exp}_{Silk} \to \mathrm{Exp}_{Silk}$$

$$\mathcal{IT}[\![\texttt{(if (primop not } E_1\texttt{)} \ E_2 \ E_3\texttt{)}]\!] = \texttt{(if } (\mathcal{IT}[\![E_1]\!]) \ (\mathcal{IT}[\![E_3]\!]) \ (\mathcal{IT}[\![E_2]\!])\texttt{)}$$

$$\mathcal{IT}[\![E]\!] = mapsub_{Silk}[\![E]\!] \ \mathcal{IT}, \text{ for all other expressions } E.$$

It is not hard to show that $\mathcal{IT}$ transforms every binding value to a binding value, so $mapsub_{Silk}$ is sensible for `cycrec`.

The $mapsub_{Silk}$ function only works for transforming one SILK expression to another. It is straightforward to define a similar $mapsub_{FL/R}$ function that transforms one FL/R$_{\text{TORTOISE}}$ expression to another; we will use this in the globalization transform.

When manipulating expressions, it is sometimes helpful to extract from an expression a collection of its immediate subexpressions. Figure 17.11 defines a $subexps_{FL/R}$ function that returns a sequence of all children expressions of a given FL/R$_{\text{TORTOISE}}$ expression. It is straightforward to define a similar $subexps_{Silk}$ function for SILK expressions.

$$subexps_{FL/R} : \mathrm{Exp}_{FL/R} \to \left(\mathrm{Exp}_{FL/R}{}^*\right)$$

$$subexps_{FL/R}[\![L]\!] = [\,]$$

$$subexps_{FL/R}[\![I]\!] = [\,]$$

$$subexps_{FL/R}[\![\texttt{(error } I_{msg}\texttt{)}]\!] = [\,]$$

$$subexps_{FL/R}[\![\texttt{(set! } I_{var} \ E_{rhs}\texttt{)}]\!] = [E_{rhs}]$$

$$subexps_{FL/R}[\![\texttt{(if } E_{test} \ E_{then} \ E_{else}\texttt{)}]\!] = [E_{test}, E_{then}, E_{else}]$$

$$subexps_{FL/R}[\![\texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{)} \ E_{body}\texttt{)}]\!] = [E_{body}]$$

$$subexps_{FL/R}[\![\texttt{(}E_{rator} \ E_1 \ \ldots \ E_n\texttt{)}]\!] = [E_{rator}, E_1, \ldots, E_n]$$

$$subexps_{FL/R}[\![\texttt{(primop } O \ E_1 \ \ldots \ E_n\texttt{)}]\!] = [E_1, \ldots, E_n]$$

$$subexps_{FL/R}[\![\texttt{(let ((}I_1 \ E_1\texttt{)} \ \ldots \ \texttt{(}I_n \ E_n\texttt{))} \ E_{body}\texttt{)}]\!] = [E_1, \ldots, E_n, E_{body}]$$

$$subexps_{FL/R}[\![\texttt{(cycrec ((}I_1 \ BV_1\texttt{)} \ \ldots \ \texttt{(}I_n \ BV_n\texttt{))} \ E_{body}\texttt{)}]\!] = [BV_1, \ldots, BV_n, E_{body}]$$

Figure 17.11: The $subexps_{FL/R}$ function returns a sequence of all immediate subexpressions of a given FL/R$_{\text{TORTOISE}}$ expression.

17.3 Transform 1: Desugaring

The first pass of the TORTOISE compiler performs desugaring, converting the convenient syntax of FL/R$_{\text{TORTOISE}}$ into a simpler kernel subset of the language. The advantage of having the first transformation desugar the program is that subsequent analyses and transforms are simpler to write and prove correct because there are fewer syntactic forms to consider. Additionally, subsequent transforms also do not require modification if the language is extended or altered through the introduction of new syntactic shorthands.

We will provide preconditions and postconditions for each of the TORTOISE transformations. In the case of desugaring, these are:

Preconditions: The input to the desugaring transform must be a syntactically correct FL/R$_{\text{TORTOISE}}$ program in which sugar forms may occur.

Postconditions: The output of the desugaring transform is a syntactically correct FL/R$_{\text{TORTOISE}}$ program in which there are no sugar forms.

Of course, another postcondition we expect is that the output program should have the same behavior as the input program! This is a fundamental property of each pass that we will not explicitly state in every postcondition.

The desugaring process for FL/R$_{\text{TORTOISE}}$ is similar to one described for FL in Figures 6.3 and 6.4, so we will not repeat the details of the transformation process here. However, since the actual syntactic abbreviations supported by FL and FL/R$_{\text{TORTOISE}}$ are rather different, we highlight the differences:

- In FL, multi-argument procedures and procedure calls are implicitly curried and desugar into abstractions and applications of single-argument procedures. But in FL/R$_{\text{TORTOISE}}$, multi-argument procedures and procedure calls are supported by the kernel language and are not curried.
- In FL, `let` is sugar for application of a manifest `lambda`, but it is considered a kernel form in FL/R$_{\text{TORTOISE}}$.
- In FL, the multi-recursion `letrec` construct desugars into the single-recursion `rec`. In FL/R$_{\text{TORTOISE}}$, the multi-recursion `funrec` is a kernel form.
- In FL, the desugaring of programs translates `define` forms into `letrec` bindings and wraps user expressions in global bindings that declare meanings for standard identifiers like `+` and `cons`. In FL/R$_{\text{TORTOISE}}$, the `define` syntax is not supported, and standard identifiers are handled by the globalization transform discussed in Section 17.5.

- The `scand`, `scor`, and `list` forms are handled in FL/R$_{\text{TORTOISE}}$ just as in FL. The `begin`, `let*`, and `recur` forms were not supported by FL proper, but were considered for various extensions to FL. Other sugared forms supported by FL (such as `cond`) are not included in FL/R$_{\text{TORTOISE}}$ but could easily be added.

Figure 17.12 shows the result of desugaring the reverse mapping example introduced in Figure 17.3. The the `(recur loop ...)` desugars into a `funrec`, the `begin` desugars into a `let` that binds the fresh variable `ignore.0`, an the `list` desugars into a `null`-terminated nested sequences of `cons`es.

```
(flr (a b)
 (let ((revmap
        (lambda (f lst)
          (let ((ans (call null)))
            (funrec
             ((loop
               (lambda (xs)
                 (if (call null? xs)
                     ans
                     (let ((ignore.0
                            (set! ans
                                  (call cons
                                        (call f (call car xs))
                                        ans))))
                       (call loop (call cdr xs)))))))
              (call loop lst))))))
   (call revmap
         (lambda (x) (call > x b))
         (primop cons a
                 (primop cons (call * a 7)
                         (primop null))))))
```

Figure 17.12: Running example after desugaring.

17.4 Transform 2: Type Reconstruction

The second stage of the TORTOISE compiler is type reconstruction. Only well-typed FL/R$_{\text{TORTOISE}}$ (and SILK) programs are allowed to proceed through the rest of the compiler. Because type reconstruction for FL/R$_{\text{TORTOISE}}$ is so similar to that for FL/R (Chapter 14), we do not repeat the details here.

Preconditions: The input to type reconstruction is a syntactically correct kernel FL/R$_{\text{TORTOISE}}$ program.

Postconditions: The output of type reconstruction is a valid kernel program. We will use the term **valid** to describe a program fragment that is both syntactically correct and well-typed.

As discusssed in Section 17.2.3.3, although neither FL/R$_{\text{TORTOISE}}$ nor SILK has explicit types, this does not mean that the type information generated by the type reconstruction phase is thrown away. We can imagine that this type information is passed through the compiler stages via a separate channel, where it is appropriately transformed by each pass. In an actual implementation, this type information might be stored in abstract syntax tree nodes for SILK expressions, in tables symbol tables mapping variable names to their types, or in explicit type derivation trees.

It is worth noting that other analysis information, such as effect information (Chapter **??**) and flow information [NNH98, DWM$^+$01], could be computed at this stage and passed along to other compiler stages.

17.5 Transform 3: Globalization

In general, a program unit being compiled may contain free identifiers that reference externally defined values in standard libraries or other program units. Such free identifiers must somehow be resolved via a **name resolution** process before they are referenced during program execution. Depending on the nature of the free identifiers, name resolution can take place during compilation, during a linking phase that typically takes places after compilation but before execution (see Section 15.5.1), or during the execution of the program unit. In cases where name resolution takes place after compilation, the compiler may still require *some* information about the free identifiers, such as their types, even though their values may be unknown.

In the TORTOISE compiler, we consider a very simple form of compile-time linking that resolves references to standard identifiers like `+`, `<`, and `cons`. We will call this linking stage **globalization** because it resolves the meanings of global variables defined in the language. Globalization has the following specification:

Preconditions: The input to globalization is a valid kernel FL/R$_{\text{TORTOISE}}$ program.

Postconditions: The output of globalization is a valid kernel FL/R$_{\text{TORTOISE}}$ program that is *closed* — i.e., it contains no free identifiers.

Removing free identifiers from a program at an early stage simplifies later transformations.

A simple approach to globalization in FL/R$_{\text{TORTOISE}}$ is to wrap the body of the program in a `let` that associates each standard identifier used in the program with an appropriate abstraction (Figure 17.13). This **wrapping strategy** was the approach taken in the desugaring of FL programs in Section 6.2.2.2. In the wrapping strategy, the program

```
(flr (x y) (+ (* x x) (* y y)))
```

would be transformed by globalization into

```
(silk (x y)
  (let ((+ (lambda (v.0 v.1) (primop + v.0 v.1)))
        (* (lambda (v.2 v.3) (primop * v.2 v.3))))
    (+ (* x x) (* y y))))
```

$\mathcal{GW}$: Program$_{FL/R}$ → Program$_{FL/R}$

$\mathcal{GW}[\![$`(flr (`I_1 ... I_n`) `E_{body}`)`$]\!] =$ `(flr (`I_1 ... I_n`) (`$wrap[\![E_{body}]\!]$ `(FreeIds`$[\![E_{body}]\!]$`)))`

$wrap$: Exp$_{FL/R}$ → $\mathcal{P}$(Identifier) → Exp$_{FL/R}$

$wrap[\![E]\!]$ $\{O_1, \ldots, O_n\} =$ `(let ((`O_1 $\mathcal{ABS}[\![O_1]\!]$`) ... (`O_1 $\mathcal{ABS}[\![O_n]\!]$`)) `E`)`

$\mathcal{ABS}$: Primop$_{FL/R}$ → Abstraction

$\mathcal{ABS}[\![O]\!] =$ `(lambda (`I_1 ... I_n`) (primop `O I_1 ... I_n`))`
where I_1, ..., I_n are fresh and ($typeof$ $[\![O]\!]$ $A_{std_{FL/R}}$) = `(->  (`T_1 ... T_n`) `T_{res}`)`.

Figure 17.13: The wrapping approach to globalization.

Constructing an abstraction for a primitive operator (via $\mathcal{ABS}$) requires knowing the number of arguments it takes. In FL/R$_{\text{TORTOISE}}$, this can be determined from the type of the standard identifier naming the global procedure associated with the operator. Note that the program P given to $\mathcal{GW}$ is required to be well-typed, so all the elements of $FreeIds[\![P]\!]$ must be standard identifiers — i.e., names of FL/R$_{\text{TORTOISE}}$ primitives. This illustrates how type-checking a program early in compilation can simplify later stages by eliminating troublesome special cases (in this case, handling unbound identifiers).

A drawback of the wrapping strategy is that global procedures are invoked via the generic procedure calling mechanism rather than the mechanism for invoking primitive operators (`primop`). We will see in later stages of the compiler that the latter is handled far more efficiently than the former. This suggests

an alternative approach in which calls to global procedures are transformed into primitive applications. Replacing a procedure call by a suitably instantiated version of its body is known as **inlining**, so we shall call this the **inlining strategy** for globalization. Using the inlining strategy, the sum-of-squares program would be transformed into:

```
(flr (x y) (primop + (primop * x x) (primop * y y)))
```

There are three situations that need to be carefully handled in the inlining strategy for globalization:

1. A reference to a global procedure can only be converted to an instance of `primop` if it occurs in the rator position of a procedure application. References in other positions must either be handled by wrapping or by converting them to abstractions. Consider the expression `(cons + (cons * (null)))`, which makes a list of two functions. The occurrences of `cons` and `null` can be transformed into `primop`s, but the `+` and `*` cannot be. They can, however, be turned into abstractions containing `primop`s:

   ```
   (primop cons (lambda (v.0 v.1) (primop + v.0 v.1))
     (primop cons (lambda (v.2 v.3) (primop * v.2 v.3))
       (primop null)))
   ```

   Alternatively, we can "lift" the abstractions for `+` and `*` to the top of the enclosing program and name them, as in the wrapping approach.

2. In languages like FL/R$_{\text{TORTOISE}}$, where local identifiers may have the same name as global standard identifiers for primitive operators, care must be taken to distinguish references to global and local identifiers.[1] For example, in the program `(flr (x) (let ((+ *)) (- (+ 2 x) 3)))`, the invocation of `+` in `(+ 2 x)` cannot be inlined, but the invocation of `-` can be:

   ```
   (flr (x)
     (let ((+ (lambda (v.0 v.1) (primop * v.0 v.1))))
       (primop - (+ 2 x) 3)))
   ```

3. In FL/R$_{\text{TORTOISE}}$, the values associated with global primitive identifier names can be modified by `set!`. For example, consider

[1] Many programming languages avoid this and related problems by treating primitive operator names as reserved keywords that may not be used as identifiers in declarations or assignments. This allows compiler writers to inline all primitives.

```
(flr (x y)
  (* (+ x (let ((ignore (set! + -))) y))
     (+ x y))),
```

in which the first occurrence of `+` denotes addition and the second occurrence denotes subtraction. It would clearly be incorrect to replace the second occurrence by an inlined addition primitive. Correctly inlining addition for the first occurrence and subtraction for the second occurrence is possible in this case, but can only be justified by a sophisticated side effect analysis. A simple conservative way to address this problem in the inlining strategy is to use wrapping rather than inlining for any global name that is mutated somewhere in the program. For the above example, this yields:

```
(flr (x y)
  (let ((+ (lambda (v.2 v.3) (primop + v.2 v.3))))
    (primop * (+ x (let ((ignore
                          (set! + (lambda (v.0 v.1)
                                    (primop - v.0 v.1)))))
                     y))
             (+ x y)))).
```

All of the above issues are handled by the definition of the inlining approach to globalization in Figure 17.14. The $\mathcal{GI}_{prog}$ function uses $MutIds_{prog}$ (Figure 17.15) to determine the primitive names that are targets of assignment in the program, and wraps the program body in abstractions for these. All other free names are primitives that may be inlined in call positions or expanded to abstractions (via $\mathcal{ABS}$) in other positions. The identifier set argument to $\mathcal{GI}_{exp}$ keeps track of the free global names that have not been locally redeclared.

Figure 17.16 shows the running example after the globalization stage (using the inlining strategy) . In this case, all references to free identifiers have been converted to primitive applications.

▷ **Exercise 17.3** What is the result of globalizing the following program using (1) the wrapping strategy and (2) the inlining strategy?

```
(flr (* /)
  (+ (let ((+ *)) (- + 1))
     (let ((* -)) (* / 2))))
```
◁

▷ **Exercise 17.4** In FL/R$_{\text{TORTOISE}}$, all standard identifiers name primitive procedures. This fact simplifies the globalization transform. Describe how to extend globalization (both the wrapping and inlining strategies) to handle standard identifiers that are (1) literal values (e.g., `zero` standing for `0` and `true` standing for `#t`) and (2) procedures

$\mathcal{GI}_{prog} : \text{Program}_{FL/R} \to \text{Program}_{FL/R}$

$\mathcal{GI}_{prog}[\![P]\!] = (\texttt{flr}\ (I_1\ \ldots\ I_n)\ (wrap[\![\mathcal{GI}_{exp}[\![E_{body}]\!]\ IS_{immuts}]\!]\ IS_{muts}))$
where $P = (\texttt{flr}\ (I_1\ \ldots\ I_n)\ E_{body})$, $IS_{muts} = MutIds_{prog}[\![P]\!]$,
$IS_{immuts} = (FreeIds[\![P]\!]) - IS_{muts}$, $wrap$ is defined in Figure 17.14,
and $MutIds_{prog}$ is defined in Figure 17.15.

$\mathcal{GI}_{exp} : \text{Exp}_{FL/R} \to \text{IdSet} \to \text{Exp}_{FL/R}$

$\mathcal{GI}_{exp}[\![(I_{rator}\ E_1\ \ldots\ E_n)]\!]\ IS$
$= \textbf{if}\ I_{rator} \in IS\ \textbf{then}\ (\texttt{primop}\ I_{rator}\ (\mathcal{GI}_{exp}[\![E_1]\!]\ IS)\ \ldots\ (\mathcal{GI}_{exp}[\![E_n]\!]\ IS))$
$\textbf{else}\ (I_{rator}\ (\mathcal{GI}_{exp}[\![E_1]\!]\ IS)\ \ldots\ (\mathcal{GI}_{exp}[\![E_n]\!]\ IS))\ \textbf{fi}$

$\mathcal{GI}_{exp}[\![I]\!]\ IS = \textbf{if}\ I \in IS\ \textbf{then}\ \mathcal{ABS}[\![I]\!]\ \textbf{else}\ I\ \textbf{fi}$

$\mathcal{GI}_{exp}[\![(\texttt{lambda}\ (I_1\ \ldots\ I_n)\ E_{body})]\!]\ IS$
$= (\texttt{lambda}\ (I_1\ \ldots\ I_n)\ (\mathcal{GI}_{exp}[\![E_{body}]\!]\ (IS - \{I_1, \ldots, I_n\})))$

$\mathcal{GI}_{exp}[\![(\texttt{let}\ ((I_1\ E_1)\ \ldots\ (I_n\ E_n))\ E_{body})]\!]\ IS$
$= (\texttt{let}\ ((I_1\ (\mathcal{GI}_{exp}[\![E_1]\!]\ IS))\ \ldots\ (I_n\ (\mathcal{GI}_{exp}[\![E_n]\!]\ IS)))$
$(\mathcal{GI}_{exp}[\![E_{body}]\!]\ (IS - \{I_1, \ldots, I_n\})))$

$\mathcal{GI}_{exp}[\![(\texttt{funrec}\ ((I_1\ AB_1)\ \ldots\ (I_n\ AB_n))\ E_{body})]\!]\ IS$
$= (\texttt{funrec}\ ((I_1\ (\mathcal{GI}_{exp}[\![AB_1]\!]\ IS'))\ \ldots\ (I_n\ (\mathcal{GI}_{exp}[\![AB_n]\!]\ IS')))$
$(\mathcal{GI}_{exp}[\![E_{body}]\!]\ IS'))$
where $IS' = IS - \{I_1, \ldots, I_n\}$

$\mathcal{GI}_{exp}[\![E]\!]\ IS = mapsub_{FL/R}[\![E]\!]\ (\lambda E_{sub}\,.\,\mathcal{GI}_{exp}[\![E_{sub}]\!]\ IS)$

Figure 17.14: The inlining approach to globalization.

$IS \in \text{IdSet} = \mathcal{P}(\text{Identifier})$

$MutIds_{prog} : \text{Program}_{FL/R} \to \text{IdSet}$

$MutIds_{prog}[\![(\texttt{flr}\ (I_1\ \ldots\ I_n)\ E_{body})]\!] = (MutIds[\![E_{body}]\!]) - \{I_1, \ldots, I_n\}$

$MutIds : \text{Exp}_{FL/R} \to \text{IdSet}$

$MutIds[\![(\texttt{set!}\ I\ E)]\!] = \{I\} \cup MutIds[\![E]\!]$

$MutIds[\![(\texttt{lambda}\ (I_1\ \ldots\ I_n)\ E_{body})]\!] = (MutIds[\![E_{body}]\!]) - \{I_1, \ldots, I_n\}$

$MutIds[\![(\texttt{let}\ ((I_1\ E_1)\ \ldots\ (I_n\ E_n))\ E_{body})]\!]$
$= (\cup_{i=1}^{n} MutIds[\![E_i]\!]) \cup MutIds[\![E_{body}]\!] - \{I_1, \ldots, I_n\}$

$MutIds[\![(\texttt{funrec}\ ((I_1\ AB_1)\ \ldots\ (I_n\ AB_n))\ E_{body})]\!]$
$= (\cup_{i=1}^{n} MutIds[\![AB_i]\!] \cup MutIds[\![E_{body}]\!]) - \{I_1, \ldots, I_n\}$

$MutIds[\![E]\!] = \textbf{let}\ [E_1, \ldots, E_n]\ \textbf{be}\ \ subexps_{FL/R}[\![E]\!]\ \textbf{in}\ \cup_{i=1}^{n} MutIds[\![E_i]\!]$, otherwise.

Figure 17.15: Calculating the mutated free identifiers of a program.

```
(flr (a b)
 (let ((revmap
        (lambda (f lst)
          (let ((ans (primop null)))
            (funrec
             ((loop
               (lambda (xs)
                 (if (primop null? xs)
                     ans
                     (let ((ignore.0
                            (set! ans (primop cons
                                              (call f (primop car xs))
                                              ans))))
                       (call loop (primop cdr xs)))))))
              (call loop lst))))))
   (call revmap
         (lambda (x) (primop > x b))
         (primop cons a
                 (primop cons (primop * a 7)
                         (primop null))))))
```

Figure 17.16: Running example after globalization.

more complex than primitive applications (e.g., `sqr` standing for a squaring procedure and `fact` standing for a factorial procedure). ◁

17.6 Transform 4: Translation

In this transformation, a kernel FL/R$_{\text{TORTOISE}}$ program is translated into the SILK intermediate language. All subsequent transformations are performed on SILK programs.

The translation is performed by the $\mathcal{T}_{\text{prog}}$ and $\mathcal{T}_{\text{exp}}$ functions presented in Figure 17.17. Because the source and target languages are so similar, the translation has the flavor of a transformation that is purely structural except that (1) $\mathcal{T}_{\text{prog}}$ changes the program keyword from `flr` to `silk`; (2) $\mathcal{T}_{\text{exp}}$ converts every `funrec` to a `cycrec`; and (3) $\mathcal{T}_{\text{exp}}$ translates FL/R$_{\text{TORTOISE}}$ cell and immutable pair operations to SILK mutable product operations. We do not give the details of the other cases because they are straightforward. Note that we cannot use the $mapsub_{FL/R}$ or $mapsub_{Silk}$ functions from Section **??** to formally specify these cases because each of these transforms an expression in a language (FL/R$_{\text{TORTOISE}}$ or SILK) to another expression in the *same* language. But $\mathcal{T}_{\text{exp}}$ translates a FL/R$_{\text{TORTOISE}}$ expression to a SILK expression.

The precondition for $\mathcal{T}_{\text{prog}}$ requires a closed FL/R$_{\text{TORTOISE}}$ program. This simplifies the transformation by making it unnecessary to translate global free identifiers like `+` and `cons`. We assume that such free identifiers have already been eliminated by performing globalization. The postcondition does not explicitly mention a closed program because all valid SILK programs are necessarily closed.

Figure 17.18 shows our running example after the translation stage. In this and subsequent code presentations, we shall "resugar" a nested sequence of `let` expressions into a `let*` expression and use the `@` abbreviation for primops to improve the legibility of the code.

It is intuitively clear that $\mathcal{T}_{\text{prog}}$ preserves typability. That is, the output of this translation is well-typed in SILK if the input is well-typed in FL/R$_{\text{TORTOISE}}$. This can be formally proved by showing how a FL/R$_{\text{TORTOISE}}$ type derivation for the original program can be transformed into a SILK type derivation for the translated program. Although types can be reconstructed for the FL/R$_{\text{TORTOISE}}$ input program to $\mathcal{T}_{\text{prog}}$, we make no claims about the type reconstructability of the output SILK program. The programs resulting from $\mathcal{T}_{\text{prog}}$ and some of the subsequent transformations may be restricted enough to support some form of type reconstruction. But in general, the type system of SILK is too expressive to support type reconstruction.

$\mathcal{T}_{\text{prog}} : \text{Program}_{FL/R_{\text{TORTOISE}}} \rightarrow \text{Program}_{Silk}$

Preconditions: The input to $\mathcal{T}_{\text{prog}}$ is a valid closed kernel FL/R$_{\text{TORTOISE}}$ program.

Postconditions: The output of $\mathcal{T}_{\text{prog}}$ is a valid kernel SILK program.

$\mathcal{T}_{\text{prog}}[\![\texttt{(flr (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] = \texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ (\mathcal{T}_{\text{exp}}[\![E_{body}]\!])\texttt{)}$

$\mathcal{T}_{\text{exp}} : \text{Exp}_{FL/R_{\text{TORTOISE}}} \rightarrow \text{Exp}_{Silk}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(funrec ((}I_1\ AB_1\texttt{)}\ \ldots\ \texttt{(}I_n\ AB_n\texttt{))}\ E_{body}\texttt{)}]\!]$
$= \texttt{(cycrec ((}I_1\ \mathcal{T}_{\text{exp}}[\![AB_1]\!]\texttt{)}\ \ldots\ \texttt{(}I_n\ \mathcal{T}_{\text{exp}}[\![AB_n]\!]\texttt{))}\ (\mathcal{T}_{\text{exp}}[\![E_{body}]\!])\texttt{)}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(primop cell }E_1\texttt{)}]\!] = \texttt{(primop mprod }E_1\texttt{)}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(primop ^ }E_{cell}\texttt{)}]\!] = \texttt{(primop (mget 1) }E_{cell}\texttt{)}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(primop := }E_{cell}\ E_{new}\texttt{)}]\!] = \texttt{(primop (mset! 1) }E_{cell}\ E_{new}\texttt{)}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(primop pair }E_1\ E_2\texttt{)}]\!] = \texttt{(primop mprod }E_1\ E_2\texttt{)}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(primop fst }E_{pair}\texttt{)}]\!] = \texttt{(primop (mget 1) }E_{pair}\texttt{)}$

$\mathcal{T}_{\text{exp}}[\![\texttt{(primop snd }E_{pair}\texttt{)}]\!] = \texttt{(primop (mget 2) }E_{pair}\texttt{)}$

All other cases of $\mathcal{T}_{\text{exp}}$ are purely structural.

Figure 17.17: Transformation translating FL/R$_{\text{TORTOISE}}$ into SILK.

```
(silk (a b)
 (let ((revmap
        (lambda (f lst)
          (let ((ans (@null)))
            (cycrec
             ((loop
               (lambda (xs)
                 (if (@null? xs)
                     ans
                     (let ((ignore.0
                            (set! ans (@cons (call f (@car xs)) ans))))
                       (call loop (@cdr xs)))))))
              (call loop lst))))))
   (call revmap
         (lambda (x) (@> x b))
         (@cons a (@cons (@* a 7) (@null))))))
```

Figure 17.18: Running example after translation.

What about meaning preservation? As argued in Section 17.2.3.2, `funrec` and `cycrec` have the same meaning in the case where bindings are abstractions, so the `funrec` to `cycrec` conversion preserves meaning. The cell and pair translations provide alternative implementations of the cell and pair abstract datatypes, so intuitively these preserve meaning as well. But not every aspect of FL/R$_{\text{TORTOISE}}$ meaning is preserved. For example, the program in Figure 17.18 returns a mutable product, whereas the original program returned a pair value. Any formal notion of meaning preservation for this translation would have to account for the type translation as well as the expression translation.

▷ **Exercise 17.5** In the TORTOISE compiler, it would be possible to perform translation *before* globalization rather than after. In this case, assume that (1) globalization is suitably modified to work on SILK programs rather than FL/R$_{\text{TORTOISE}}$ programs and (2) SILK programs are extended to support the same standard identifiers as FL/R$_{\text{TORTOISE}}$ (but in some cases — which ones? — these must have different types than in FL/R$_{\text{TORTOISE}}$.) Describe the advantages and disadvantages of switching the order of these transforms. As a concrete example, consider the following program:

```
(flr (x)
  (let ((c (cell x)))
    (pair (^ c) (+ x 1)))).
```

◁

▷ **Exercise 17.6** Consider the language SILK$_{sum}$ that is just like SILK except:

- it does not have the boolean literals `#t` and `#f`;
- it has no `if` expressions;
- it does not have the list operators `cons`, `car`, `cdr`, `null`, or `null?`;
- it supports oneofs (see Section 10.2 and Section **??**) via the following syntax:

  E ::= ...
  | `(one` I_{tag} E`)` [Oneof Intro]
  | `(tagcase` E_{disc} I_{val} `(`I_{tag} E_{body}`)`* [`(else` E_{else}`)`]) [Oneof Elim]

Show how to translate FL/R$_{\text{TORTOISE}}$ boolean literals, `if` expressions, and list operations into SILK$_{sum}$. ◁

▷ **Exercise 17.7**

Suppose that FL/R$_{\text{TORTOISE}}$'s `funrec` construct were replaced by a `letrec` construct with arbitrary expressions for bindings.

a. Show how to translate `letrec` into SILK. Your translation should be similar to the `letrec` desugaring presented in Section 8.3, except that it needs to preserve typability as well as meaning. *Hint:* Use empty and non-empty lists to distinguish unassigned and assigned variables.

b. `letrec` can also be translated into a target language supporting oneofs, such as SILK$_{sum}$ (see the previous exercise). Give a translation of `letrec` into SILK$_{sum}$.

c. Since `funrec` is a restricted form of `letrec`, the above parts show that it is possible to translate FL/R$_{\text{TORTOISE}}$ into a dialect of SILK that does not contain `cycrec`. What are the advantages and disadvantages of using `cycrec` in the translation of `funrec`? (You may wish to study the remaining stages of the compiler before answering this question.) ◁

17.7 Transform 5: Assignment Conversion

Assignment conversion removes all mutable variables from a program by converting mutable variables to mutable cells. We will say that the resulting program is **assignment-free**.

Assignment conversion makes all mutable storage explicit and simplifies later passes by making all variable bindings immutable. After assignment conversion, all variables effectively denote values rather than implicit cells containing values. A variable may be bound to an explicit cell value whose contents varies with time, but the explicit cell value bound to the variable cannot change. As we will see later in the closure conversion stage (Section 17.10), assignment conversion is important because it allows environments to be treated as immutable data structures that can be freely shared and copied without concerns about side effects.

A straightforward approach to assignment conversion is to make an explicit cell for *every* variable in a given program. For example, the factorial program

```
(silk (x)
  (let ((ans 1))
    (cycrec ((loop (lambda (n)
                     (if (@= n 0)
                         ans
                         (let ((ignore.0 (set! ans (@* n ans))))
                           (call loop (@- n 1)))))))
      (call loop x))))
```

can be assignment converted to

```
(silk (x)
  (let ((x (@mprod x)))
    (let ((ans (@mprod 1)))
      (cycrec ((loop
                (@mprod
                 (lambda (n)
                   (let ((n (@mprod n)))
                     (if (@= (@mget 1 n) 0)
                         (@mget 1 ans)
                         (let ((ignore.0
                                (@mprod
                                 (@mset! 1 ans
                                         (@* (@mget 1 n)
                                             (@mget 1 ans))))))
                           (call (@mget 1 loop)
                                 (@- (@mget 1 n) 1)))))))))
        (call (@mget 1 loop) (@mget 1 x))))))).
```

In the converted program, each of the variables in the original program (`x`, `ans`, `loop`, `n`, and `ignore.0`) is bound to an explicit cell (i.e., a one-slot mutable product). Each variable reference I in the original program is converted to a cell reference `(@mget 1` I`)`, and each variable assignment `(set!` I E`)` in the original program is converted to an cell assignment of the form `(@mset! 1` I E'`)` (where E' is the converted E).

The code generated by the naïve approach to assignment conversion can contain many unnecessary cell allocations, references, and assignments. A cleverer strategy is to make explicit cells only for those variables that are mutated in the program. Determining exactly which variables are mutated when a program executes is generally undecidable. We employ a simple conservative syntax-based approximation that defines a variable to be mutable if it is `set!` within its scope. In the factorial example, the alternative strategy yields the following program, in which only the `ans` variable is converted to a cell:

```
(silk (x)
  (let ((ans (@mprod 1)))
    (cycrec ((loop (lambda (n)
                     (if (@= n 0)
                         (@mget 1 ans)
                         (let ((ignore.0
                                (@mset! 1 ans (@* n (@mget 1 ans)))))
                           (call loop (@- n 1)))))))
      (call loop x))))
```

The improved approach to assignment conversion is formalized in Figure 17.19.

The $\mathcal{AC}_{prog}$ function wraps the tranformed body of a SILK program in a `let` that binds each mutable program parameter to a cell. The free identifiers syntactically assigned within an expression is determined by the *MutIds* function, which is a SILK version of the FL/R$_{\text{TORTOISE}}$ function defined in Figure 17.15.

Expressions are transformed by the $\mathcal{AC}_{exp}$ function, whose second argument is the set of in-scope identifiers naming variables that have been transformed to cells. Such identifiers are transformed to cell references and assignments, respectively, when processing variable references and variable assignments.

The only other non-trivial cases for $\mathcal{AC}_{exp}$ are the binding forms `lambda`, `let`, and `cycrec`. All of these cases use *partition* to partition the identifiers declared by the forms into two identifier sets: the mutable identifiers IS_m that are assigned somewhere in the given expressions, and the immutable identifiers IS_i that are nowhere assigned. In each of these cases, any subexpression in the scope of the declared identifiers is processed by $\mathcal{AC}_{exp}$ with an identifier set that includes IS_m but excludes IS_i. The exclusion is necessary to prevent converting local immutable variables having the same name as external mutable variables. For example,

```
(silk (x) (@mprod (set! x (@* x 2)) (lambda (x) x)))
```

is converted to

```
(silk (x)
  (let ((x (@mprod x)))
    (@mprod (@mset! 1 x (@* (@mget 1 x) 2))
            (lambda (x) x))))
```

Even though the program parameter `x` is converted to a cell, the `x` in the abstraction body is not.

Abstractions are processed like programs in that the transformed abstraction body is wrapped in a `let` binding each mutable identifier to a cell. This preserves the call-by-value-sharing semantics of SILK since an assignment to the formal parameter of an abstraction modifies the contents of a local cell initially containing a *copy* of the parameter value.

In processing `let` and `cycrec`, *maybe-cell* is used to wrap the binding expressions for mutable identifiers in a cell. These two forms are processed similarly except for scoping differences in their declared names.

In the precondition for $\mathcal{AC}_{prog}$ in Figure 17.19, there is a subtle restriction involving `cycrec` that is a consequence of its syntax. Recall that `cycrec` expressions have the form `(cycrec ((`I BV`)*)` E_{body}`)`, where a binding value BV is either a literal, abstraction, or mutable product of the form `(@mprod` DV`*)`,

$\mathcal{AC}_{prog}$: $\text{Program}_{Silk} \rightarrow \text{Program}_{Silk}$

Preconditions: The input to $\mathcal{AC}_{prog}$ is a valid, closed, kernel SILK program in which no unguarded variable appearing in a `cycrec` binding is assigned to.

Postconditions: The output of $\mathcal{AC}_{prog}$ is a valid, closed, assignment-free, kernel SILK program.

$\mathcal{AC}_{prog}[\![\texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!]$
$= \texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ (\textit{wrap-cells}\ IS_{muts}\ \ (\mathcal{AC}_{exp}[\![E_{body}]\!]\ IS_{muts})))$
where $IS_{muts} = \mathit{MutIds}[\![E_{body}]\!]$ and *MutIds* is a version of the function defined in Figure 17.15 adapted to SILK.

$\mathcal{AC}_{exp}$: $\text{Exp}_{Silk} \rightarrow \text{IdSet} \rightarrow \text{Exp}_{Silk}$

$\mathcal{AC}_{exp}[\![I]\!]\ IS\ =\ \textbf{if}\ I \in IS\ \ \textbf{then}\ \texttt{(@mget 1}\ I\texttt{)}\ \ \textbf{else}\ I\ \ \textbf{fi}$

$\mathcal{AC}_{exp}[\![\texttt{(set!}\ I\ E\texttt{)}]\!]\ IS\ =\ \texttt{(@mset! 1}\ I\ (\mathcal{AC}_{exp}[\![E]\!]\ IS)\texttt{)}$

$\mathcal{AC}_{exp}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!]\ IS$
$=\ \textbf{let}\ \langle IS_m, IS_i\rangle\ \ \textbf{be}\ (\textit{partition}\ \{I_1, \ldots, I_n\}\ [E_{body}])$
$\quad\textbf{in}\ \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}$
$\qquad(\textit{wrap-cells}\ IS_m\ \ (\mathcal{AC}_{exp}[\![E_{body}]\!]\ \ ((IS \cup IS_m) - IS_i))))$

$\mathcal{AC}_{exp}[\![\texttt{(let ((}I_1\ \ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ \ E_n\texttt{))}\ E_{body}\texttt{)}]\!]\ IS$
$=\ \textbf{let}\ \langle IS_m, IS_i\rangle\ \ \textbf{be}\ (\textit{partition}\ \{I_1, \ldots, I_n\}\ [E_{body}])$
$\quad\textbf{in}\ \texttt{(let ((}I_1\ \ (\textit{maybe-cell}\ I_1\ \ IS_m\ \ (\mathcal{AC}_{exp}[\![E_1]\!]\ IS)))$
$\qquad\ldots\ \texttt{(}I_n\ \ (\textit{maybe-cell}\ I_n\ \ IS_m\ \ (\mathcal{AC}_{exp}[\![E_n]\!]\ IS))))$
$\qquad(\mathcal{AC}_{exp}[\![E_{body}]\!]\ \ ((IS \cup IS_m) - IS_i)))$

$\mathcal{AC}_{exp}[\![\texttt{(cycrec ((}I_1\ \ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ \ E_n\texttt{))}\ E_{body}\texttt{)}]\!]\ IS$
$=\ \textbf{let}\ \langle IS_m, IS_i\rangle\ \ \textbf{be}\ (\textit{partition}\ \{I_1, \ldots, I_n\}\ [E_1, \ldots, E_n, E_{body}])$
$\quad\textbf{in}\ \texttt{(cycrec ((}I_1\ \ (\textit{maybe-cell}\ I_1\ \ IS_m\ \ (\mathcal{AC}_{exp}[\![E_1]\!]\ IS')))$
$\qquad\ldots\ \texttt{(}I_n\ \ (\textit{maybe-cell}\ I_n\ \ IS_m\ \ (\mathcal{AC}_{exp}[\![E_n]\!]\ IS'))))$
$\qquad(\mathcal{AC}_{exp}[\![E_{body}]\!]\ IS')\texttt{)}$
where $IS' = ((IS \cup IS_m) - IS_i)$.

$\mathcal{AC}_{exp}[\![E]\!]\ IS\ =\ \mathit{mapsub}_{Silk}[\![E]\!]\ \ (\lambda E_{sub}\ .\ \mathcal{AC}_{exp}[\![E_{sub}]\!]\ IS)$, otherwise.

wrap-cells : $\text{IdSet} \rightarrow \text{Exp}_{Silk} \rightarrow \text{Exp}_{Silk}$
wrap-cells $\{I_1 \ldots I_n\}\ E\ =\ \texttt{(let ((}I_1\ \ \texttt{(@mprod}\ I_1\texttt{))}\ \ldots\ \texttt{(}I_n\ \ \texttt{(@mprod}\ I_n\texttt{)))}\ E\texttt{)}$

partition : $\text{IdSet} \rightarrow \text{Exp}_{Silk}{}^* \rightarrow (\text{IdSet} \times \text{IdSet})$
partition $IS\ \ [E_1 \ldots E_n]\ =\ \textbf{let}\ IS_M\ \ \textbf{be}\ \cup_{i=1}^{k}(\mathit{MutIds}[\![E_i]\!])\ \ \textbf{in}\ \langle IS \cup IS_M, IS - IS_M\rangle$

maybe-cell : $\text{Identifier} \rightarrow \text{IdSet} \rightarrow \text{Exp}$
maybe-cell $I\ IS\ \ E\ =\ \textbf{if}\ I \in IS\ \ \textbf{then}\ \texttt{(@mprod}\ E\texttt{)}\ \ \textbf{else}\ E\ \ \textbf{fi}$

Figure 17.19: An assignment conversion transformation that converts only those variables that are syntactically assigned in the program.

and a data value DV is either a binding value or an identifier. We will say that any variable reference I appearing in a DV position is **unguarded**. All other variable references occurring in a `cycrec` binding E are necessarily contained within an abstraction in E; we say that these other references are **guarded** (by the abstraction). For example, in

```
(cycrec ((a (@mprod x (lambda (f) (f b))))
         (b (@mprod a (lambda (g) (g x)))))
  (@mprod a b))
```

the first binding expression contains an unguarded `x` and a guarded `b`, while the second binding expression contains an unguarded `a` and a guarded `x`.

Assignment conversion cannot handle a program in which an unguarded variable reference I in a `cycrec` binding needs to be converted to a cell reference `(@mget 1` I`)` because the grammar for DV does not allow an `mget` form. For example, in the above `cycrec`, it would be possible to convert `b` to a cell, but not `a` or `x`. So the precondition for assignment conversion prohibits any program containing an assignment to a variable that appears unguarded in a cycrec binding. In the TORTOISE compiler, it turns out that any program reaching the assignment conversion stage will necessarily satisfy this precondition (see Exercise 17.10).

Figure 17.20 shows our running example after the assignment conversion stage. The only variable assigned in the input program is `ans`, and this is converted to a cell. There are several spots where the *wrap-cells* function introduces empty `let` wrappers of the form `(let () ...)`, but these are removed by the [*empty-let*] simplification in Figure 17.5.

Intuitively, consistently converting a mutable variable along with its references and assignments into explicit cell operations should not change the observable behavior of a program. So we expect that assignment conversion should preserve both the type safety and the meaning of a program. However, formally proving such intuitions can be rather challenging. See [WS97] for a proof that a version of assignment conversion for SCHEME is a meaning-preserving transformation.

▷ **Exercise 17.8** Show the result of assignment converting the following programs using $\mathcal{AC}_{prog}$:

```
(silk (a b)
 (let ((revmap
        (lambda (f lst)
          (let ((ans (@mprod (@null)))) ; converted to a cell
            (cycrec
             ((loop
               (lambda (xs)
                 (if (@null? xs)
                     (@mget 1 ans)
                     (let ((ignore.0
                            (@mset! 1 ans
                                    (@cons (call f (@car xs))
                                           (@mget 1 ans)))))
                       (call loop (@cdr xs)))))))
              (call loop lst))))))
   (call revmap
         (lambda (x) (@> x b))
         (@cons a (@cons (@* a 7) (@null))))))
```

Figure 17.20: Running example after assignment conversion.

```
(silk (a b c)
  (@mprod (set! a (@+ a 1))
          (lambda (a d)
            (let ((ignore.0 (set! c (@* a b))))
              (set! d (@+ c d))))))

(silk (x)
  (cycrec ((f (lambda (y) (@mprod y (call g (@- y 1)))))
           (g (lambda (z)
                (let ((ignore.0 (set! g (lambda (w) w))))
                  (call f z)))))
    (call f x)))
```

◁

▷ **Exercise 17.9** Can assignment conversion be performed before globalization? Explain. (Assume that $\mathcal{AC}_{prog}$ and $\mathcal{AC}_{exp}$ are suitably modified to work on FL/R$_{\text{TORTOISE}}$ programs rather than SILK programs.) ◁

▷ **Exercise 17.10** Argue that any SILK program that is the result of applying the first four stages of the TORTOISE compiler (desugaring, type-checking, globalization, and translation) automatically satisfies all the preconditions for $\mathcal{AC}_{prog}$. ◁

▷ **Exercise 17.11** A straightforward implementation of the $\mathcal{AC}_{prog}$ and $\mathcal{AC}_{exp}$ func-

tions in Figure 17.19 is inefficient because (1) it traverses the AST of every declaration node at least twice: once to determine the free mutable identifiers, and once to transform the node; and (2) it may recalculate the free mutable identifiers for the same expression many times. Describe how to modify the assignment conversion algorithm so that it works in a single traversal over the program AST and calculates the free mutable identifiers only once at every node. Note: you may need to modify the information stored in the nodes of a SILK AST. ◁

17.8 Transform 6: Renaming

A program fragment is **uniquely named** if no two logically distinct variables appearing in the fragment have the same name. For example, the following two expressions have the same structure and meaning, but the second is uniquely named while the first is not:

```
((lambda (x) (x w)) (lambda (x) (let ((x (* x 2))) (+ x 1))))

((lambda (x) (x w)) (lambda (y) (let ((z (* y 2))) (+ z 1))))
```

Several of the subsequent program transformations we will study require that programs are uniquely named to avoid problems with variable capture or otherwise simplify the transformation. Here we describe a renaming transformation whose output program is a uniquely named version of the input program. We will argue that subsequent transformations preserve the unique naming property. This means that the property will hold for all those transformations that require it of input programs.

The renaming transformation is presented in Figure 17.21. In this transformation, every bound identifier in the program is replaced by a fresh identifier. Fresh names are introduced in all declaration forms: the `silk` program form and the `lambda`, `let`, and `cycrec` expression forms. Renaming environments in the domain *RenEnv* are used to associate these fresh names with the original names and communicate the renamings to the variable reference and assignment forms. Renaming is a purely structural transformation for all other nodes.

As in many other transformations, we gloss over the mechanism for generating fresh identifiers. This mechanism can be formally specified and implemented by threading some sort of name generation state through the transformation. For example, this state could be a natural number that is initially 0 and is incremented every time a fresh name is generated. The fresh name can combine the original name and the number in some fashion. In our examples, we assume that renamed identifiers have the form *prefix*.*number*, where *prefix* is

Renaming Environments

$re \in RenEnv = \text{Identifier} \to \text{Identifier}$

$rbind : \text{Identifier} \to \text{Identifier} \to RenEnv \to RenEnv$
$= \lambda I_{old}\ I_{new}\ re\ .\ \lambda I_{key}\ .\ \textbf{if}\ (\textit{same-identifier?}\ I_{key}\ I_{old})\ \textbf{then}\ I_{new}\ \textbf{else}\ (re\ I_{key})$

$rbind\ I_{old}\ I_{new}\ re$ will be abbreviated $[I_{old}{:}I_{new}]re$; this notation associates to the right. I.e., $[I_1{:}I_1{\,'}][I_2{:}I_2{\,'}]re = [I_1{:}I_1{\,'}]([I_2 : I_2{\,'}]re)$

Renaming Transformation

$\mathcal{R}_{\text{prog}} : \text{Program}_{Silk} \to \text{Program}_{Silk}$

Preconditions: The input to $\mathcal{R}_{\text{prog}}$ is a valid kernel SILK program.

Postconditions: The output of $\mathcal{R}_{\text{prog}}$ is a valid and uniquely named kernel SILK program.

Other properties: If the input program is assignment-free, so is the output program.

$\mathcal{R}_{\text{prog}}[\![\texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!]$
$= \texttt{(silk (}I_1{\,'}\ \ldots\ I_n{\,'}\texttt{)}\ (\mathcal{R}_{\text{exp}}[\![E_{body}]\!]\ ([I_1 : I_1{\,'}]\ldots[I_n : I_n{\,'}]\,(\lambda I\,.\,I))))$,
where $I_1{\,'}\ \ldots\ I_n{\,'}$ are fresh.

$\mathcal{R}_{\text{exp}} : \text{Exp}_{Silk} \to RenEnv \to \text{Exp}_{Silk}$

$\mathcal{R}_{\text{exp}}[\![I]\!]\ re\ =\ (re\ I)$

$\mathcal{R}_{\text{exp}}[\![\texttt{(set!}\ I\ E\texttt{)}]\!]\ re\ =\ \texttt{(set!}\ (re\ I)\ (\mathcal{R}_{\text{exp}}[\![E]\!]\ re)\texttt{)}$

$\mathcal{R}_{\text{exp}}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!]\ re$
$= \texttt{(lambda (}I_1{\,'}\ \ldots\ I_n{\,'}\texttt{)}\ (\mathcal{R}_{\text{exp}}[\![E_{body}]\!]\ ([I_1 : I_1{\,'}]\ldots[I_n : I_n{\,'}]re))\texttt{)}$,
where $I_1{\,'}\ \ldots\ I_n{\,'}$ are fresh.

$\mathcal{R}_{\text{exp}}[\![\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}]\!]\ re$
$= \texttt{(let ((}I_1{\,'}\ (\mathcal{R}_{\text{exp}}[\![E_1]\!]\ re)\texttt{)}\ \ldots\ \texttt{(}I_n{\,'}\ (\mathcal{R}_{\text{exp}}[\![E_n]\!]\ re)\texttt{))}$
$(\mathcal{R}_{\text{exp}}[\![E_{body}]\!]\ ([I_1 : I_1{\,'}]\ldots[I_n : I_n{\,'}]re))\texttt{)}$,
where $I_1{\,'}\ \ldots\ I_n{\,'}$ are fresh.

$\mathcal{R}_{\text{exp}}[\![\texttt{(cycrec ((}I_1\ BV_1\texttt{)}\ \ldots\ \texttt{(}I_n\ BV_n\texttt{))}\ E_{body}\texttt{)}]\!]\ re$
$= \texttt{(cycrec ((}I_1{\,'}\ (\mathcal{R}_{\text{exp}}[\![E_1]\!]\ re\,'\texttt{))}\ \ldots\ \texttt{(}I_n{\,'}\ (\mathcal{R}_{\text{exp}}[\![E_n]\!]\ re\,'\texttt{)))}\ (\mathcal{R}_{\text{exp}}[\![E_{body}]\!]\ re\,'\texttt{))}$,
where $I_1{\,'}\ \ldots\ I_n{\,'}$ are fresh and $re\,' = ([I_1 : I_1{\,'}]\ldots[I_n : I_n{\,'}]re)$

$\mathcal{R}_{\text{exp}}[\![E]\!]\ re\ =\ mapsub_{Silk}[\![E]\!]\ (\lambda E_{sub}\,.\,\mathcal{R}_{\text{exp}}[\![E_{sub}]\!]\ re)$, otherwise.

Figure 17.21: Renaming transformation.

the original identifier, *number* is the current name generator state value, and `.` is a special character that may appear in compiler-generated names but not user-specified names.[2]Later compiler stages may rename generated names from previous stages; in this case we assume that only the prefix of the old generated name is used as the prefix for the new generated name. For example, `x` can be renamed to `x.17`, and `x.17` can be renamed to `x.42` (not `x.17.42`). Figure 17.22 shows our running example after the renaming stage.

```
(silk (a.1 b.2)
 (let ((revmap.3
        (lambda (f.5 lst.6)
          (let ((ans.7 (@mprod (@null))))
            (cycrec
             ((loop.8
               (lambda (xs.9)
                 (if (@null? xs.9)
                     (@mget 1 ans.7)
                     (let ((ignore.10
                            (@mset! 1 ans.7
                                    (@cons (call f.5 (@car xs.9))
                                           (@mget 1 ans.7)))))
                       (call loop.8 (@cdr xs.9)))))))
              (call loop.8 lst.6))))))
   (call revmap.3
         (lambda (x.4) (@> x.4 b.2))
         (@cons a.1 (@cons (@* a.1 7) (@null))))))
```

Figure 17.22: Running example after renaming.

▷ **Exercise 17.12** What changes need to be made to $\mathcal{R}_{\text{exp}}$ to handle the SILK$_{sum}$ language (see Exercise 17.6)? ◁

▷ **Exercise 17.13**

The multiple bindings $(I_1\ E_1)\ \ldots\ (I_n\ E_n)$ of the `let` form are so-called **parallel bindings** in which the expressions $E_1\ \ldots\ E_n$ cannot refer to any of the internal variables $I_1\ \ldots\ I_n$ – i.e., the ones declared in the bindings. Any occurrences of $I_1\ \ldots\ I_n$ in $E_1\ \ldots\ E_n$ must refer to externally declared variables that happen to have the same names as the internal ones. In contrast, the bindings of the `let*` form (which desugars into nested single-binding `let` forms) are **sequential bindings** in which references to $I_1\ \ldots\ I_{i-1}$ within E_i refer to the internal variables, but references to $I_i\ \ldots\ I_n$ refer to external variables. For example:

[2] *prefix* is not really necessary, since *number* itself is unique. But maintaining the original names helps human readers track variables through the compiler transformations.

```
;; Illustrates parallel bindings
(let ((a 1) (b 2))
  (let ((a (+ b 1))  ; Reference to external b
        (b (* a 2))) ; Reference to external a
    (+ a b)))

;; Illustrates sequential bindings
(let ((a 1) (b 2))
  (let* ((a (+ b 1))  ; Reference to external b
         (b (* a 2))) ; Reference to internal a
    (+ a b)))
```

After the renaming stage, only single-binding `let` forms are necessary, since there can never be any confusion between internal and external names. For instance, after renaming, the above examples can be expressed as:

```
;; Illustrates parallel bindings
(let* ((a.0 1) (b.1 2))
  (let* ((a.2 (+ b.1 1))  ; Reference to external b
         (b.3 (* a.0 2))) ; Reference to external a
    (+ a.2 b.3)))

;; Illustrates sequential bindings
(let* ((a.0 1) (b.1 2))
  (let* ((a.2 (+ b.1 1))  ; Reference to external b
         (b.3 (* a.2 2))) ; Reference to internal a
    (+ a.2 b.3)))
```

We will say that an expression E is in **single binding form** if all `let` expressions occurring within E have single bindings.

a. Modify $\mathcal{R}_{\text{exp}}$ so that the resulting expression is in single binding form.

b. Rather than modifying $\mathcal{R}_{\text{exp}}$, an alternative way to guarantee single binding form after renaming is to add an extra simplification rule [*singlify*] to those presented in Figure 17.5. Define [*singlify*].

c. A disadvantage of the [*singlify*] rule is it makes simplification ambiguous. Show this by giving a simple SILK expression that has two different simplifications depending on whether the [*eta-let*] or [*singlify*] rule is applied first. ◁

▷ **Exercise 17.14** This exercise explores ways to formalize the generation of fresh names in the renaming transformation. Assume that *rename* is a function that renames variables according to the conventions described above. E.g., (*rename* `x` 17) = `x.17` and (*rename* `x.17` 42) = `x.42`.

a. Suppose that the signature of $\mathcal{R}_{\text{exp}}$ is changed to accept and return a natural number that represents the state of the fresh name generator:

$\mathcal{R}_{\text{exp}} : \text{Exp}_{Silk} \to RenEnv \to Nat \to (\text{Exp}_{Silk} \times Nat)$

Give modified definitions of $\mathcal{R}_{\text{prog}}$ and $\mathcal{R}_{\text{exp}}$ in which *rename* is used to generate all fresh names uniquely. Define any auxiliary functions you find helpful

b. An alternative way to thread the name generation state through the renaming transformation is to use continuations. Suppose that the signature of $\mathcal{R}_{\text{exp}}$ is changed as follows:

$\mathcal{R}_{\text{exp}} : \text{Exp}_{Silk} \to RenEnv \to RenameCont \to Nat \to \text{Exp}$

RenameCont is a renaming continuation defined as follows:

$rc \in RenameCont = \text{Exp} \to Nat \to \text{Exp}$

Give modified definitions of $\mathcal{R}_{\text{prog}}$ and $\mathcal{R}_{\text{exp}}$ in which *rename* is used to generate all fresh names uniquely. Define any auxiliary functions you find helpful.

c. The *mapsub* function cannot be used in the above two parts because it does not thread the name generation state through the processing of subexpressions. Develop modified versions of *mapsub* that would handle the purely structural cases in the above parts. ◁

17.9 Transform 7: CPS Conversion

In Chapter 9, we saw that continuations are a powerful mathematical tool for modeling sophisticated control features like non-local exits, unrestricted jumps, exceptions, backtracking, coroutines, and threads. Section 9.2 showed how such features can be simulated in any language supporting first-class procedures. The key idea in these simulations is to represent a possible future of the current computation as an explicit procedure, called a **continuation**. The continuation takes as its single parameter the value of the current computation. When invoked, the continuation proceeds with the rest of the computation. In these simulations, procedures no longer return to their caller when invoked. Rather, they are transformed so that they take one or more explicit continuations as arguments and invoke one of these continuations on their result instead of returning the result. A program in which every procedure invokes an explicit continuation parameter in place of returning is said to be written in **continuation-passing style (CPS)**.

As an example of CPS, consider the SILK expression E_{sos} in Figure 17.23. It defines a squaring procedure `sqr` and a sum-of-squares procedure `sos` and applies the latter to `3` and `4`. E_{sos}^{cps} is the result of transforming E_{sos} into CPS form. In E_{sos}^{cps}, each of the two procedures `sqr` and `sos` has been extended with a continuation parameter, which by our convention will come last in the

```
E_sos = (let* ((sqr (lambda (x) (@* x x)))
               (sos (lambda (a b) (@+ (call sqr a) (call sqr b)))))
         (call sos 3 4))

E_sos^cps = (let* ((sqr_cps (lambda (x ksqr) (call ksqr (@* x x))))
                   (sos_cps (lambda (a b ksos)
                              (call sqr_cps a
                                (lambda (asqr)
                                  (call sqr_cps b
                                    (lambda (bsqr)
                                      (call ksos (@+ asqr bsqr)))))))))
             (call sos_cps 3 4 klet*))
```

Figure 17.23: E_{sos}^{cps} is a CPS version of E_{sos}.

parameter list and begin with the letter `k`. The $\texttt{sqr}_{cps}$ procedure invokes its continuation `ksqr` on the square of its input. The $\texttt{sos}_{cps}$ procedure first calls $\texttt{sqr}_{cps}$ on `a` with a continuation that names the result `asqr`. This continuation then calls $\texttt{sqr}_{cps}$ on `b` with a second continuation that names the second result `bsqr`. Finally, $\texttt{sqr}_{cps}$ invokes its continuation `ksos` on the sum of these two results. The initial call `(sos 3 4)` must also be converted. We assume that `klet*` names a continuation that proceeds with the rest of the computation given the value of the `let*` expresssion.

The process of transforming a program into CPS form is called **CPS conversion**. Here we shall study CPS conversion as a stage in the TORTOISE compiler. Whereas globalization makes explicit the meaning of standard identifiers and assignment conversion makes explicit the implicit cells of mutable variables, CPS conversion makes explicit all control flow in a program. Performing CPS conversion as a compiler stage has several benefits:

- *Procedure-calling mechanism*: Continuations are an explicit representation of the procedure call stacks used in traditional compilers to implement the call/return mechanism of procedures. In CPS-converted code, a continuation (such as `(lambda (asqr) ...)` above) corresponds to a pair of (1) a call stack frame that saves variables needed after the call (i.e., the free variables of the continuation, which are `b` and `ksos` in the case of `(lambda (asqr) ...)`) and (2) a return address (i.e., a specification of the code to be executed after the call). Since no CPS procedure returns, every procedure call in a CPS-converted program can be viewed as an assembly code jump that passes arguments. In particular, invoking a continuation corresponds in assembly code to jumping to a return address with a return

value in a distinguished return register.

- *Code linearization*: CPS conversion makes explicit the order in which subexpressions are evaluated, yielding code that linearizes basic computation steps in a way similar to assembly code. For example, the body of $\texttt{sos}_{cps}$ clarifies that the square of `a` is calculated before the square of `b`.

- *Sophisticated control features*: Representing control explicitly in the form of continuations facilitates the implementation of advanced control features (such as non-local exits, exceptions, and backtracking) that can be challenging to implement in traditional stack-based approaches.

- *Uniformity*: Representing control features via procedures keeps intermediate program representations simple and flexible. Moreover, any optimizations that improve procedures will work on continuations as well. But this uniformity also has a drawback: because of its liberal use of procedures, the efficiency of procedure calls in CPS code are of the utmost importance, making certain optimizations almost mandatory.

The TORTOISE CPS transform is presented in four stages. The structure of CPS code is formalized in Section 17.9.1. A straightforward approach to CPS conversion that is easy to understand but leads to intolerable ineffeciences in the converted code is described in Section 17.9.2. Section 17.9.3 presents a more complicated but considerably more efficient CPS transformation that is used in TORTOISE. Finally, we consider the CPS conversion of additional control constructs in Section 17.9.4.

17.9.1 The Structure of CPS Code

All procedure applications can be classified according to their relationship to the innermost enclosing procedure declaration (or program). A procedure application is a **tail call** if its implicit continuation is the same as that of its enclosing procedure. In other words, no computational work must be done between the termination of the inner tail call and the termination of its enclosing procedure; these two events can be viewed as happening simultaneously. All other procedure applications are **non-tail calls**. These are characterized by pending computations that must take place between the termination of the non-tail call and the termination of a call to its enclosing procedure. The notion of a tail call is important in CPS conversion because every procedure call in CPS code must be a tail call. Otherwise, it would have to return to perform a pending computation.

```
AB1 = (lambda (f g x) (call g (call f x) (call f (@+ x 1))))

AB2 = (lambda (p q r s y)
          (let ((a (call p (call q y))))
            (call r a (call s a))))

AB3 = (lambda (filter pred base zs)
          (if (@null? zs)
              (call base zs)
              (if (pred (@car zs))
                  (@cons (@car zs) (call filter pred base (@cdr zs)))
                  (call filter pred base (@cdr zs)))))
```

Figure 17.24: Sample abstractions for understanding tail vs. non-tail calls.

As concrete examples of tail vs. non-tail calls, consider the SILK abstractions in Figure 17.24.

- In AB_1, the call to `g` is a tail call because a call to AB_1 returns a value v when `g` returns v. But both calls to `f` are non-tail calls because the results of these calls must be processed by `g` before AB_1 returns.
- In AB_2, only the call to `r` is a tail call. The results of the calls to `p`, `q`, and `s` must be further processed before AB_2 returns.
- In AB_3, there are two tail calls: the call to `base`, and the second call to `filter`. The result of the first call to `filter` must be processed by `@cons` before AB_3 returns, so this is a non-tail call. The result of `pred` must be checked by the `if`, so this is a non-tail call as well. In this example, we see that (1) a procedure body may have multiple tail calls and (2) the same procedure can be invoked in both tail calls and non-tail calls within the same expression.

Tail and non-tail calls can be characterized syntactically. The SILK contexts in which tail calls can appear is defined by TC in the following grammar:

$TC \in$ TailContext

TC ::=	$\square$	[Hole]
	| `(if` E_{test} TC E`)`	[Left Branch]
	| `(if` E_{test} E TC`)`	[Right Branch]
	| `(let ((`I E`)*)` TC`)`	[Let Body]
	| `(cycrec ((`I BV`)*)` TC`)`	[Cycrec Body]

$P_{cps} \in \text{Program}_{cps}$ $\quad L \in \text{Lit}$
$E_{cps} \in \text{Exp}_{cps}$ $\quad I \in \text{Identifier} = \text{usual identifiers}$
$V_{cps} \in \text{ValueExp}_{cps}$ $\quad B \in \text{Boollit} = \{\texttt{\#t}, \texttt{\#f}\}$
$AB_{cps} \in \text{Abstraction}_{cps}$ $\quad N \in \text{Intlit} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$LE_{cps} \in \text{LetableExp}_{cps}$ $\quad O \in \text{Primop} = \text{as in full SILK.}$
$BV_{cps} \in \text{BindingValue}_{cps}$
$DV_{cps} \in \text{DataValue}_{cps}$

$P_{cps} ::= \texttt{(silk (}I_{fml}\texttt{*) }E_{cps}\texttt{)}$
$E_{cps} ::= \texttt{(call }V_{cps}\ V_{cps}\texttt{*)} \mid \texttt{(if }V_{cps}\ E_{cps}\ E_{cps}\texttt{)} \mid \texttt{(error }I\texttt{)}$
$\quad \mid \texttt{(let ((}I\ LE_{cps}\texttt{)) }E_{cps}\texttt{)} \mid \texttt{(cycrec ((}I\ BV_{cps}\texttt{)*) }E_{cps}\texttt{)}$
$V_{cps} ::= L \mid I$
$AB_{cps} ::= \texttt{(lambda (}I\texttt{*) }E_{cps}\texttt{)}$
$LE_{cps} ::= V_{cps} \mid AB_{cps} \mid \texttt{(primop }O_{op}\ V_{cps}\texttt{*)} \mid \texttt{(set! }I\ V\texttt{)}$
$BV_{cps} ::= L \mid AB_{cps} \mid \texttt{(primop mprod }V_{cps}\texttt{*)}$
$DV_{cps} ::= V \mid AB$
$L ::= \texttt{\#u} \mid B \mid N$

Figure 17.25: Grammar for SILK$_{cps}$, the subset of SILK in CPS form. The result of CPS conversion is a SILK$_{cps}$ program. If the input to CPS is assignment-free, so is the output.

In SILK, a call expression E_{call} is a tail call if and only if the body expression of the innermost abstraction or program enclosing E_{call} is the result $TC\{E_{call}\}$ of filling some context TC with E_{call}. Any call that does not appear in one of these contexts is a non-tail call.

With the notion of tail call in hand, we are ready to study the structure of CPS code, which is defined by the grammar for SILK$_{cps}$, a restricted dialect of SILK presented in Figure 17.25. Observe the following properties of the SILK$_{cps}$ grammar:

- The definition of E_{cps} in SILK$_{cps}$ only allows `call` expressions to appear precisely in the tail contexts TC studied above. So every call in a SILK$_{cps}$ program is guaranteed to be a tail call. In such a program, the implicit continuation of a every call must be exactly the same, so there is never a nontrivial computation (other than the initial continuation of the program invocation) for any call to return to. This is the sense in which calls in a CPS program never return. It also explains why calls in a CPS program can be viewed as assembly-language jumps (that happen to additionally pass arguments).

- Subexpressions of a `call` and `primop` must be literals or variables, so one

application may not be nested within another. The test subexpression of an `if` must also be a literal or variable. The definition subexpression of a `let` can only be one of a restricted number of simple "letable expressions" that does not include `call`s, `if`s, `cycrec`s, or other `let`s. These restrictions impose the straight-line nature of assembly code on the bodies of SILK abstractions and programs, which must be derived from E_{cps}. The only violation of the straight-line property is the `if` expression, which has one E_{cps} subexpression for each branch. This branching code would need to be linearized elsewhere in order to generate assembly language (see Exercise 17.18).

- The order of evaluation for primitive applications is explicitly represented via a sequence of nested single-binding `let` expressions that introduce names for the intermediate results returned by these constructs. For example, CPS converting the expression

```
(@+ (@- 0 (@* b b)) (@* 4 (@* a c)))
```

in the context of an initial continuation `ktop.0` yields:[3]

```
(let* ((t.3 (@* b b))
       (t.2 (@- 0 t.3))
       (t.5 (@* a c))
       (t.4 (@* 4 t.5))
       (t.1 (@+ t.2 t.4)))
  (call ktop.0 t.1))).
```

The `let`-bound names represent abstract registers in assembly code. Mapping these abstract registers to the actual registers of a real machine (a process known as **register allocation**) must be performed by a later compilation stage.

- Every execution path through an abstraction or program body must end in either a `call` or an `error`. Since procedures never return, the last action in a procedure body must be calling another procedure or signaling an error. Moreover, `call`s and `error`s can only appear as the final expression executed in such bodies. Modulo the branching allowed by `if`, program and abstraction bodies in SILK$_{cps}$ are similar in structure to **basic blocks** in traditional compiler technology. A basic block is a sequence of statements such that the only control tranfers into the block are at the very beginning and the only control transfers out of the block are at the very

[3]The particular `let`-bound names used is irrelevant. Here and below, we show the results of CPS conversion using our implementation of the transformation in described in Section 17.9.3.

end.

- Note that SILK$_{cps}$ includes `set!` expressions. In the TORTOISE compiler, both the input and output of CPS conversion will be assignment-free, but in general CPS code may have assignments. Including assignments in SILK$_{cps}$ allows us to experiment with moving assignment conversion after CPS conversion (see Exercise 17.24).

- In classical CPS conversion, abstractions are usually included in the value expressions ValueExp$_{cps}$. However, we require that they be named in a `let` or `cycrec` binding so that certain properties of the SILK$_{cps}$ structure are preserved by later TORTOISE transformations. In particular, the subsequent closure conversion stage will transform abstractions into applications of the `mprod` primitive. Such applications cannot appear in the context of CPS values V_{cps}, but can appear in "letable expressions" LE_{cps}.

The fact that ValueExp$_{cps}$ does not include abstractions or primitive applications means that E^{cps}_{sos} in Figure 17.23 is not a legal SILK$_{cps}$ expression. A SILK$_{cps}$ version of the E_{sos} expression is presented in Figure 17.26. Note that `let`-bound names must be introduced to name abstractions (the continuations `k1` and `k2`) and the results of primitive applications (`t1` and `t2`). Note that some calls (to `sqr`$_{silkcps}$ and `sos`$_{silkcps}$) are to transformed versions of procedures in the original program. These correspond to the jump-to-subroutine idiom in assembly code. The other calls (to `ksqr` and `ksos`) are to continuation procedures introduced by CPS conversion. These model the return-from-subroutine idiom in assembly code.

```
(let* ((sqr_silkcps (lambda (x ksqr)
                      (let ((t1 (@* x x)))
                        (call ksqr t1))))
       (sos_silkcps (lambda (a b ksos)
                      (let ((k1 (lambda (asqr)
                                  (let ((k2 (lambda (bsqr)
                                              (let ((t2 (@+ asqr bsqr)))
                                                (call ksos t2)))))
                                    (call sqr_silkcps b k2)))))
                        (call sqr_silkcps a k1)))))
  (call sos_silkcps 3 4 klet*))
```

Figure 17.26: A CPS version of E_{sos} expressed in SILK$_{cps}$.

17.9.2 A Simple CPS Transform

CPS conversion is a meaning-preserving transformation that converts every procedure call in program into a tail call. In the TORTOISE compiler, CPS conversion has the following specification:

Preconditions: The input to CPS conversion is valid, uniquely named SILK program.

Postconditions: The output of CPS conversion is a valid, uniquely named SILK_{cps} program.

Other properties: If the input program is assignment-free, so is the output program.

In this section, we present the first of two CPS transformations that we will study. The first transformation, which we call $\mathcal{SCPS}$ (for *Simple* CPS conversion) is easier to explain, but generates code that is much less efficient than that produced by the second transformation.

The $\mathcal{SCPS}$ transformation is defined in Figures 17.27 and 17.28. The heart of the transformation is $\mathcal{SCPS}_{exp}$, which transforms expressions into CPS form. $\mathcal{SCPS}_{exp}$ transforms any given expression E to an abstraction `(lambda (`I_k`)` E'`)` that expects as its argument I_k an explicit continuation for E and eventually calls this continuation on the value of E in E'. This explicit continuation is immediately invoked to "return" the values of literals, identifiers, and abstractions. Each abstraction is transformed to take as a new additional final parameter a continuation I_{kcall} that is passed as the explicit continuation to its transformed body. Because the grammar of SILK_{cps} does not allow abstractions to appear directly as `call` arguments, it is also necessary to name the transformed abstraction in a `let` via a fresh identifier I_{abs}.

In the transformation of a `call` expression `(call` E_0 E_1 ... E_n`)`, explicit continuations are used to specify that the rator E_0 and rands E_1 ... E_n are evaluated in left to right order before the invocation takes place. The fresh variables I_0 ... I_n are introduced to name the values of each subexpression. Since every procedure has been transformed to expect an explicit continuation as its final argument, the transformed `call` must supply its continuation I_k as the final rand. The `let` transformation is similar, except that the `let`-bound names are used in place of fresh names for naming the values of the definition expressions. The unique naming requirement on input programs to $\mathcal{SCPS}$ guarantees that no variable capture can take place in the `let` transformation (see Exercise 17.17).

The transformation of `primop` expressions is similar to that for `call` and `let`. The syntactic constraints of SILK_{cps} require that a fresh variable (here named

$\mathcal{SCPS}_{prog} : \text{Program}_{Silk} \to \text{Program}_{cps}$

$\mathcal{SCPS}_{prog}[\![\texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] =$
```
  (silk (I1 ... In Iktop) ; Iktop fresh
    (let ((Ibody (SCPS_exp[[Ebody]])) ; Ibody fresh
      (call Ibody Iktop)))
```

$\mathcal{SCPS}_{exp} : \text{Exp}_{Silk} \to \text{Exp}_{cps}$

$\mathcal{SCPS}_{exp}[\![L]\!] = \texttt{(lambda (}I_k\texttt{) (call }I_k\ L\texttt{))}\ \texttt{;}\ I_k\ \texttt{fresh}$

$\mathcal{SCPS}_{exp}[\![I]\!] = \texttt{(lambda (}I_k\texttt{) (call }I_k\ I\texttt{))}\ \texttt{;}\ I_k\ \texttt{fresh}$

$\mathcal{SCPS}_{exp}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] =$
```
  (lambda (Ik) ; Ik fresh
    (let ((Iabs ; Iabs fresh
           (lambda (I1 ... In Ikcall) ; Ikcall fresh
             (call (SCPS_exp[[Ebody]]) Ikcall))))
      (call Ik Iabs)))
```

$\mathcal{SCPS}_{exp}[\![\texttt{(call }E_0\ \ldots\ E_n\texttt{)}]\!] =$
```
  (lambda (Ik) ; Ik fresh
    (call (SCPS_exp[[E0]])
          (lambda (I0) ; I0 fresh
                ...
                (call (SCPS_exp[[En]])
                      (lambda (In) ; In fresh
                        (call I0 ... In Ik))) ...)))
```

$\mathcal{SCPS}_{exp}[\![\texttt{(let ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}]\!] =$
```
  (lambda (Ik) ; Ik fresh
    (call (SCPS_exp[[E1]])
          (lambda (I1)
                ...
                (call (SCPS_exp[[En]])
                      (lambda (In)
                        (call (SCPS_exp[[Ebody]]) Ik))))))
```

Figure 17.27: A simple CPS transform, part 1.

$\mathcal{SCPS}_{exp}[\![$`(primop` O E_1 ... E_n`)`$]\!] =$
```
  (lambda (I_k) ; I_k fresh
    (call (SCPS_exp[[E_1]])
          (lambda (I_1) ; I_1 fresh
                ...
                 (call (SCPS_exp[[E_n]])
                       (lambda (I_n) ; I_n fresh
                         (let ((I_ans (primop O I_1 ... I_n))) ; I_ans fresh
                           (call I_k I_ans)))) ...)))
```

$\mathcal{SCPS}_{exp}[\![$`(cycrec ((`I_1 BV_1`) ... (`I_n BV_n`)) `E_{body}`)`$]\!] =$
```
  (lambda (I_k) ; I_k fresh
    (cycrec ((I_1 (SCPS_bv[[BV_1]]))
                  :
             (I_n (SCPS_bv[[BV_n]])))
      (call (SCPS_exp[[E_body]]) I_k)))
```

$\mathcal{SCPS}_{exp}[\![$`(set!` I_{lhs} E_{rhs}`)`$]\!] =$
```
  (lambda (I_k) ; I_k fresh
    (call (SCPS_exp[[E_rhs]])
          (lambda (I_rhs) ; I_rhs fresh
            (let ((I_ans (set! I_lhs I_rhs))) ; I_ans fresh
              (call I_k I_ans)))))
```

$\mathcal{SCPS}_{exp}[\![$`(if` E_{test} E_{then} E_{else}`)`$]\!] =$
```
  (lambda (I_k) ; I_k fresh
    (call (SCPS_exp[[E_test]])
          (lambda (I_test) ; I_test fresh
            (if I_test
                (call (SCPS_exp[[E_then]]) I_k)
                (call (SCPS_exp[[E_else]]) I_k)))))
```

$\mathcal{SCPS}_{exp}[\![$`(error` I_{msg}`)`$]\!] =$ `(lambda (`I_k`) (error` I_{msg}`)) ; `I_k `fresh`

$\mathcal{SCPS}_{bv} : \text{BindingValue}_{Silk} \rightarrow \text{BindingValue}_{cps}$

$\mathcal{SCPS}_{bv}[\![L]\!] = L$

$\mathcal{SCPS}_{bv}[\![$`(@mprod` DV_1 ... DV_n`)`$]\!] =$ `(@mprod` $\mathcal{SCPS}_{dv}[\![DV_1]\!]$ ... $\mathcal{SCPS}_{dv}[\![DV_n]\!]$`)`

$\mathcal{SCPS}_{bv}[\![$`(lambda (`I_1 ... I_n`) `E_{body}`)`$]\!]$
```
  (lambda (I_1 ... I_n I_kcall) ; I_kcall fresh
    (call (SCPS_exp[[E_body]]) I_kcall))
```

$\mathcal{SCPS}_{dv} : \text{DataValue}_{Silk} \rightarrow \text{DataValue}_{cps}$

$\mathcal{SCPS}_{dv}[\![V]\!] = V$

$\mathcal{SCPS}_{dv}[\![AB]\!] = \mathcal{SCPS}_{bv}[\![AB]\!]$

Figure 17.28: A simple CPS transform, part 2.

I_{ans}) be introduced to name the results of these expressions before passing them to the continuation. A similar `let` binding is needed in the `set!` transformation. The transformation of `cycrec` uses $\mathcal{SCPS}_{bv}$ to transform binding value expressions. This function acts as the identity on literals and mutable tuple creation forms, but transforms abstractions to take an extra continuation parameter.

In a transformed `if` expression, a fresh name I_{test} names the result of the test expression and the same continuation I_k is supplied to both transformed branches. This is the only place in $\mathcal{SCPS}$ where the explicit continuation I_k is referenced more than once in the transformed expression. The transformed `error` construct is the only place where the continuation is never referenced. All other constructs use I_k in a linear fashion — i.e., they reference it exactly once. This makes intuitive sense for regular control flow, which has only one possible "path" out of every expression other than `if` and `error`. Even in the `if` case, only one branch can be taken in a dynamic execution even though the the continuation is mentioned twice. Later we will study how CPS conversion exposes the non-linear nature of some sophisticated control features.

SILK programs are converted to CPS form by $\mathcal{SCPS}_{prog}$, which adds an additional parameter I_{ktop} that is an explicit top-level continuation for the program. It is assumed that the mechanism for program invocation will supply an appropriate procedure for this argument. For example, an operating system might construct a top-level continuation that displays the result of the program on the standard output stream or in a window within a graphical user interface.

The clauses for $\mathcal{SCPS}_{exp}$ contain numerous instances of the pattern

`(call (`$\mathcal{SCPS}_{exp}[\![E_1]\!]$`) `$E_2$`)`,

where E_2 is an abstraction or variable reference. But $\mathcal{SCPS}_{exp}$ is guaranteed to return a `lambda` expression, and the SILK$_{cps}$ grammar does not allow any subexpression of a `call` to be a `lambda`. Doesn't this yield an illegal SILK$_{cps}$ expression? The result of $\mathcal{SCPS}_{exp}$ would be illegal if were not for the [*implicit-let*] simplification, which transforms every `call` of the form

`(call (lambda (`I_k`) `$E_1{}'$`) `E_2`)`

into to the expression

`(let ((`I_k E_2`)) `$E_1{}'$`)`.

Since the grammar for letable expressions LE permits definition expressions that are variables and abstractions, the result of $\mathcal{SCPS}_{exp}$ is guaranteed to be a legal SILK$_{cps}$ expression. Note that when E_2 is a variable the [*copy-prop*] simplification will also be performed. This simplification is always valid in the CPS stage of the TORTOISE compiler, because the input and output of CPS

conversion are guaranteed to be assignment-free.

As a simple example of $\mathcal{SCPS}$, consider the CPS conversion of the incrementing program P_{inc} = `(silk (a) (@+ a 1))`. Before any simplifications are performed, $\mathcal{SCPS}_{prog}[\![P_{inc}]\!]$ yields

```
(silk (a ktop.0)
  (call (lambda (k.2)
          (call (lambda (k.6)
                  (call k.6 a))
                (lambda (v.3)
                  (call (lambda (k.5)
                          (call k.5 1))
                        (lambda (v.4)
                          (let ((ans.1 (@+ v.3 v.4)))
                            (call k.2 ans.1)))))))
        ktop.0)).
```

Four applications of [*implicit-let*] simplify this code to

```
(silk (a ktop.0)
  (let ((k.2 ktop.0))
    (let ((k.6 (lambda (v.3)
                 (let ((k.5 (lambda (v.4)
                              (let ((ans.1 (@+ v.3 v.4)))
                                (call k.2 ans.1)))))
                   (call k.5 1)))))
      (call k.6 a)))).
```

A single [*copy-prop*] simplification replaces `k.2` by `k.top` to yield the final result $P_{inc}{}'$:

```
(silk (a ktop.0)
  (let ((k.6 (lambda (v.3)
               (let ((k.5 (lambda (v.4)
                            (let ((ans.1 (@+ v.3 v.4)))
                              (call ktop.0 ans.1)))))
                 (call k.5 1)))))
    (call k.6 a))).
```

You should verify that $P_{inc}{}'$ is a legal SILK$_{cps}$ program. The convoluted nature of $P_{inc}{}'$ makes it a bit tricky to read. Here is one way to "pronounce" this program:

> The program is given an input `a` and top-level continuation `ktop.0`. First evaluate `a` and pass its value to continuation `k.6`, which gives it the name `v.3`. Then evaluate `1` and pass it to continuation `k.5`,

> which gives it the name `v.4`. Next, calculate the sum of `v.3` and `v.4` and name the result `ans.1`. Finally, return this answer as the result of the program by invoking `ktop.0` on `ans.1`.

This seems like an awful lot of work to increment a number! Even though the [*implicit-let*] and [*copy-prop*] rules have simplified the program, it could still be simpler. In particular, the continuations `k.5` and `k.6` merely rename the values of `a` and `1` to `v.3` and `v.4`, which is unnecessary.

In larger programs, the extent of these undesirable inefficiencies becomes more apparent. For example, Figure 17.29 shows the result of using $\mathcal{SCPS}$ to transform a numerical program P_{quad} with several nested subexpressions. Try to "pronounce" the transformed program as illustrated above. Along the way you will notice numerous unnecessary continuations and renamings. The result of performing $\mathcal{SCPS}$ on our running `revmap` example is so large that it would require several pages to display. The `revmap` program has an abstract syntax tree with 46 nodes; transforming it with $\mathcal{SCPS}_{prog}$ yields a result with 230 nodes. And this is after simplification — the unsimplified transformed program has 317 nodes!

Can anything be done to automatically eliminate the inefficiences introduced by $\mathcal{SCPS}$? Yes. It is possible to define additional simplification rules that will make the CPS converted code much more reasonable. For example, in `(let ((`I E_{defn}`))` E_{body}`)`, if E_{defn} is a literal or abstraction, it is possible to replace the `let` by the substitution of E_{defn} for I in E_{body}. This simplification is traditionally called **constant propagation** and (when followed by [*implicit-let*]) is called **inlining** for abstractions. For example, two applications of inlining on $P_{inc}{}'$ yield

```
(silk (a ktop.0)
  (let ((v.3 a))
    (let ((v.4 1))
      (let ((ans.1 (@+ v.3 v.4)))
        (call ktop.0 ans.1)))))
```

and then copy propagation and constant propagation simplify the program to

```
(silk (a ktop.0)
  (let ((ans.1 (@+ a 1)))
    (call ktop.0 ans.1))).
```

Performing these additional simplifications on $P_{quad}{}'$ gives the following *much* improved CPS code:

P_{quad} = `(silk (a b c) (@+ (@- 0 (@* b b)) (@* 4 (@* a c))))`

$\mathcal{SCPS}_{prog}[\![P_{quad}]\!]$ = P_{quad}', where P_{quad}' =

```
(silk (a b c ktop.0)
 (let* ((k.17
         (lambda (v.3)
           (let* ((k.6
                   (lambda (v.4)
                     (let ((ans.1 (@+ v.3 v.4)))
                       (call ktop.0 ans.1))))
                  (k.15
                   (lambda (v.7)
                     (let* ((k.10 (lambda (v.8)
                                    (let ((ans.5 (@* v.7 v.8)))
                                      (call k.6 ans.5))))
                            (k.14
                             (lambda (v.11)
                               (let ((k.13
                                      (lambda (v.12)
                                        (let ((ans.9 (@* v.11 v.12)))
                                          (call k.10 ans.9)))))
                                 (call k.13 c)))))
                       (call k.14 a)))))
             (call k.15 4))))
        (k.26
         (lambda (v.18)
           (let* ((k.21 (lambda (v.19)
                          (let ((ans.16 (@- v.18 v.19)))
                            (call k.17 ans.16))))
                  (k.25 (lambda (v.22)
                          (let ((k.24 (lambda (v.23)
                                        (let ((ans.20 (@* v.22 v.23)))
                                          (call k.21 ans.20)))))
                            (call k.24 b)))))
             (call k.25 b)))))
   (call k.26 0)))
```

Figure 17.29: Simple CPS conversion of a numeric program.

```
(silk (a b c ktop.0)
  (let* ((ans.20 (@* b b))
         (ans.16 (@- 0 ans.20))
         (ans.9 (@* a c))
         (ans.5 (@* 4 ans.9))
         (ans.1 (@* ans.16 ans.5)))
    (call ktop.0 ans.1))).
```

These examples underscore the inefficiency of the code generated by $\mathcal{SCPS}$.

Why don't we just modify SILK to include the constant propagation and inlining simplifications? Constant propagation is not problematic, but inlining is a delicate transformation. In SILK_{cps}, it is only legal to copy an abstraction to certain positions (such as the rator of a `call`, where it can be removed via [*implicit-let*]). When a named abstraction is used more than once in the body of a `let`, copying the abstraction multiple times makes the program bigger. Unrestricted inlining can lead to **code bloat**, a dramatic increase in the size of a program. In the presence of recursive procedures, special care must often be taken to avoid infinitely unwinding a recursive definition via inlining. Since we intend that SILK simplifications should be straightforward to implement, we prefer not to include inlining as a simplification. Inlining issues are further explored in Exercise 17.19.

Does that mean we are stuck with an inefficient CPS transformation? No. In the next section, we study a cleverer approach to CPS conversion that avoids generating unnecessary code in the first place.

▷ **Exercise 17.15** Consider the SILK program

$P =$ `(silk (x y) (@* (@+ x y) (@- x y)))`.

a. Show the result P_1 generated by $\mathcal{SCPS}_{prog}[\![P]\!]$ without performing any simplifications.

b. Show the result P_2 of simplifying P_1 using the standard SILK simplifications (including [*implicit-let*] and [*copy-prop*]).

c. Show the result P_3 of further simplifying P_2 using inlining in addition to the standard SILK simplifications. ◁

▷ **Exercise 17.16**

a. Suppose that `(begin` E^*`)`, `(scand` E^*`)`, and `(scor` E^*`)` were not syntactic sugar but a kernel SILK constructs. Give the $\mathcal{SCPS}_{exp}$ clauses for `begin`, `scand`, and `scor`.

b. Suppose that SILK were extended with FL's `cond` construct (as a kernel form, not sugar). Give the $\mathcal{SCPS}_{exp}$ clause for `cond`. ◁

▷ **Exercise 17.17**

a. Give a concrete example of how variable capture can take place in the `let` clause of $\mathcal{SCPS}_{exp}$ if the initial program is not uniquely named.

b. Modify the `let` clause of $\mathcal{SCPS}_{exp}$ so that it works properly even if the initial program is not uniquely named ◁

▷ **Exercise 17.18** Control branches in linear assembly language code are usually provided via branch instructions that perform a control jump if a certain condition holds but "drop through" to the next instruction if the condition does not hold. We can model branch instructions in SILK$_{cps}$ by restricting `if` expressions to have the form

`(if` V_{cps} `(call` V_{cps} V_{cps}*`)` E_{cps}`)`.

Modify the $\mathcal{SCPS}_{exp}$ clause for `if` so that all transformed `if`s have this restricted form. ◁

▷ **Exercise 17.19** Consider the following [*copy-abs*] simplification rule:

$$\texttt{(let ((}I\ AB\texttt{))}\ E_{body}\texttt{)} \xrightarrow{simp} [AB/I]E_{body} \qquad [\textit{copy-abs}]$$

Together, [*copy-abs*] and the standard SILK [*implicit-let*] and [*copy-prop*] rules implement a form of procedure inlining. For example

```
(let ((inc (lambda (x) (@+ x 1))))
  (@* (call inc a) (call inc b)))
```

can be simplified via [*copy-abs*] to

```
(@* (call (lambda (x) (@+ x 1)) a)
    (call (lambda (x) (@+ x 1)) b)).
```

Two applications of [*implicit-let*] give

```
(@* (let ((x a)) (@+ x 1))
    (let ((x b)) (@+ x 1))),
```

and two applications of [*copy-prop*] yield the inlined code

```
(@* (@+ a 1) (@+ b 1)).
```

In this exercise, we explore some issues with inlining.

a. Use inlining to remove all calls to `sqr` in the following SILK expression. How many multiplications does the resulting expression contain?

```
(let ((sqr (lambda (x) (@* x x))))
  (call sqr (call sqr (call sqr a))))
```

b. Use inlining to remove all calls to `sqr`, `quad`, and `oct` in the following SILK expression. How many multiplications does the resulting expression contain?

```
(let* ((sqr (lambda (x) (@* x x)))
       (quad (lambda (y) (@* (call sqr y) (call sqr y))))
       (oct (lambda (z) (@* (call quad z) (call quad z)))))
  (@* (call oct a) (call oct b)))
```

c. What happens if inlining is used to simplify the following SILK expression?

```
(let ((f (lambda (g) (call g g))))
  (call f f))
```

(For the purposes of this part, ignore the SILK type system.)

d. Using only standard SILK simplifications, the result of $\mathcal{SCPS}_{prog}$ is guaranteed to be uniquely named if the input is uniquely named. This property does not hold in the presence of inlining. Write an example program P_{nun} such that the result of simplifying $\mathcal{SCPS}_{prog}[\![P_{nun}]\!]$ via inlining is not uniquely named. *Hint:* Where can duplication occur in a CPS converted program?

e. Inlining multiple copies of an abstraction can lead to code bloat. Develop an example SILK P_{bloat} where performing inlining on the result of $\mathcal{SCPS}_{prog}[\![P_{bloat}]\!]$ yields a *larger* transformed program rather than a smaller one. *Hint:* Where can duplication occur in a CPS converted program? ◁

17.9.3 A More Efficient CPS Transform

Reconsider the output of $\mathcal{SCPS}$ on the incrementing program `(silk (a) (@+ a 1))`:

```
(silk (a ktop.0)
  (let ((k.6 (lambda (v.3)
               (let ((k.5 (lambda (v.4)
                            (let ((ans.1 (@+ v.3 v.4)))
                              (call ktop.0 ans.1)))))
                 (call k.5 1)))))
    (call k.6 a)).
```

In the above code, we have used gray to highlight the inefficient portions of the code that we wish to eliminate. These are exactly the portions we were able to eliminate via extra simplifications like inlining and constant propagation in the previous section. Our goal in developing a more efficient CPS transform is to perform these simplifications as part of CPS conversion itself rather than waiting to do them later. Instead of sweeping away unsightly gray code as an afterthought, we want to simply avoid generating it in the first place!

The key insight is that we can avoid generating the gray code if we somehow make it part of metalanguage that specifies the CPS conversion algorithm. Suppose we change the gray SILK `let`s, `lambda`s, and `call`s to metalanguage **let**s, procs and applications. Then our example would become:

```
(silk (a ktop.0)
  let k6 be (λ V3 .
              let k5 be (λ V4 .
                          (let ((ans.1 (@+ V3 V4)))
                            (call ktop.0 ans.1)))
              in (k5 1))
   in (k6 a))
```

To enhance readability, we will keep the metalanguage notation in gray and the SILK code in black teletype font. Note that k_5 and k_6 name metalanguage functions whose parameters (V_3 and V_4) must be pieces of SILK syntax — in particular, SILK value expressions. Indeed, k_5 is applied to the SILK literal `1` and k_6 is applied to the SILK literal `a`. The result of evaluating the gray metalanguage expressions in our example yields

```
(silk (a ktop.0)
  (let ((ans.1 (@+ a 1)))
    (call ktop.0 ans.1))),
```

which is exactly the simplified result we want!

What we have done is taken computation that would have been performed when executing the code generated by CPS conversion and moved it so that it is performed when the code is generated. The output of CPS conversion can now be viewed as code that is executed in two stages: the gray code is the code that can be executed immediately, while the black code is the **residual code** that can only be executed later. This notion of **staged computation** is a key idea in an approach to optimization known as **partial evaluation**. By expressing the gray code in the metalanguage, it gets executed "for free" as part of the CPS translation itself.

Our improved approach to CPS conversion will make heavy use of gray abstractions of the form $(\lambda V \ldots)$ that map SILK$_{cps}$ value expressions (i.e., literals and variable references) to other SILK$_{cps}$ expressions. Because these abstractions play the role of continuations at the metalanguage level, we call them **meta-continuations**. In the above example, k_5 and k_6 are examples of meta-continuations.

A meta-continuation can be viewed as a metalanguage representation of a special kind of context: a SILK$_{cps}$ expression with named holes that can be filled only with SILK$_{cps}$ value expresssions. Such contexts may contain more than one hole, but a hole with a given name can appear only once. For example, here are meta-continuations that arise in the CPS conversion of the incrementing example:

Context Notation	Metalanguage Notation
`(call ktop.0` $\square_1$`)`	λV_1 . `(call ktop.0` V_1`)`
`(let ((ans.1 (@+` $\square_3$ $\square_4$`)))` `(call ktop.0 ans.1))`	λV_4 . `(let ((ans.1 (@+` V_3 V_4`)))` `(call ktop.0 ans.1))`
`(let ((ans.1 (@+` $\square_3$ `1)))` `(call ktop.0 ans.1))`	λV_3 . `(let ((ans.1 (@+` V_3 `1)))` `(call ktop.0 ans.1))`

Figures 17.30 and 17.31 present an efficient version of CPS conversion that is based on the notions of staged computation and meta-continuations. The metavariable m ranges over meta-continuations in the domain *MetaCont*, which consists of functions that map Silk$_{cps}$ value expressions to Silk$_{cps}$ expressions. The $mc{\rightarrow}exp$ and $exp{\rightarrow}mc$ functions perform conversions between compile-time meta-continuations and Silk$_{cps}$ expressions denoting run-time continuations.

The CPS conversion clauses in Figures 17.30 and 17.31 are similar to the ones in Figures 17.27 and 17.28. Indeed, the former are obtained from the latter by:

- transforming every continuation-accepting Silk$_{cps}$ abstraction of the form `(lambda (`I_k`) ...)` into a metalanguage abstraction of the form $(\lambda m.\ \ldots)$;
- transforming every Silk$_{cps}$ continuation of the form `(lambda (`I`) ...)` into a meta-continuation of the form $(\lambda V.\ \ldots)$;
- transforming every Silk$_{cps}$ application `(call` E_k V`)` in which E_k is a continuation (either an abstraction or a variable) to a **meta-application** of the form $(m\ V)$, where m is the meta-continuation that corresponds to E_k.
- using the $mc{\rightarrow}exp$ and $exp{\rightarrow}mc$ functions where necessry to ensure that all the types work out.

The key benefit of the meta-continuation approach to CPS conversion is that many reductions that would be left as residual run-time code in the simple approach are guaranteed to be performed at compile-time in the metalanguage. We illustrate this in Figure 17.32 by showing the CPS conversion of the expression `(call f (@* x (if (call g y) 2 3)))` relative to an initial continuation named `k`. In the figure, each meta-application of the form

$$((\lambda m\ .\ \mathbb{E}\{(m\ V_{actual})\})\ (\lambda V_{formal}\ .\ E))$$

(where $\mathbb{E}$ is a Silk$_{cps}$ expression context with one hole) is reduced to

$$\mathbb{E}\{[V_{actual}/V_{formal}]E\}$$

$E_{cps} \in \text{Exp}_{cps}$
$V_{cps} \in \text{ValueExp}_{cps}$
$m \in \mathit{MetaCont} = \text{ValueExp}_{cps} \rightarrow \text{Exp}_{cps}$

$mc{\rightarrow}exp : \mathit{MetaCont} \rightarrow \text{Exp}_{cps} = (\lambda m\ .\ \texttt{(lambda (}I_{temp}\texttt{)}\ (m\ I_{temp}))$
$exp{\rightarrow}mc : \text{Exp}_{cps} \rightarrow \mathit{MetaCont} = (\lambda E_{cps}\ .\ (\lambda V_{cps}\ .\ \texttt{(call}\ E_{cps}\ V_{cps}\texttt{)}))$

$\mathcal{MCPS}_{prog} : \text{Program}_{Silk} \rightarrow \text{Program}_{cps}$

$\mathcal{MCPS}_{prog}[\![\texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] =$
$\quad\texttt{(silk (}I_1\ \ldots\ I_n\ I_{ktop}\texttt{)\ ;}\ I_{ktop}\ \texttt{fresh}$
$\quad\quad(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ (exp{\rightarrow}mc\ I_{ktop}))\texttt{)}$

$\mathcal{MCPS}_{exp} : \text{Exp}_{Silk} \rightarrow \mathit{MetaCont} \rightarrow \text{Exp}_{cps}$

$\mathcal{MCPS}_{exp}[\![L]\!] = (\lambda m\ .\ (m\ L))$

$\mathcal{MCPS}_{exp}[\![I]\!] = (\lambda m\ .\ (m\ I))$

$\mathcal{MCPS}_{exp}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] =$
$\quad(\lambda m\ .\ \texttt{(let ((}I_{abs}\ \texttt{;}\ I_{abs}\ \texttt{fresh}$
$\quad\quad\quad\texttt{(lambda (}I_1\ \ldots\ I_n\ I_{kcall}\texttt{)\ ;}\ I_{kcall}\ \texttt{fresh}$
$\quad\quad\quad\quad(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ (exp{\rightarrow}mc\ I_{kcall}))))$
$\quad\quad(m\ I_{abs}))\texttt{)})$

$\mathcal{MCPS}_{exp}[\![\texttt{(call}\ E_0\ \ldots\ E_n\texttt{)}]\!] =$
$\quad(\lambda m\ .\ (\mathcal{MCPS}_{exp}[\![E_0]\!]$
$\quad\quad(\lambda V_0\ .$
$\quad\quad\quad\ldots$
$\quad\quad\quad\quad(\mathcal{MCPS}_{exp}[\![E_n]\!]$
$\quad\quad\quad\quad\quad(\lambda V_n\ .\ \texttt{(call}\ V_0\ \ldots\ V_n\ (mc{\rightarrow}exp\ m)\texttt{)})\ \ldots)))$

$\mathcal{MCPS}_{exp}[\![\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}]\!] =$
$\quad(\lambda m\ .\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
$\quad\quad(\lambda V_1\ .$
$\quad\quad\quad\ldots$
$\quad\quad\quad\quad(\mathcal{MCPS}_{exp}[\![E_n]\!]$
$\quad\quad\quad\quad\quad(\lambda V_n\ .\ \texttt{(let ((}I_1\ V_1\texttt{)}\ \ldots\ \texttt{(}I_n\ V_n\texttt{))}$
$\quad\quad\quad\quad\quad\quad\quad(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ m)\texttt{)}))\ \ldots\ )))$

Figure 17.30: An efficient CPS transform based on meta-continuations, part 1.

$$\mathcal{MCPS}_{exp}[\![\texttt{(primop } O\ E_1\ \ldots\ E_n\texttt{)}]\!] =$$
$$(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$$
$$(\lambda V_1\,.$$
$$\ddots$$
$$(\mathcal{MCPS}_{exp}[\![E_n]\!]$$
$$(\lambda V_n\,.\,\texttt{(let ((}I_{ans}\ \texttt{(primop } O\ V_1\ \ldots\ V_n\texttt{))) ; } I_{ans}\ \texttt{fresh}$$
$$(m\ I_{ans}))\ \ldots\ )))$$

$$\mathcal{MCPS}_{exp}[\![\texttt{(cycrec ((}I_1\ BV_1\texttt{) } \ldots\ \texttt{(}I_n\ BV_n\texttt{)) } E_{body}\texttt{)}]\!] =$$
$$(\lambda m\,.\,\texttt{(cycrec ((}I_1\ (\mathcal{MCPS}_{bv}[\![BV_1]\!])\texttt{)}$$
$$\vdots$$
$$\texttt{(}I_n\ (\mathcal{MCPS}_{bv}[\![BV_n]\!])\texttt{)))}$$
$$(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ m))$$

$$\mathcal{MCPS}_{exp}[\![\texttt{(set! } I_{lhs}\ E_{rhs}\texttt{)}]\!] =$$
$$(\lambda m\,.\,(\mathcal{MCPS}_{exp}[\![E_{rhs}]\!]$$
$$(\lambda V_{rhs}\,.\ \texttt{(let ((}I_{ans}\ \texttt{(set! } I_{lhs}\ V_{rhs}\texttt{))) ; } I_{ans}\ \texttt{fresh}$$
$$(m\ I_{ans}))))$$

$$\mathcal{MCPS}_{exp}[\![\texttt{(if } E_{test}\ E_{then}\ E_{else}\texttt{)}]\!] =$$
$$(\lambda m\,.\,(\mathcal{MCPS}_{exp}[\![E_{test}]\!]$$
$$(\lambda V_{test}\,.\ \texttt{(let ((}I_{kif}\ (\text{mc}\rightarrow\text{exp}\ m)\texttt{))}$$
$$\texttt{(if } V_{test}$$
$$(\mathcal{MCPS}_{exp}[\![E_{then}]\!]\ (\text{exp}\rightarrow\text{mc}\ I_{kif}))$$
$$(\mathcal{MCPS}_{exp}[\![E_{else}]\!]\ (\text{exp}\rightarrow\text{mc}\ I_{kif}))))))$$

$$\mathcal{MCPS}_{exp}[\![\texttt{(error } I_{msg}\texttt{)}]\!] = (\lambda m\,.\ \texttt{(error } I_{msg}\texttt{)})$$

$$\mathcal{SCPS}_{bv} : \text{BindingValue}_{Silk} \rightarrow \text{BindingValue}_{cps}$$

$$\mathcal{MCPS}_{bv}[\![L]\!] = L$$

$$\mathcal{MCPS}_{bv}[\![\texttt{(@mprod } V_1\ \ldots\ V_n\texttt{)}]\!] = \texttt{(@mprod } V_1\ \ldots\ V_n\texttt{) ; unchanged}$$

$$\mathcal{MCPS}_{bv}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) } E_{body}\texttt{)}]\!] =$$
$$\texttt{(lambda (}I_1\ \ldots\ I_n\ I_{kcall}\texttt{) ; } I_{kcall}\ \texttt{fresh}$$
$$(\mathcal{MCPS}_{exp}[\![E]\!]\ (\text{exp}\rightarrow\text{mc}\ I_{kcall}))\texttt{)}$$

$$\mathcal{MCPS}_{dv} : \text{DataValue}_{Silk} \rightarrow \text{DataValue}_{cps}$$

$$\mathcal{MCPS}_{dv}[\![V]\!] = V$$

$$\mathcal{MCPS}_{dv}[\![AB]\!] = \mathcal{SCPS}_{bv}[\![AB]\!]$$

Figure 17.31: An efficient CPS transform based on meta-continuations, part 2.

and each meta-application of the form $(\mathcal{MCPS}_{exp}[\![V_{actual}]\!]\ (\lambda V_{formal}\,.\,E))$ is reduced to $[V_{actual}/V_{formal}]E$. Each of these reductions removes a potential run-time application that might remain after simple CPS conversion.

The example illustrates how $\mathcal{MCPS}$ effectively turns the input expression "inside out". In the input expression, the call to `f` is the outermost call, and `(call g y)` is the innermost call. But in the CPS-converted result, the call to `g` is the outermost call and the call to `f` is nested deep inside. This reorganization is necessary to make explicit the order in which operations are performed:

1. `g` is applied to `y`;
2. the result of the `g` application (call it `t.4`) is tested by `if`;
3. the test determines which of `2` or `3` (call it `t.3`) is mulitplied by `x`;
4. `f` is invoked on the result of the multiplication (call it `ans.1`);
5. the result of the `f` application is supplied to the continuation `k`.

Variables such as `ans.1`, `t.3`, and `t.4` can be viewed as denoting temporary registers.

Note that $(mc{\rightarrow}exp\ (exp{\rightarrow}mc\ \texttt{k}))$ is simplified to `k` in our example. To see why, observe that

$$\begin{aligned}
&(mc{\rightarrow}exp\ (exp{\rightarrow}mc\ \texttt{k}))\\
&= (\ (\lambda m\,.\ \texttt{(lambda (}I_{temp}\texttt{)}\ (m\ I_{temp}))\ (\lambda V\,.\,\texttt{(call k }V\texttt{)}))\\
&= \texttt{(lambda (}I_{temp}\texttt{)}\ ((\lambda V\,.\,\texttt{(call k }V\texttt{)})\ I_{temp}\texttt{)})\\
&= \texttt{(lambda (}I_{temp}\texttt{) (call k }I_{temp}\texttt{))}.
\end{aligned}$$

The final expression is simplified to `k` by the [*eta-lambda*] rule. This eta-reduction eliminates a `call` in cases where the CPS transform would have generated a continuation that simply passed its argument along to another continuation with no additional processing. This simplification is sometimes called the **tail call optimization** because it guarantees that tail calls in the source program require no additional control storage in the compiled program; they can be viewed as assembly code jumps that pass arguments. Languages are said to be **properly tail recursive** if implementations are required to compile source tail calls into jumps. Our SILK mini-language is properly tail recursive, as is the real language SCHEME. Such languages can leave out iteration constructs (like `while` and `for` loops) and still have programs with iterative behavior.

Observe that in each clause of $\mathcal{MCPS}$, the meta-continuation m is referenced at most once. This guarantees that each meta-application makes the metalanguage expression smaller. Thus there is no specter of duplication-induced code bloat that haunts more general inlining optimizations.

```
(MCPS_exp⟦(call f (@* x (if (call g y) 2 3)))⟧ (exp→mc k))
 = ((λm . (MCPS_exp⟦f⟧ (λV1 . (MCPS_exp⟦(@* x (if (call g y) 2 3))⟧
                                (λV2 . (call V1 V2 (mc→exp m))))))))
    (exp→mc k))
 = (MCPS_exp⟦f⟧ (λV1 . (MCPS_exp⟦(@* x (if (call g y) 2 3))⟧
                          (λV2 . (call V1 V2 (mc→exp (exp→mc k)))))))
 = (MCPS_exp⟦(@* x (if (call g y) 2 3))⟧ (λV2 . (call f V2 k)))
 = ((λm . (MCPS_exp⟦x⟧
           (λV3 . (MCPS_exp⟦(if (call g y) 2 3)⟧
                    (λV4 . (let ((ans.1 (@* V3 V4))) (m ans.1))))))))
    (λV2 . (call f V2 k)))
 = (MCPS_exp⟦x⟧
    (λV3 . (MCPS_exp⟦(if (call g y) 2 3)⟧
             (λV4 . (let ((ans.1 (@* V3 V4))) (call f ans.1 k))))))
 = (MCPS_exp⟦(if (call g y) 2 3)⟧
    (λV4 . (let ((ans.1 (@* x V4))) (call f ans.1 k))))
 = ((λm . (MCPS_exp⟦(call g y)⟧
           (λV5 . (let ((kif.2 (mc→exp m)))
                    (if V5
                        (MCPS_exp⟦2⟧ (exp→mc kif.2))
                        (MCPS_exp⟦3⟧ (exp→mc kif.2))))))))
    (λV4 . (let ((ans.1 (@* x V4))) (call f ans.1 k))))
 = (MCPS_exp⟦(call g y)⟧
    (λV5 . (let ((kif.2 (lambda (t.3) (let ((ans.1 (@* x t.3)))
                                        (call f ans.1 k)))))
             (if V5
                 (MCPS_exp⟦2⟧ (λV6 . (call kif.2 V6)))
                 (MCPS_exp⟦3⟧ (λV7 . (call kif.2 V7)))))))
 = ((λm . (MCPS_exp⟦g⟧ (λV8 . )(MCPS_exp⟦y⟧ (λV9 . (call V8 V9 (mc→exp m)))))))
    (λV5 . (let ((kif.2 (lambda (t.3) (let ((ans.1 (@* x t.3)))
                                        (call f ans.1 k)))))
             (if V5 (call kif.2 2) (call kif.2 3)))))
 = (call g y (lambda (t.4)
                (let ((kif.2 (lambda (t.3)
                               (let ((ans.1 (@* x t.3)))
                                 (call f ans.1 k)))))
                  (if t.4 (call kif.2 2) (call kif.2 3)))))
```

Figure 17.32: An example of CPS conversion using meta-continuations.

Converting the meta-continuation to a SILK$_{cps}$ abstraction named I_{kif} in the `if` clause is essential for ensuring this guarantee. It is important to note that the I_{kif} abstraction does not destroy proper tail recursion. Consider the expression

```
(if (f x) (g y) (h z)).
```

The call to `f` is not a tail call, but the calls to `g` and `h` are tail calls. Without simplifications, the result of CPS converting this expression relative to an initial continuation `k` is

```
(call f x (lambda (t.3)
            (let ((kif.1 (lambda (t.2) (call k t.2))))
              (if t.3 (call g y kif.1) (call h z kif.1))))).
```

Fortunately, the standard simplifications implement proper tail recursion in this case. The [*eta-lambda*] simplification yields

```
(call f x (lambda (t.3)
            (let ((kif.1 k))
              (if t.3 (call g y kif.1) (call h z kif.1)))))
```

and the [*copy-prop*] simplification yields

```
(call f x (lambda (t.3) (if t.3 (call g y k) (call h z k)))).
```

Figure 17.33 shows the result of using $\mathcal{MCPS}$ to CPS convert our running `revmap` example. Observe that the output of CPS conversion looks quite a bit closer to assembly language code than the input. You should study the code to convince yourself that this program has the same behavior as the original program. CPS conversion has introduced only one non-trivial continuation abstraction: `k.38` names the continuation of the call to `f` in the body of the loop. Each input abstraction has been extended with a final argument naming its continuation: `abs.12` (this is just a renamed version of `revmap`) takes continuation argument `k.22`; the loop takes continuation argument `k.27`; and the greater-than-`b` procedure takes continuation `k.20`. Note that the `loop.8` procedure is invoked tail recursively within its body, so it requires only constant control space and is thus a true iteration construct like loops in traditional languages.

It is worth noting that the conciseness of the code in Figure 17.33 is a combination of the simplifications performed by reducing meta-applications at compile time *and* the standard SILK$_{cps}$ simplifications. To underscore the importance of the latter, Figure 17.34 shows the result of $\mathcal{MCPS}$ before any SILK$_{cps}$ simplifications are performed.

▷ **Exercise 17.20** Use $\mathcal{MCPS}_{exp}$ to CPS convert the following expressions relative to an initial meta-continuation (*exp*→*mc* `k`).

```
(silk (a.1 b.2 ktop.11)
 (let* ((abs.12
         (lambda (f.5 lst.6 k.22)
           (let* ((t.24 (@null))
                  (t.23 (@mprod t.24)))
             (cycrec
              ((loop.8
                (lambda (xs.9 k.27)
                  (let ((t.29 (@null? xs.9)))
                    (if t.29
                        (let ((t.39 (@mget 1 t.23)))
                          (call k.27 t.39))
                        (let* ((t.32 (@car xs.9))
                               (k.38 (lambda (t.33)
                                       (let* ((t.34 (@mget 1 t.23))
                                              (t.31 (@cons t.33 t.34))
                                              (t.30 (@mset! 1 t.23 t.31))
                                              (t.35 (@cdr xs.9)))
                                         (call loop.8 t.35 k.27)))))
                          (call f.5 t.32 k.38)))))))
               (call loop.8 lst.6 k.22)))))
        (abs.13
         (lambda (x.4 k.20)
           (let ((t.21 (@> x.4 b.2)))
             (call k.20 t.21))))
        (t.16 (@* a.1 7))
        (t.17 (@null))
        (t.15 (@cons t.16 t.17))
        (t.14 (@cons a.1 t.15)))
   (call abs.12 abs.13 t.14 ktop.11)))
```

Figure 17.33: Running example after CPS conversion (with simplifications).

```
(silk (a.1 b.2 ktop.11)
  (let* ((abs.12
          (lambda (f.5 lst.6 k.22)
            (let* ((t.24 (@null))
                   (t.23 (@mprod t.24))
                   (ans.7 t.23))
              (cycrec
               ((loop.8
                 (lambda (xs.9 k.27)
                   (let* ((kif.29 (lambda (t.28) (call k.27 t.28)))
                          (t.30 (@null? xs.9)))
                     (if t.30
                         (let ((t.40 (@mget 1 ans.7)))
                           (call kif.29 t.40))
                         (let* ((t.33 (@car xs.9))
                                (k.39
                                 (lambda (t.34)
                                   (let* ((t.35 (@mget 1 ans.7))
                                          (t.32 (@cons t.34 t.35))
                                          (t.31 (@mset! 1 ans.7 t.32))
                                          (ignore.10 t.31)
                                          (t.36 (@cdr xs.9))
                                          (k.38
                                           (lambda (t.37)
                                             (call kif.29 t.37))))
                                     (call loop.8 t.36 k.38)))))
                           (call f.5 t.33 k.39)))))))
                (let ((k.26 (lambda (t.25) (call k.22 t.25))))
                  (call loop.8 lst.6 k.26))))))
         (revmap.3 abs.12)
         (abs.13
          (lambda (x.4 k.20)
            (let ((t.21 (@> x.4 b.2)))
              (call k.20 t.21))))
         (t.16 (@* a.1 7))
         (t.17 (@null))
         (t.15 (@cons t.16 t.17))
         (t.14 (@cons a.1 t.15))
         (k.19 (lambda (t.18) (call ktop.11 t.18))))
    (call revmap.3 abs.13 t.14 k.19)))
```

Figure 17.34: Running example after CPS conversion (without simplifications).

a. `(lambda (f) (+ 1 (call f 2)))`

b. `(lambda (g x) (+ 1 (g x)))`

c. `(lambda (f g h y) (call f (call g x) (call h y)))`

d. `(lambda (f) (@* (if (f 1) 2 3) (if (f 4) 5 6)))` ◁

▷ **Exercise 17.21** Use $\mathcal{MCPS}_{prog}$ to CPS convert the following programs:

a. The program P_{quad} from Figure 17.29.

b.
```
(silk (x)
  (cycrec ((fact (lambda (n)
                   (if (@= n 0)
                       1
                       (@* n (call fact (@- n 1)))))))
    (call fact x)))
```

c.
```
(silk (x)
  (cycrec ((fib (lambda (n)
                  (if (@<= n 1)
                      n
                      (@+ (call fib (@- n 1))
                          (call fib (@- n 2)))))))
    (fib x)))
```
◁

▷ **Exercise 17.22** Do Exercise 17.16, giving $\mathcal{MCPS}_{exp}$ clauses instead of $\mathcal{SCPS}_{exp}$ clauses. ◁

▷ **Exercise 17.23** The unique naming prerequisite on programs is essential for the correctness of $\mathcal{MCPS}_{prog}[\![\,.\,]\!]$ To demonstrate this, show that the output of $\mathcal{MCPS}_{prog}[\![P_{mnun}]\!]$ has a different behavior from P_{mnun}, where P_{mnun} is:

```
(silk (a b)
  (@+ (let ((a (@* b b)))
        a)
      a))
```

◁

▷ **Exercise 17.24**

a. Show the result of using $\mathcal{MCPS}_{exp}[\![]\!]$ to convert the following program $P_{set!}$:

```
(silk (a b)
  (let ((ignore.0 (set! a (set! b (@+ a b)))))
    (@mprod a b))).
```

b. In the TORTOISE compiler, assignment conversion is performed before CPS conversion. Show the result of $\mathcal{MCPS}_{prog}[\![\mathcal{AC}_{prog}[\![P_{set!}]\!]]\!]$.

c. It is possible to perform assignment conversion *after* closure conversion. Show the result of $\mathcal{AC}_{prog}[\![\mathcal{MCPS}_{prog}[\![P_{set!}]\!]]\!]$. Is the result in CPS form?

d. Describe how to modify assignment conversion to guarantee that if its input is in CPS form then its output is in CPS form. ◁

▷ **Exercise 17.25** Bud Lojack thinks he can improve $\mathcal{MCPS}$ by extending meta-continuations to take letable expressions rather than just value expressions:

$$m \in \mathit{MetaCont} = \text{LetableExp}_{cps} \rightarrow \text{Exp}_{cps}$$

Recall (from Figure 17.25) that letable expressions include abstractions, primitive applications, and assignment expressions in addition to value expressions. Bud reasons that if meta-continuations are changed in this way, then he can call them directly on abstractions, primitive applications, and assignment expressions. Of course, it will also be necessary to wrap all letable expressions in `let`s, but Bud figures that SILK's syntactic simplifications will remove most unnecessary `let` bindings. Bud changes several $\mathcal{MCPS}_{exp}$ clauses as shown in Figure 17.35.

a. Show the result of using Bud's clauses to CPS convert the following expression relative to an initial continuation `k`:

```
(call f (lambda (a b) (@+ (@* a a) (@* b b))))
```

b. Bud proudly shows his new clauses to Abby Stracksen. Abby says "Your approach is interesting, but it has a major bug: it can change the meaning of programs by reordering side effects!" Show that Abby is right by giving simple programs involving `mprod` and `set!` in which Bud's CPS converter changes the meaning of the program. ◁

17.9.4 CPS Converting Control Constructs

[This section still needs text!]

▷ **Exercise 17.26** The CPS transformation can be used to implement seemingly complex control structures in a simple way. This problem examines the implementation of a simplified form of dynamically scoped exceptions with termination semantics (as presented in Section 9.5). Suppose we extend the kernel with two new constructs, `catch` and `throw`, as follows:

E ::= ... | `(catch` E_1 E_2`)` | `(throw` E`)`

In this simplified form, we have only one possible exception; therefore, we don't need exception identifiers. Here is the informal semantics of the new constructs:

- `(catch` E_1 E_2`)` evaluates E_1 to a procedure and installs it as the dynamic exception handler active during the evaluation of E_2. It is an error if E_1 does not evaluate to a procedure.

$m \in \mathit{MetaCont} = \text{LetableExp}_{cps} \rightarrow \text{Exp}_{cps}$

$mc{\rightarrow}exp : \mathit{MetaCont} \rightarrow \text{Exp}_{cps} = (\lambda m\,.\ \texttt{(lambda (}I_{temp}\texttt{)}\ (m\ I_{temp}))$
$exp{\rightarrow}mc : \text{Exp}_{cps} \rightarrow \mathit{MetaCont}$
$\quad = (\lambda E_{cps}\,.\ (\lambda LE_{cps}\,.\ \texttt{(let ((}I_{rand}\ LE_{cps}\texttt{)) (call }E_{cps}\ I_{rand}\texttt{))}))$

$\mathcal{MCPS}_{exp}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] =$
$\quad (\lambda m\,.(m\ \texttt{(lambda (}I_1\ \ldots\ I_n\ I_{kcall}\texttt{)}\ \texttt{;}\ I_{kcall}\ \texttt{fresh}$
$\qquad (\mathcal{MCPS}_{exp}[\![E]\!]\ (exp{\rightarrow}mc\ I_{kcall}))))$

$\mathcal{MCPS}_{exp}[\![\texttt{(call }E_0\ \ldots\ E_n\texttt{)}]\!] =$
$\quad (\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
$\qquad (\lambda LE_1\,.$
$\qquad\quad \ldots$
$\qquad\qquad (\mathcal{MCPS}_{exp}[\![E_n]\!]$
$\qquad\qquad\quad (\lambda LE_n\,.\texttt{(let* ((}I_1\ LE_1\texttt{)}\ \ldots\ \texttt{(}I_n\ LE_n\texttt{)))}\ \texttt{;}\ I_1\ \ldots\ I_n\ \texttt{fresh}$
$\qquad\qquad\qquad\quad \texttt{(call }I_1\ \ldots\ I_n\ (mc{\rightarrow}exp\ m)))))\ \ldots)))$

$\mathcal{MCPS}_{exp}[\![\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}]\!] =$
$\quad (\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
$\qquad (\lambda LE_1\,.$
$\qquad\quad \ldots$
$\qquad\qquad (\mathcal{MCPS}_{exp}[\![E_n]\!]$
$\qquad\qquad\quad (\lambda LE_n\,.\texttt{(let ((}I_1\ LE_1\texttt{)}\ \ldots\ \texttt{(}I_n\ LE_n\texttt{))}$
$\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ m)))\ \ldots\ )))$

$\mathcal{MCPS}_{exp}[\![\texttt{(primop }O\ E_1\ \ldots\ E_n\texttt{)}]\!] =$
$\quad (\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
$\qquad (\lambda LE_1\,.$
$\qquad\quad \ldots$
$\qquad\qquad (\mathcal{MCPS}_{exp}[\![E_n]\!]$
$\qquad\qquad\quad (\lambda LE_n\,.\texttt{(let* ((}I_1\ LE_1\texttt{)}\ \ldots\ \texttt{(}I_n\ LE_n\texttt{))}\ \texttt{;}\ I_1\ \ldots\ I_n\ \texttt{fresh}$
$\qquad\qquad\qquad\quad (m\ \texttt{(primop }O\ I_1\ \ldots\ I_n\texttt{)}))\ \ldots\ )))$

$\mathcal{MCPS}_{exp}[\![\texttt{(set! }I_{lhs}\ E_{rhs}\texttt{)}]\!] =$
$\quad (\lambda m\,.(\mathcal{MCPS}_{exp}[\![E_{rhs}]\!]$
$\qquad (\lambda LE_{rhs}\,.\ \texttt{(let ((}I_{rhs}\ LE_{rhs}\texttt{))}\ \texttt{;}\ I_{rhs}\ \texttt{fresh}$
$\qquad\quad (m\ \texttt{(set! }I_{lhs}\ I_{rhs}\texttt{))})))$

$\mathcal{MCPS}_{exp}[\![\texttt{(if }E_{test}\ E_{then}\ E_{else}\texttt{)}]\!] =$
$\quad (\lambda m\,.(\mathcal{MCPS}_{exp}[\![E_{test}]\!]$
$\qquad (\lambda LE_{test}\,.\ \texttt{(let ((}I_{kif}\ (mc{\rightarrow}exp\ m)\texttt{)}$
$\qquad\qquad\qquad\quad \texttt{(}I_{test}\ LE_{test}\texttt{))}\ \texttt{;}\ I_{test}\ \texttt{fresh}$
$\qquad\qquad\quad \texttt{(if }I_{test}$
$\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{then}]\!]\ (exp{\rightarrow}mc\ I_{kif}))$
$\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{else}]\!]\ (exp{\rightarrow}mc\ I_{kif}))))))$

Figure 17.35: Bud's alternative form of CPS conversion. Clauses not shown are unchanged.

- `(throw` E`)` evaluates E and passes the resulting value, along with control, to the currently active exception handler.

Here is a short example:

```
(catch (lambda (x) #f)
       (let ((f (lambda (x) (throw 5))))
         (catch (lambda(x) (+ 1 x))
                (f #f))))
```
$\xrightarrow[eval]{}$ *6*

The standard $\mathcal{SCPS}$ conversion rules can be modified to translate every expression into a procedure taking two continuations: a normal continuation and an exception continuation. The $\mathcal{SCPS}$ conversion rules for top level expressions, identifiers and literals are:

$\mathcal{CPS}[\![E]\!]$ = `(program (define *top* (lambda (v) v))`
`(define *except* (lambda (v)`
`"throw without catch"))`
($\mathcal{SCPS}[\![E]\!]$ `*top* *except*`))

$\mathcal{SCPS}[\![I]\!]$ = `(lambda (kn ke) (kn` I`))`

$\mathcal{SCPS}[\![L]\!]$ = `(lambda (kn ke) (kn` L`))`

a. Give the conversion rules for `(lambda (`$I_1 \ldots I_n$`)` E`)` and `(call` $E_1 \ldots E_n$`)`.

b. Give the $\mathcal{SCPS}$ conversion rules for `(throw` E`)` and `(catch` E_1 E_2`)`.

◁

▷ **Exercise 17.27** Louis Reasoner wants you to modify the CPS transformation to add a little bit of profiling information. Specifically, the modified CPS transformation should produce code that keeps a count of user procedure (not continuation) calls. Users will be able to access this information with the new construct `(call-count)` which was added to the grammar of kernel expressions:

E ::= ... | `(call-count)`

Here are some examples:

```
(begin (call (lambda (x) x) #u)
       (call-count))
```
$\xrightarrow[eval]{}$ *1*

```
(begin (call (lambda (x)
               (call (lambda (y) y) x))
             #u)
       (call-count))
```
$\xrightarrow[eval]{}$ *2*

In the modified CPS transformation, all procedures (including continuations) should take as an extra argument the number of user procedure calls so far. For example, here's the new $\mathcal{SCPS}$ rule for identifiers:

$\mathcal{SCPS}[\![I]\!]$ = `(lambda (k n) (call k` I `n))`

Give the revised $\mathcal{SCPS}$ conversion rules for `(lambda (`I`)` E`)`, `(call` E_p E_a`)`, and `(call-count)`. ◁

17.10 Transform 8: Closure Conversion

In languages with nested procedure/object declarations, code can refer to variables declared outside the innermost procedure/object declaration. As we have seen in Chapters 6–7, the meaning of such non-local references is often explained in terms environments. Traditional interpreters and compilers have a good deal of special-purpose machinery to manage environments.

The TORTOISE compiler avoids such machinery by a transformation that makes all environments explicit in the intermediate language. Each procedure is transformed into an abstract pair of code and environment, where the code explicitly accesses the environment to retrieve values formerly referenced by free variables. The resulting abstract pair is known as a **closure** because its code component is closed — i.e., it contains no free variables. The process of transforming all procedures into closures is traditionally called **closure conversion**. Because it makes all environments explicit, **environment conversion** is another good name for this transformation.

Closure conversion transforms a program that may contain higher-order procedures into one that contains only first-order procedures. It is useful not only as a transformation pass in a compiler but also as a technique that programmers can apply manually to simulate higher-order procedures in a language that supports only first-order procedures, such as C, PASCAL, and ADA.

There are numerous approaches to closure conversion that differ in terms of how environments and closures are represented. We shall first focus on one class of representations — so-called **flat closures** — and then briefly discuss some of the other options.

17.10.1 Flat Closures

We introduce closure conversion in the context of the following example:

```
(let ((linear
        (lambda (a b)
          (lambda (x)
            (@+ (@* a x) b)))))
  (let ((f (call linear 4 5))
        (g (call linear 6 7)))
    (@+ (call f 8) (call g 9)))).
```

Given `a` and `b`, the `linear` procedure returns a procedural representation of a line with slope `a` and y-intercept `b`. The `f` and `g` procedures two such lines, each of which is associated with the abstraction `(lambda (x) ...)`, which has free variables `a` and `b`. In the case of `f`, these variables have the bindings 4 and 5, respectively, while for `g` they have the bindings 6 and 7.

We begin by considering how to closure convert this example by hand, and then will develop a transformation that performs the conversion automatically. One way to represent `f` and `g` as closed procedures is shown below:

```
(let ((fg_code
        (lambda (env x)
          (let* ((a (@mget 1 env))
                 (b (@mget 2 env)))
            (@+ (@* a x) b))))
      (f_env (@mprod 4 5))
      (g_env (@mprod 6 7)))
  (let ((f_clopair (@mprod fg_code f_env))
        (g_clopair (@mprod fg_code g_env)))
    (@+ (call (@mget 1 f_clopair) (@mget 2 f_clopair) 8)
        (call (@mget 1 g_clopair) (@mget 2 g_clopair) 9))))
```

In this approach, the two procedures share the same code component, $\texttt{fg}_{code}$, which takes an explicit environment argument `env` in addition to the normal argument `x`. The argument is assumed to be a tuple whose two components are the values of the former free variables `a` and `b`. These values are extracted from the environment and given their former names in a wrapper around the body expression `(@+ (@* a x) b)`. Note that $\texttt{fg}_{code}$ has no free variables and so is a closed procedure. The environments $\texttt{f}_{env}$ and $\texttt{g}_{env}$ are tuples holding the free variable values. The closures $\texttt{f}_{clopair}$ and $\texttt{g}_{clopair}$ are formed by making explicit **code/env pairs** that pair the shared code component with the individual environment. To handle the change in procedure representation, each call of the form `(call f` E`)` must be transformed to `(call (@mget 1` $\texttt{f}_{clopair}$`) (@mget 2` $\texttt{f}_{clopair}$`)` E`)` (and similarly for `g`) in order to pass the environment component as the first argument to the code component.

It's worth emphasizing at this point that closure conversion is basically an

exercise in abstract data type implementation. The abstract data type being considered is the procedure, which is manipulated by an interface with two operations: `lambda`, which creates procedures, and `call`, which applies procedures. The goal of closure conversion is to find a different implementation of this interface that has the same behavior but in which the procedure creation form has no free variables. As in traditional data structure problems, we're keen on designing implementations that not only have the correct implementation, but are as efficient as possible.

For example, a more efficient approach to using explicit code/env pairs is to collect the code and free variable values into a single tuple, as shown below.

```
(let ((fg_code'
       (lambda (clo x)
         (let* ((a (@mget 2 clo))
                (b (@mget 3 clo)))
           (@+ (@* a x) b)))))
  (let ((f_clo (@mprod fg_code' 4 5))
        (g_clo (@mprod fg_code' 6 7)))
    (@+ (call (@mget 1 f_clo) f_clo 8)
        (call (@mget 1 g_clo) g_clo 9))))
```

This approach, which is known as **closure passing style**, avoids creating a separate environment tuple every time a closure is created, and avoids extracting this tuple from the code/environment pair every time the closure is invoked.

If we systematically use closure passing style to transform every abstraction and application site in the original `linear` example, we get the result show in Figure 17.36. The inner `lambda` has been transformed into a tuple that combines $\texttt{fg}_{code}$ with the value of the free variables `a` and `b` from the outer `lambda`. For consistency, the outer `lambda`, has also been transformed; its tuple has only a code component since the original `lambda` has no free variables.

Before we study the formal closure conversion transformation, we consider one more example (Figure 17.37), which involves nesting of open procedures and unreferenced variables. In the unconverted `clotest`, the outermost abstraction, `(lambda (c d) ...)`, is closed; the middle abstraction, `(lambda (r s t) ...)`, has `c` as its only free variable (`d` is never used); and the innermost abstraction, `(lambda (y) ...)`, has {`c`, `r`, `t`} as its free variables (`d` and `s` are never used). In the converted `clotest`, each abstraction has been transformed into a tuple that combines a closed code component with all the free variables of the original abstraction. The resulting tuples are call **flat closures** because all the environment information has been condensed into a single tuple that does not reflect any of the original nesting structure. Note that unreferenced variables from an enclosing scope are ignored. For example, the innermost body does not reference

```
(let ((linear
        (@mprod ;; this product has only a code component
          (lambda (clo1 a b) ;; clo1 unused
            (@mprod ;; this product has code + vars a,b
              (lambda (clo2 x)
                (let* ((a (@mget 2 clo2))
                       (b (@mget 3 clo2)))
                  (@+ (@* a x) b)))
              a b)) ;; free vars of clo2
          ))) ;; clo1 has no free vars
  (let ((f (call (@mget 1 linear) linear 4 5))
        (g (call (@mget 1 linear) linear 6 7)))
    (@+ (call (@mget 1 f) f 8)
        (call (@mget 1 g) g 9))))
```

Figure 17.36: Result of closure converting the `linear` example.

`d` and `s`, so these variables are not extracted from `clo3` and are not included in the innermost tuple.

A formal specification of the flat closure conversion transformation is presented in Figure 17.38. The transformation is specified via the $\mathcal{CL}$ function on SILK expressions. The only non-trivial clauses for $\mathcal{CL}$ are `lambda` and `call`. $\mathcal{CL}$ converts a `lambda` to a tuple containing a closed code component and all the free variables of the abstraction. The code component is derived from the original `lambda` by adding a closure argument and extracting the free variables from this argument in a wrapper around the body. The order of the free variables is irrelevant as long as it is consistent between the tuple creation and projection forms.

A `call` is converted to another call that applies the code component of the converted rator closure to the closure and the converted operands. A difference from the examples studied above is that $\mathcal{CL}$ introduces a `let*` to name the closure and its code component.[4] This guarantees that any input in CPS form will be translated to an output in CPS form. However, the unique naming property is *not* preserved by $\mathcal{CL}$. The names I_{fv_i} declared in the body of the closed abstraction stand for variables that are logically distinct from variables with the same names in the closure tuple.

In order to work properly, $\mathcal{CL}$ requires that the input expression contain no occurrences of `set!`. This is because the copying of free variable values by

[4] In the `call` clause, the binding of I_{clo} to E_{rator} is only necessary if E_{rator} is not already an identifier. We will omit I_{clo} in examples unless it is necessary.

Unconverted expression

```
(let ((clotest
        (lambda (c d)
          (lambda (r s t)
            (lambda (y)
              (@+ (@/ (* r y) t) (@- r c)))))))
  (let ((p (call clotest 4 5)))
    (let ((q1 (call p 6 7 8))
          (q2 (call p 9 10 11)))
      (+ (call q1 12) (call q2 13))))).
```

Converted expression

```
(let ((clotest
        (@mprod ;; this product has only a code component
          (lambda (clo1 c d) ;; clo1 is unused
            (@mprod ;; this product has code + var c
              (lambda (clo2 r s t)
                (let* ((c (@mget 2 clo2)))
                  (@mprod ;; this product has code + vars c,r,t
                    (lambda (clo3 y)
                      (let* ((c (@mget 2 clo3))
                             (r (@mget 3 clo3))
                             (t (@mget 4 clo3)))
                        (@+ (@/ (* r y) t) (@- r c))))
                    c r t))) ;; free vars of clo3 = c,r,t
              c)) ;; free vars of clo2 = c
          ))) ;; clo1 has no free vars
  (let ((p (call (@mget 1 clotest) clotest 4 5)))
    (let ((q1 (call (@mget 1 p) p 6 7 8))
          (q2 (call (@mget 1 p) p 9 10 11)))
      (+ (call (@mget 1 q1) q1 12) (call (@mget 1 q2) q2 13))))).
```

Figure 17.37: Flat closure conversion on an example with nested open procedures.

$\mathcal{CL}$: Exp → Exp

Preconditions: The input expression is assignment-free.

Postconditions:

- All `lambda`s in the output expression are closed.
- The output expression is assignment-free.

Other properties:

- If the input expression is in CPS form, so is the output expression.

$$
\begin{array}{l}
\mathcal{CL}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] \\
\quad = \textbf{let}\,\{I_{fv_1},\ldots,I_{fv_k}\}\ \textbf{be}\ \mathit{FreeIds}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!] \\
\qquad \textbf{in}\ \texttt{(@mprod (lambda (}I_{clo}\ I_1\ \ldots\ I_n\texttt{)}\ \texttt{;}\ I_{clo}\ \texttt{fresh} \\
\qquad\qquad \texttt{(let* ((}I_{fv_1}\ \texttt{(mget 2}\ I_{clo}\texttt{))} \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \texttt{(}I_{fv_k}\ \texttt{(mget k+1}\ I_{clo}\texttt{)))} \\
\qquad\qquad \mathcal{CL}[\![E_{body}]\!]\texttt{))} \\
\qquad\quad I_{fv_1}\ \ldots\ I_{fv_k}\texttt{)}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CL}[\![\texttt{(call}\ E_{rator}\ E_1\ \ldots\ E_n\texttt{)}]\!] \\
\quad = \texttt{(let* ((}I_{clo}\ \mathcal{CL}[\![E_{rator}]\!]\texttt{)}\ \texttt{;}\ I_{clo}\ \texttt{fresh} \\
\qquad\quad \texttt{(}I_{code}\ \texttt{(mget 1}\ I_{clo}\texttt{)))}\ \texttt{;}\ I_{code}\ \texttt{fresh} \\
\qquad \texttt{(call}\ I_{code}\ I_{clo}\ \mathcal{CL}[\![E_1]\!]\ \ldots\ \mathcal{CL}[\![E_n]\!]\texttt{))}
\end{array}
$$

All other clauses of $\mathcal{CL}$ are purely structural.

Figure 17.38: The flat closure conversion transformation $\mathcal{CL}$ of TORTOISE.

$\mathcal{CL}$ in the `lambda` clause does not preserve the semantics of mutable variables. Consider the following example of a nullary function that increments a counter every time it is called:

```
(let ((count 0))
  (lambda ()
    (let* ((new-count (+ count 1))
           (ignore (set! count new-count)))
      new-count)))
```

Closure converting this example yields:

```
(let ((count 0))
  (@mprod
    (lambda (clo)
      (let* ((count (@mget clo 2)))
        (let* ((new-count (+ count 1))
               (ignore (set! count new-count)))
          new-count)))
    count))
```

The `set!` in the tranformed code changes the local variable `count` within the lambda, which is always initially bound to the value 0. So the closure converted procedure always returns 1, which is not the correct behavior. Performing assignment conversion before closure conversion fixes this problem, since `count` will then name a sharable mutable cell rather than a number.

Figure 17.39 shows the running `revmap` example after closure conversion. In addition to transforming procedures present in the original code (`.clo56`[5] is `revmap`, `.clo55` is `loop`, `.clo45` is the greater-than-`b` procedure), closure conversion also tranforms the continuation procedures introduced by CPS conversion (`.clo54` is the continuation for the `f` call). The free variables in converted continuations are those values that would typically be saved on the stack across the subroutine call associated with the continuation. For example, continuation closure `.clo54` includes the values needed by the loop after a call to `f`: the loop state variable `xs.9`, the looping procedure `loop.8`, the end-of-loop continuation `k.27`, and the mutable cell `t.23` resulting from the assignment conversion of `ans`. Note that the closure named `loop.8` contains `loop.8` in its environment. The recursive scope of `cycrec` guarantees that the the looping procedure has been transformed into a looping (i.e., cyclic) data structure.

[5] By convention, we will refer to a closure tuple by the name of the first argument of its code component.

```
(silk (a.1 b.2 ktop.11)
 (let* ((abs.12
         (@mprod ; MPROD1
          (lambda (clo.56 f.5 lst.6 k.22)
            (let* ((t.24 (@null)) (t.23 (@mprod t.24))) ; MPROD2
              (cycrec
               ((loop.8
                 (@mprod ; MPROD3
                  (lambda (clo.55 xs.9 k.27)
                    (let* ((t.23 (mget 2 clo.55)) (loop.8 (mget 3 clo.55))
                           (f.5 (mget 4 clo.55)) (t.29 (@null? xs.9)))
                      (if t.29
                          (let* ((t.39 (mget 1 t.23)) (rator.49 k.27)
                                 (code.48 (mget 1 rator.49)))
                            (call code.48 rator.49 t.39))
                          (let* ((t.32 (@car xs.9))
                                 (k.38 (@mprod ; MPROD4
                                        (lambda (clo.54 t.33)
                                          (let* ((t.23 (mget 2 clo.54))
                                                 (xs.9 (mget 3 clo.54))
                                                 (loop.8 (mget 4 clo.54))
                                                 (k.27 (mget 5 clo.54))
                                                 (t.34 (mget 1 t.23))
                                                 (t.31 (@cons t.33 t.34))
                                                 (t.30 (mset! 1 t.23 t.31))
                                                 (t.35 (@cdr xs.9))
                                                 (rator.53 loop.8)
                                                 (code.52 (mget 1 rator.53)))
                                            (call code.52 rator.53 t.35 k.27)))
                                        t.23 xs.9 loop.8 k.27)) ; end MPROD4
                                 (rator.51 f.5)
                                 (code.50 (mget 1 rator.51)))
                            (call code.50 rator.51 t.32 k.38)))))
                  t.23 loop.8 f.5))) ; end MPROD3
                (let* ((rator.47 loop.8) (code.46 (mget 1 rator.47)))
                  (call code.46 rator.47 lst.6 k.22)))))) ; end MPROD1
        (abs.13 (@mprod ; MPROD5
                 (lambda (clo.45 x.4 k.20)
                   (let* ((b.2 (mget 2 clo.45)) (t.21 (@> x.4 b.2))
                          (rator.44 k.20) (code.43 (mget 1 rator.44)))
                     (call code.43 rator.44 t.21)))
                 b.2)) ; end MPROD5
        (t.16 (@* a.1 7)) (t.17 (@null))
        (t.15 (@cons t.16 t.17)) (t.14 (@cons a.1 t.15))
        (rator.42 abs.12) (code.41 (mget 1 rator.42)))
   (call code.41 rator.42 abs.13 t.14 ktop.11)))
```

Figure 17.39: Running example after closure conversion.

17.10.2 Variations on Flat Closure Conversion

Now we consider several variations on flat closure conversion. We begin with an optimization to $\mathcal{CL}$. Why does $\mathcal{CL}$ transform an already closed `lambda` into a closure tuple? This strategy simplifies the transformation by enabling all call sites to be transformed uniformly to "expect" such a tuple. But it is also possible to use non-uniform transformations on abstractions and call sites as long as the correct behavior is maintained. Given accurate **flow information** that indicates which procedures flow to which call sites, we can do a better job via so-called **selective closure conversion**. In this approach, originally closed procedures that flow only to call sites where only originally closed procedures are called are left unchanged by the closure conversion process, as are their call sites. This avoids unnecessary tuple creations and projections. The result of selective closure conversion for the `linear` example is presented in Figure 17.40. The kind of flow analysis necessary to enable selective closure conversion is beyond the scope of this text; see the reading section at the end of this chapter for more information.

```
(let ((linear
       (lambda (a b) ;; this closed lambda is not transformed
         (@mprod ;; this product has three components
           (lambda (clo2 x)
             (let* ((a (@mget 2 clo2))
                    (b (@mget 3 clo2)))
               (@+ (@* a x) b)))
           a b)))) ;; free vars of clo2
  (let ((f (call linear 4 5))  ;; this call site is not transformed
        (g (call linear 6 7))) ;; this call site is not transformed
    (@+ (call (@mget 1 f) f 8)
        (call (@mget 1 g) g 9))))
```

Figure 17.40: Result of selective closure conversion in the `linear` example.

In selective closure conversion, a closed procedure p_{closed} cannot be optimized when it is called at the same call site s as an open procedure p_{open} in the original program. The call site must be transformed to expect for its rator a closure tuple for p_{open}, and so p_{closed} must also be represented as a closure tuple since it flows to rator position of s. This representation constraint can similarly force other closed procedures that share call sites with p_{closed} to be converted, leading to a contagious phenomenon called **representation pollution**. For example, although `f` is closed in the following example, because it flows to the same call site as open procedure `g`, selective closure conversion must still convert `f` to a

closure tuple:

```
(lambda (b c)
  (let ((f (lambda (x) (+ x 1)))
        (g (let ((a (if b 4 5)))
             (lambda (y) (+ (* a y) c)))))
    (+ (call f 2)
       (call (if b f g) 3))))
```

Representation pollution can sometimes be avoided by duplicating a closed procedure, and using different representations for the two copies. For instance, if we split `f` in the above example into two copies, then the copy that flows to the call site `(call f 2)` need not be converted to a tuple.

It is always possible to handle heterogeneous procedure representations by affixing tags to procedures that indicate their representation and then dispatching on these tags at every call site where different representations are known to flow together. For example, using the oneof notation introduced in Section 10.2.2, we can use `code` to tag a closed procedure and `closure` to tag a closure tuple, as in the following conversion of the above example:

```
(lambda (b c)
  (let ((f1 (lambda (x) (+ x 1)))
        (f2 (one code (lambda (x) (+ x 1))))
        (g  (let ((a (if b 4 5)))
              (one closure
                (@mprod (lambda (clo y)
                          (let ((a (@mget 2 clo))
                                (c (@mget 3 clo)))
                            (+ (* a y) c)))
                        a c)))))
    (+ (call f1 2)
       (call-generic (if b f2 g) 3)))),
```

where `(call-generic` E_{rator} E_1 ... E_n`)` desugars to

```
(let ((I1 E1) ... (In En)) ; I1 ... In are fresh
  (tagcase Erator Irator
    (code (call Irator I1 ... In))
    (closure (call (@mget 1 Irator) Irator I1 ... In))).
```

Note that `(call f1 2)` is a regular call to an unconverted closed procedure. This tagging strategy is not necessarily a good idea. Analyzing and converting programs to handle tags is complex, and the overhead of tag manipulation can offset the gains made by reducing representation pollution.

In an extreme version of the tagging strategy, all procedures that flow to

a given call site are viewed as members of a sum-of-products datatype. Each element in this datatype is a tagged environment tuple, where the tag indicates which abstraction created the procedure and the environment tuple holds the free variable values of the procedure. A procedure call can then be converted to a dispatch on the environment tag that calls a associated closed procedure. For example:

```
(lambda (b c)
  (let ((fcode (lambda (x) (+ x 1)))
        (fenv (one abs1 (@mprod)))
        (gcode (lambda (y a c) (+ (* a y) c)))
        (genv (let ((a (if b 4 5))) (one abs2 (@mprod a c)))))
    (+ (call fcode 2)
       (call-env1 (if E_test fenv genv) 3))),
```

where `(call-env1` E_{env} E_{rand}`)` is an abbreviation for

```
(let ((I_rand E_1))
  (tagcase E_env I_rator
    (abs1 (call fenv I_rand))
    (abs2 (call genv I_rand (@mget 1 env) (@mget 2 env))))).
```

The procedure call overhead in the dispatch can often be reduced by an **inlining** process that replaces some calls by appropriately rewritten copies of their bodies. E.g., `call-env1` could be rewritten to:

```
(let ((I_rand E_1))
  (tagcase E_env I_env
    (abs1 (+ I_rand 1))
    (abs2 (+ (* (@mget 1 I_env) I_rand) (@mget 2 I_env))))).
```

The environment tagging strategy is known as **defunctionalization** because it removes all higher-order functions from a program. Defunctionalization is an important closure conversion technique for languages (such as ADA and PASCAL) in which function pointers cannot be stored in data structures — a feature required in all the previous techniques. Some drawbacks of defunctionalization are that it requires the whole program (it cannot be performed on individual modules) and application functions like `call-env1` might need to dispatch on all abstractions in the entire program. In practice, type and flow information can be used to significantly narrow the set of abstractions that need to be considered at a given call site.

A closure need not carry with it the value of a free variable if that variable is available in all contexts where the closure is invoked. This observation is the key idea in so-called **lightweight closure conversion**, which can decrease the number of free variables by adding extra arguments to procedures if those

arguments are always dynamically available at all call sites for the procedures. In our example, the lightweight optimization is realized by rewriting the original example as follows before performing other closure conversion techniques:

```
(lambda (b c)
  (let ((f (lambda (x c) (+ x 1))) ; 3. By 2, need param c here.
        (g (let ((a (if b 4 5)))
             (lambda (y c) (+ (* a y) c))))) ; 1. Add c as param.
    (+ (call f 2 c) ; 4. By 3, must add c as an arg here, too.
       (call (if b f g) 3 c)))) ; 2. By 1, need arg c here.
```

Since `g`'s free variable `c` is available at the one site where `g` is called, we should be able to pass it as an argument at the site rather than storing it in the closure for `g`. But representation constraints also force us to add `c` as an argument to `f`, since `f` shares a call site with `g`. If `f` were called in some context outside the scope of `c`, this fact would invalidate the proposed optimization. This example only hints at the sophistication in analysis that is necessary to perform lightweight closure conversion in practice.

17.10.3 Linked Approaches

Thus far we have assumed that all free variables values of a procedure are stored in a single flat environment or closure. This strategy minimizes the information carried in a particular closure. However, it is often the case that a free variable is referenced by several closures. Setting aside a slot for (a pointer to) the value of this variable in several closures/environments increases the space requirements of the program. For example, in the flat `clotest` example of Figure 17.37, closures `p`, `q1`, and `q2` all contain a slot for the value of free variable `c`.

An alternative approach is to structure closures to enhance sharing and reduce copying. In a code/env model, a high degree of sharing is achieved when every call site bundles the environment of the called procedure (a.k.a., the **parent environment**) together with the argument values to create the environment for the body of the called procedure. In this approach, each closed abstraction takes a single argument, its environment, and all variables are accessed through this environment. This is called the **linked environment** approach because environments are linked together in chains.

Figure 17.41 shows this approach for the `clotest` example. Note that the first slot of environments `env1`, `env2`, and `env3` contains (a pointer to) its parent environment. Variables declared by the closest enclosing `lambda` are accessed directly from the environment, but variables declared in outer `lambdas` require one or more indirections through parent environments. For instance, in the body of the innermost `lambda`, variable `r`, which is the first

argument one environment back, is accessed via `(@mget 2 (@mget 1 env3))`, while variable `y`, which is the first argument two environments back, is accessed via `(@mget 2 (@mget 1 (@mget 1 env3)))`. In general, each variable has a **lexical address** $\langle back, over \rangle$, where *back* indicates how many environments back the variable is located and *over* indicates its argument position in the the resulting environment. A variable with lexical address $\langle b, o \rangle$ is translated to `(@mget` o `(@mget`^{b} `1` env`))`, where env is the current lexical environment and `(@mget`^{b} `1` e`)` stands for the b-fold composition of the first projection starting with e. Traditional compilers often use such lexical addresses to locate variables on a stack, where so-called **static links** are used to model chains of parent environments.

```
(let ((env0 (@mprod)))
  (let ((clotest
         (@mprod
           (lambda (env1) ; env1 = <env0,c,d>
             (@mprod
               (lambda (env2) ; env2 = <env1,r,s,t>
                 (@mprod
                   (lambda (env3) ; env3 = <env2,y>
                     (+ (/ (* (@mget 2 (@mget 1 env3)) ; r
                              (@mget 2 env3)) ; y
                           (@mget 4 (@mget 1 env3))) ; t
                        (- (@mget 2 (@mget 1 env3)) ; r
                           (@mget 2 (@mget 1 (@mget 1 env3))))))) ; c
                   env2))
               env1))
           env0)))
    (let ((p (call (@mget 1 clotest) (@mprod (@mget 2 clotest) 4 5))))
      (let ((q1 (call (@mget 1 p) (@mprod (@mget 2 p) 6 7 8)))
            (q2 (call (@mget 1 pP (@mprod (@mget 2 p) 9 10 11)))))
        (+ (call (@mget 1 q1) (@mprod (@mget 2 q1) 12))
           (call (@mget 1 q2) (@mprod (@mget 2 q2) 13)))))))
```

Figure 17.41: A version of the `clotest` example with linked environments.

Figure 17.42 depicts the shared environment structure in the `clotest` example with linked environments. Note how the environment of `p` is shared as the parent environment of `q1`'s environment and `q2`'s environment. In contrast with the flat environment case, `p`, `q1`, and `q2` all share the same slot holding `c`, so less slot space is needed for `c`. Another advantage of sharing is that the linked environment approach to closure conversion can support `set!` directly without

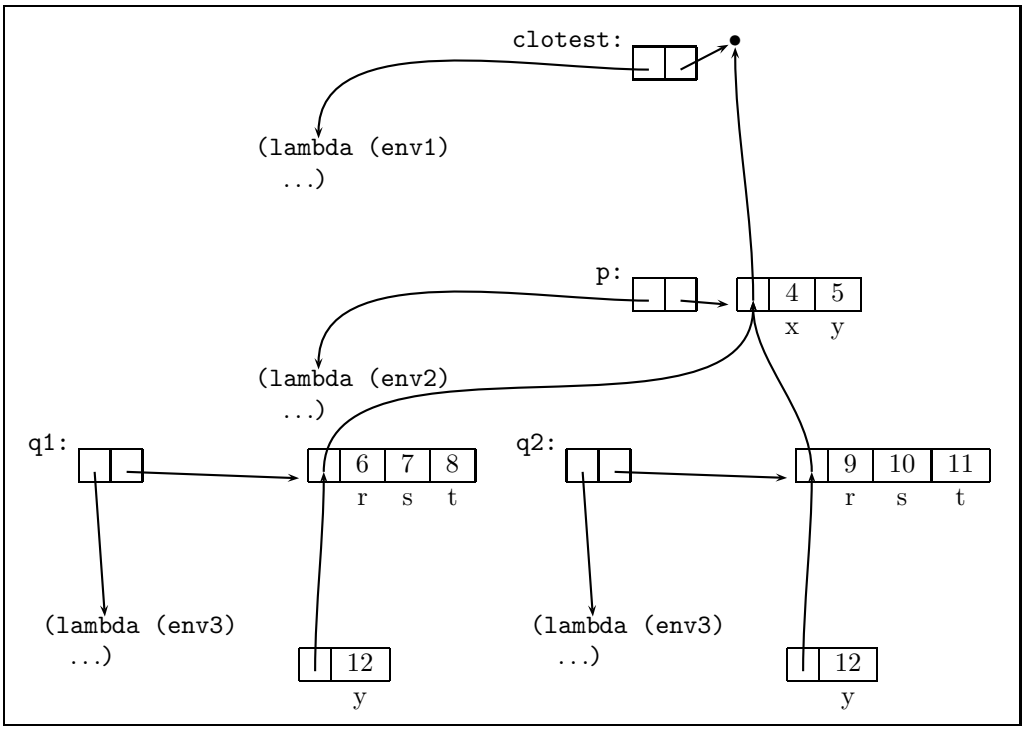


Figure 17.42: Figure depicting the links in the linked `clotest` example. (This figure needs lots of reformatting work!

the need for assignment conversion (see Exercise **??**).

However, there are several downsides to linked environments. First, variable access is slower than for flat closures due to the indirections through parent environment links. Second, environment slots hold values (such as `d` and `s`) that are never referenced, so space is wasted on these slots. A final subtle point is that shared slots can hold onto values longer than they are actually needed by a program, leading to space leaks. Some of these points and some alternative linked strategies are explored in the exercises.

▷ **Exercise 17.28**

a. $\mathcal{CL}$ is not idempotent. Explain why. Can any closure conversion transformation be idempotent?

b. In the `lambda` clause for $\mathcal{CL}$, suppose $\mathit{FreeIds}[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}]\!]$ is replaced by the set of all variables in scope at that point. Is this a meaning-preserving change? What are the advantages and disadvantages of of such a change?

c. In a SILK-based compiler, $\mathcal{CL}$ must be necessarily be performed after an assignment conversion pass. Could we perform it before a renaming pass? A globalization pass? A CPS-conversion pass? Explain. ◁

▷ **Exercise 17.29** In the `lambda` clause, the $\mathcal{CL}$ function uses a **wrapping strategy** to wraps the body of the original `lambda` in a `let*` that extracts and names each free variable value in the closure. An alternative **substitution strategy** is to replace each free reference in the original `lambda` by a closure access. E.g, here is a modified version of $\texttt{fg}_{code}{}'$ that uses the substitution strategy:

```
(lambda (clo x) (@+ (@* (@mget 2 env) x) (@mget 3 env)))
```

Neither strategy is the best in all situations. Describe situations in which the wrapping strategy is superior and in which the substitution strategy is superior. State all the assumptions of your argument. ◁

▷ **Exercise 17.30** Consider the following SILK abstraction E_{abs}:

```
(lambda (b)
  (let ((f (lambda (x) (@+ x 1)))
        (g (lambda (y) (@* y 2)))
        (h (lambda (a) (lambda (z) (@/ z a))))
        (p (lambda (r) (call r 3))))
    (@+ (call (if b f g) 4)
        (@* (call p (call h 5)) (call p (call h 6))))))
```

a. Show the result of applying flat closure conversion to E_{abs}.

b. The transformation can be improved if we use selective closure conversion instead. Show the result of selective closure conversion on E_{abs}.

c. Suppose we replace `(call h 6)` by `g` in E_{abs} to give E_{abs}'. Then selective closure conversion on E_{abs}' does not yield an improvement over regular closure conversion on E_{abs}'. Explain why.

d. Describe a simple meaning-preserving change to E_{abs}' after which selective closure conversion will be an improvement over regular closure conversion. ◁

▷ **Exercise 17.31** Consider the following SILK program

```
(silk (n)
  (let* ((p (lambda (w)
              (if (@= 0 x)
                  (lambda (x) x)
                  (if (@= 0 (@% n 2))
                      (let ((p1 (p (@/ w 2))))
                        (lambda (y) (@* 2 (call p1 y))))
                      (let ((p2 (p (@- w 1))))
                        (lambda (z) (@+ 1 (call p2 z)))))))))
    (let ((q (call p n)))
      (+ (call q 1) (call q n)))))
```

Using closure conversion techniques presented in this section, translate this program into C, PASCAL, and JAVA. The program has the property that equality and remainder primops are performed only when `p` is called, not when `q` is called. Your translated programs should also have this property. ◁

17.11 Transform 9: Lifting

Programmers nest procedures when an inner procedure needs to use variables that are defined in an outer procedure. The free variables in such an inner procedure are bound by the outer procedure. We have seen that closure conversion eliminates free varaibles in every procedure. However, because it leaves abstractions in place, it does not eliminate procedure nesting.

A procedure is said to be at **top-level** when it is defined at the outermost scope of a program. **Lifting** (also called **lambda lifting**) is the process of eliminating nested procedures by making all procedures top-level. Of course, all procedures must be closed before lifting is performed. The process of bringing all procedures to top level would necessarily remove the fundamental connection between free variable references and their associated declarations.

Compiling a procedure with nested internal procedures requires placing branch instructions around the code for the internal procedures. We eliminate such branches by insisting that all procedures be lifted after they are closed. Once

$P_{lft} \in \text{Program}_{lft}$ $L \in \text{Lit}$
$E_{lft} \in \text{Exp}_{lft}$ $I \in \text{Identifier}_{lft}$ = usual identifiers
$BV_{lft} \in \text{BindingValue}_{lft}$ $B \in \text{Boollit}_{lft} = \{\texttt{\#t}, \texttt{\#f}\}$
$LE_{lft} \in \text{LetableExp}_{lft}$ $N \in \text{Intlit}_{lft} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$V_{lft} \in \text{ValueExp}_{lft}$ $O \in \text{Primop}_{lft}$ = as in full SILK.

P_{lft} ::= `(silk (`I_{fml}`*) (cycrec ((`I `(lambda (`I`*) `E_{lft}`))*) `E_{lft}`))`
E_{lft} ::= `(call `V_{lft} V_{lft}`*)` | `(if `V_{lft} E_{lft} E_{lft}`)` | `(error `I`)`
| `(let ((`I LE_{lft}`)) `E_{lft}`)` | `(cycrec ((`I BV_{lft}`)*) `E_{lft}`)`
V_{lft} ::= L | I
LE_{lft} ::= V_{lft} | `(primop `O_{op} V_{lft}`*)`
BV_{lft} ::= L | `(primop mprod `V_{lft}`*)`
L ::= `#u` | B | H | N

Figure 17.43: Grammar for SILK$_{lft}$, the target language of the TORTOISE compiler.

all of the procedures in a program are at top-level, each can be compiled into straight-line code. Avoiding unnecessary unconditional branches is especially important for processors that have instruction caches, instruction prefetching, or pipelined architectures. Lifting is also an important transform when compiling to certain, less common, architectures, like combinator reduction machines[Hug82].

The result of the lifting phase is a program in SILK$_{lft}$, a restricted form of SILK$_{cps}$ presented in Figure 17.43. The key difference between SILK$_{lft}$ and SILK$_{cps}$ is that SILK$_{lft}$ abstractions may only occur in a top-level `cycrec` in the program body. Each such abstraction may be viewed as an assembly code subroutine.

We now specify the lifting conversion transformation $\mathcal{LC}_{prog}$:

Preconditions: The input to $\mathcal{LC}_{prog}$ is a program in which every abstraction is closed.

Postconditions: The output of $\mathcal{LC}_{prog}$ is a program in which every abstraction is in the top-level `cycrec` of a program, as specfied in the SILK$_{lft}$ grammar in Figure 17.43.

Here is the algorithm employed by $\mathcal{LC}_{prog}$:

1. Associate with each `lambda` abstraction a new name. This name must be unique in the sense that it is distinct from every variable name in the program and the name chosen for every other abstraction.

2. Replace each abstraction by a reference to its unique name.

3. Replace the body E_{body} of the program with a `cycrec` of the form

$$\begin{array}{l}\texttt{(cycrec ((}I_{lam1}\ \ AB_1\texttt{')}\\ \qquad\qquad\vdots\\ \qquad\texttt{(}I_{lamn}\ \ AB_n\texttt{'))}\\ E_{body}\texttt{'),}\end{array}$$

where

- $AB_1\texttt{'} \ldots AB_n\texttt{'}$ are the transformed versions of all the abstractions in the original program;
- $I_{lam1}\ \ldots I_{lamn}$ are the unique names associated with the original abstractions; and
- $E_{body}\texttt{'}$ is the transformed body of the program.

For example, Figure 17.44 shows our running example after lambda lifting. Note that replacing each abstraction with its unique variable name can introduce free variables into otherwise closed abstractions. For instance the body of the abstraction named `lam.58` contains a reference to `lam.59` and the body of the abstraction named `lam.59` contains a reference to `lam.60`. So the abstractions are no longer closed after lifting! All free variables thus introduced are declared in the top-level `cycrec` and are effectively treated as global names. In the analogy with assembly code, these names correspond to assembly code labels that name the first instruction in the subroutine corresponding to the abstraction.

17.12 Transform 10: Data Conversion

[This section is still under construction. Stay tuned!]

17.13 Garbage Collection

*[This section is also still under construction. Stay tuned!]*x

Reading

The literature on traditional compiler technology is vast. A classic text is the "Dragon book" [ASU86]. More modern treatments are provide by Cooper and

```
(silk (a.1 b.2 ktop.11)
 (cycrec
    ((lam.57 (lambda (clo.45 x.4 k.20)
               (let* ((b.2 (mget 2 clo.45))
                      (t.21 (@> x.4 b.2))
                      (code.43 (mget 1 k.20)))
                 (call code.43 k.20 t.21))))
     (lam.58 (lambda (clo.56 f.5 lst.6 k.22)
               (let* ((t.24 (@null))
                      (t.23 (@mprod t.24)))
                 (cycrec ((loop.8 (@mprod lam.59 t.23 loop.8 f.5)))
                   (let ((code.46 (mget 1 loop.8)))
                     (call code.46 loop.8 lst.6 k.22))))))
     (lam.59 (lambda (clo.55 xs.9 k.27)
               (let* ((t.23 (mget 2 clo.55))
                      (loop.8 (mget 3 clo.55))
                      (f.5 (mget 4 clo.55))
                      (t.29 (@null? xs.9)))
                 (if t.29
                     (let* ((t.39 (mget 1 t.23))
                            (code.48 (mget 1 k.27)))
                       (call code.48 k.27 t.39))
                     (let* ((t.32 (@car xs.9))
                            (k.38 (@mprod lam.60 t.23 xs.9 loop.8 k.27))
                            (code.50 (mget 1 f.5)))
                       (call code.50 f.5 t.32 k.38))))))
     (lam.60 (lambda (clo.54 t.33)
               (let* ((t.23 (mget 2 clo.54))
                      (xs.9 (mget 3 clo.54))
                      (loop.8 (mget 4 clo.54))
                      (k.27 (mget 5 clo.54))
                      (t.34 (mget 1 t.23))
                      (t.31 (@cons t.33 t.34))
                      (t.30 (mset! 1 t.23 t.31))
                      (t.35 (@cdr xs.9))
                      (code.52 (mget 1 loop.8)))
                 (call code.52 loop.8 t.35 k.27)))))
  (let* ((abs.12 (@mprod lam.58))
         (abs.13 (@mprod lam.57 b.2))
         (t.16 (@* a.1 7))
         (t.17 (@null))
         (t.15 (@cons t.16 t.17))
         (t.14 (@cons a.1 t.15))
         (code.41 (mget 1 abs.12)))
    (call code.41 abs.12 abs.13 t.14 ktop.11))))
```

Figure 17.44: Running example after lambda lifting.

Torczon [CT03] and by Appel's textbooks [App98b, App98a, AP02]. Comprehensive coverage of advanced compilation topics, especially optimizations, can be found in Muchnick's text [Muc97]. Inlining is a particularly important but subtle optimization [**?**, **?**, **?**, **?**]. Issues in functional language compilation are considered by Peyton Jones in [Pey87].

The notion of compiling programs via transformations on a lambda-calculus based intermediate language was pioneered in the Scheme community through a series of Scheme compilers that started with Steele's Rabbit [Ste78], and was followed many others [Roz84, KKR$^+$86, **?**, **?**, **?**]. Kelsey [Kel89, KH89] demonstrated that the transformational technique was viable for languages other than Scheme.

The next major innovation along these lines was developing transformation-oriented compilers based on explicitly **typed intermediate languages** (e.g. [Mor95, TMC$^+$96, Jon96, JM97, Sha97, BKR99, TO98, MWCG99, FKR$^+$00, CJW00, DWM$^+$01]. The type information guides program analyses and transformations, supports run-time operations such as garbage collection, and is an important debugging aid in the compiler development process. In [TMC$^+$96], Tarditi and others explored how to express classical optimizations within a typed intermediate langauge framework. In some compilers (e.g. [MWCG99]) type information is carried all the way through to a **typed assembly language**, where types can be used to verify certain safety properties of the code. The notion that untrusted low-level code should carry information that allows safety properties to be verified is the main idea in **proof-carrying code**[NL98, AF00].

Early transformation-based compilers typically included a stage converting the program to CPS form. The view that procedure calls can be viewed as jumps that pass arguments was first championed by Steele in [Ste77]. He observed that a stack discipline in compilation is not implied by the procedure call mechanism but rather by the evaluation of nested subexpressions. The Tortoise $\mathcal{MCPS}$ transform is based on a study of CPS conversion by Danvy and Filinski [DF92]. They distinguish so-called **static continuations** (what we call "meta-continuations") from **dynamic continuations** and used these notions to derive an efficient form of CPS conversion from the simple-but-inefficient definition. Appel studied the use of continuations for compiler optimizations in [App92]. In [FSDF93], Flanagan et al. argued that explicit CPS form was not necessary for such optimizations. They showed that transformations performed on CPS code could be expressed directly in a non-CPS form they called **A-normal form**. Although modern tranformation-based compilers tend to use something like A-normal form, we adopted CPS form in the Tortoise compiler because it is an important illustration of the theme of making implicit structures explicit.

Closure conversion is an important stage in a transformation-based compiler. Johnsson's lambda lifting transformation [Joh85] lifts abstractions to top level after they have been extended with initial parameters for free variables. It uses curried functions that are partially applied to these initial parameters to represent closures. The TORTOISE lifting stage also lifts closed abstractions to top level, but uses a different representation for closures: the closure-passing style invented by Appel and Jim in [AJ88]. Defunctionalization (a notion due to Reynolds [**?**]) was used by Cejtin et al. as the basis for closure conversion in an efficient ML compiler [CJW00]. Selective and lightweight closure conversion were studied by Steckler and Wand [SW97]. The notion of representation pollution was studied by Dimock et al. [DWM+01] in the context of developing a compiler that chooses the representation of a closure depending on how it is used in a program. Sophisticated closure conversion systems rely on **flow analysis** information to determine how procedures are in a program. Nielson, Nielson, and Hankin in [NNH98] provide a good introduction to data flow analysis, control flow analysis, and other program analyses.

For more information on data layout and runtime systems, see Appel's description of ML runtime data structures and support [App90]. For a survey of garbage collection algorithms, see [Wil92]. For a replication-based strategy for garbage collection, see [NOPH92, NO93, NOG93]. [Ape89] shows how static typing can eliminate the need for almost all tag bits in a garbage collected language.

[BCT94] contains a good summary of work on register allocation and spilling. The classic approach to register allocation and spilling involves graph coloring algorithms [CAC+81, Cha82]. See [BWD95] for one approach to managing registers across procedure calls.

Appendix A

A Metalanguage

Man acts as though he were the shaper and master of language, while in fact language remains the master of man.

— "Building Dwelling Thinking," Poetry, Language, Thought (1971), Martin Heidegger

This book explores many aspects of programming languages, including their form and their meaning. But we need some language in which to carry out these discussions. A language used for describing other languages is called a **metalanguage**. This appendix introduces the metalanguage used in the body of the text.

The most obvious choice for a metalanguage is a natural language, such as English, that we use in our everyday lives. When it comes to talking about programming languages, natural language is certainly useful for describing features, explaining concepts at a high level, expressing intuitions, and conveying the big picture. But natural language is too bulky and imprecise to adequately treat the details and subtleties that characterize programming languages. For these we require the precision and conciseness of a mathematical language.

We present our metalanguage as follows. We begin by reviewing the basic mathematics upon which the metalanguage is founded. Next, we explore two concepts at the core of the metalanguage: **functions** and **domains**. We conclude with a summary of the metalanguage notation.

A.1 The Basics

The metalanguage we will use is based on set theory. Since set theory serves as the foundation for much of popular mathematics, you are probably already

familiar with many of the basics described in this section. However, since some of our notation is nonstandard, we recommend that you at least skim this section in order to familiarize yourself with our conventions.

A.1.1 Sets

A **set** is an unordered collection of elements. Sets with a finite number of elements are written by enclosing the written representations of the elements within braces and separating them by commas. So $\{2,\ 3,\ 5\}$ denotes the set of the first three primes. Order and duplication don't matter within set notation, so $\{3,\ 5,\ 2\}$ and $\{3,\ 2,\ 5,\ 5,\ 2,\ 2\}$ also denote the set of the first three primes. A set containing one element, such as $\{19\}$, is called a **singleton**. The set containing no elements is called the **empty set** and is written $\{\}$.

We will assume the existence of certain sets:

$Unit = \{unit\}$;*The standard singleton*
$Bool = \{true, false\}$;*Truth values*
$Int = \{\ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\}$;*Integers*
$Pos = \{1,\ 2,\ 3,\ \ldots\}$;*Positive integers*
$Neg = \{-1,\ -2,\ -3,\ \ldots\}$;*Negative integers*
$Nat = \{0,\ 1,\ 2,\ \ldots\}$;*Natural numbers*
$Rat = \{0,\ 1,\ -1,\ \frac{1}{2},\ -\frac{1}{2},\ \frac{1}{3},\ -\frac{1}{3},\ \frac{2}{3},\ -\frac{2}{3},\ \ldots\}$;*Rationals*
$String = \{$ "", "a", "b",..., "foo",..., "a string",... $\}$;*All text strings*

(The text in *slanted font* following the semi-colon is just a comment and is not a part of the definition. We use this commenting convention throughout the book.) *Unit* (the canonical singleton set) and *Bool* (the set of boolean truth values) are finite sets, but the other examples are infinite. Since it is impossible to write down all elements of an infinite set, we use ellipses ("...") to stand for the missing elements, and depend on the reader's intuition to fill them out. Note that our definition of *Nat* includes 0.

We consider numbers, truth values, and the unit value to be **primitive** elements that cannot be broken down into subparts. Set elements are not constrained to be primitive; sets can contain any structure, including other sets. For example,

$$\{Int,\ Nat,\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\}$$

is a set with three elements: the set of integers, the set of natural numbers, and a set of four elements (one of which is itself a set of two numbers). Here the names *Int* and *Nat* are used as synonyms for the set structure they denote.

Membership is specified by the symbol $\in$ (pronounced "element of" or "in"). The notation $e \in S$ asserts that e is an element of the set S, while $e \notin S$ asserts

that e is not an element of S. (In general, a slash through a symbol indicates the negation of the property denoted by that symbol.) For example,

$$\begin{array}{rcl} 0 & \in & \mathit{Nat} \\ 0 & \notin & \mathit{Neg} \\ \mathit{Int} & \in & \{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\} \\ \mathit{Neg} & \notin & \{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\} \\ 2 & \notin & \{\mathit{Int},\ \mathit{Nat},\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\} \end{array}$$

In the last example, 2 is not an element of the given set even though it is an element of one of that set's elements.

Two sets A and B are **equal** (written $A = B$) if they contain the same elements, i.e., if every element of one is an element of the other. A set A is a **subset** of a set B (written $A \subseteq B$) if every element of A is also an element of B. Every set is a subset of itself, and the empty set is trivially a subset of every set. E.g.,

$$\begin{array}{rcl} \{\} & \subseteq & \{1,\ 2,\ 3\} \subseteq \mathit{Pos} \subseteq \mathit{Nat} \subseteq \mathit{Int} \subseteq \mathit{Rat} \\ \mathit{Nat} & \subseteq & \mathit{Nat} \\ \mathit{Nat} & \not\subseteq & \mathit{Pos} \end{array}$$

Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$. A is said to be a **proper subset** of B (written $A \subset B$) if $A \subseteq B$ and $A \neq B$.

Sets are often specified by describing a defining property of their elements. The **set builder** notation $\{x \mid P_x\}$ (pronounced "the set of all x such that P_x") designates the set of all elements x such that the property P_x is true of x. For example, *Nat* could be defined as $\{n \mid n \in \mathit{Int} \text{ and } n \geq 0\}$. The sets described by set builder notation are not always well-defined. For example, $\{s \mid s \notin s\}$, (the set of all sets that are not elements of themselves) is a famous nonsensical description known as **Russell's paradox**.

Some common binary operations on sets are defined below using set builder notation:

$$\begin{array}{lcll} A \cup B & = & \{x \mid x \in A \text{ or } x \in B\} & \textit{; union} \\ A \cap B & = & \{x \mid x \in A \text{ and } x \in B\} & \textit{; intersection} \\ A - B & = & \{x \mid x \in A \text{ and } x \notin B\} & \textit{; difference} \end{array}$$

The notions of union and intersection can be extended to (potentially infinite) collections of sets. If A is a set of sets, then $\bigcup A$ denotes the union of all of the component sets of A. That is,

$$\bigcup A = \{x \mid \text{ there exists an } a \in A \text{ such that } x \in a\}$$

If A_i is a family of sets indexed by elements i of some given index set I, then

$$\bigcup_{i \in I} A_i = \bigcup \{A_i \mid i \in I\}$$

denotes the union of all the sets A_i as i ranges over I. Intersections of collections of sets are defined in a similar fashion.

Two sets B and C are said to be **disjoint** if and only if $B \cap C = \{\}$. A set of sets $A = \{A_i \mid i \in I\}$ is said to be **pairwise disjoint** if and only if A_i and A_j are disjoint for any distinct i and j in I. A is said to **partition** (or **be a partition of**) a set S if and only if $S = \bigcup_{i \in I} A_i$ and A is pairwise disjoint.

The **cardinality** of a set A (written $|A|$) is the number of elements in A. The cardinality of an infinite set is said to be infinite. Thus $|Int|$ is infinite, but

$$|\{Int,\ Nat,\ \{2,\ 3,\ \{4,\ 5\},\ 6\}\}| = 3$$

Still, there are distinctions between infinities. Informally, two sets are said to be in a **one-to-one correspondence** if it is possible to pair every element of one set with a unique and distinct element in the other set without having any elements left over. Any set that is either finite or in a one-to-one correspondence with *Int* is said to be **countable**. For instance, the set *Even* of even integers is countable because every element $2n$ in *Even* can be paired with n in *Int*. Similarly, *Nat* is obviously countable, and a little thought shows that *Rat* is countable as well. Informally, all countably infinite sets "have the same size." On the other hand, any infinite set that is not in a one-to-one correspondence with *Int* is said to be **uncountable**. Cantor's celebrated diagonalization proof shows that the real numbers are uncountable.[1] Informally, the size of the reals is a much "bigger" infinity than the size of the integers.

The **powerset** of a set A (written $\mathcal{P}(A)$) is the set of all subsets of A. For example,

$$\mathcal{P}(\{1,\ 2,\ 3\}) = \{\{\},\ \{1\},\ \{2\},\ \{3\},\ \{1,\ 2\},\ \{1,\ 3\},\ \{2,\ 3\},\ \{1,\ 2,\ 3\}\}$$

The cardinality of the powerset of a finite set is given by:

$$|\mathcal{P}(A)| = 2^{|A|}$$

In the above example, the powerset has size $2^3 = 8$. The set of all subsets of the integers, $\mathcal{P}(Int)$, is an uncountable set.

[1] A description of Cantor's method can be found in many books on mathematical analysis and computability. We particularly recommend [Hof80].

A.1.2 Tuples

A tuple is an ordered collection of elements. A tuple of length n, called an **n-tuple**, can be envisioned as a structure with n slots arranged in a row, each of which is filled by an element. Tuples with a finite length are written by writing the slot values down in order, separated by commas, and enclosing the result in angle brackets. Thus $\langle 2,\ 3,\ 5\rangle$ is a tuple of the first three primes. Length and order of elements in a tuple matter, so $\langle 2,\ 3,\ 5\rangle$, $\langle 3,\ 2,\ 5\rangle$, and $\langle 3,\ 2,\ 5,\ 5,\ 2,\ 2\rangle$ denote three distinct tuples. Tuples of size 2 through 5 are called, respectively, **pairs**, **triples**, **quadruples**, and **quintuples**. The 0-tuple, $\langle\rangle$, and 1-tuples also exist.

The element of the ith slot of a tuple t can be obtained by **projection**, written $t \downarrow i$. For example, if s is the triple $\langle 2,\ 3,\ 5\rangle$, then $s \downarrow 1 = 2$, $s \downarrow 2 = 3$, and $s \downarrow 3 = 5$. If t is an n-tuple, then $t \downarrow i$ is only well-defined when $1 \leq i \leq n$. Two tuples s and t are **equal** if they have the same length n and $s \downarrow i = t \downarrow i$ for all $1 \leq i \leq n$.

As with sets, tuples may contain other tuples; e.g. $\langle\langle 2,\ 3,\ 5,\ 7\rangle,\ 11,\ \langle 13,\ 17\rangle\rangle$ is a tuple of three elements: a quadruple, an integer, and a pair. Moreover, tuples may contain sets and sets may contain tuples. For instance, the following is a well-defined mathematical structure:

$$\langle\langle 2,\ 3,\ 5\rangle,\ \mathit{Int},\ \{\{2,\ 3,\ 5\},\ \langle 7,\ 11\rangle\}\rangle$$

If A and B are sets, then their **Cartesian product** (written $A \times B$) is the set of all pairs whose first slot holds an element from A and whose second slot holds an element from B. This can be expressed using set builder notation as:

$$A \times B = \{\langle a,\ b\rangle \mid a \in A \text{ and } b \in B\}$$

For example,

$$\{2,\ 3,\ 5\} \times \{7,\ 11\} = \{\langle 2,7\rangle,\ \langle 2,11\rangle,\ \langle 3,7\rangle,\ \langle 3,11\rangle,\ \langle 5,7\rangle,\ \langle 5,11\rangle\}$$
$$\mathit{Nat} \times \mathit{Bool} = \{\langle 0,\mathit{false}\rangle,\ \langle 1,\mathit{false}\rangle,\ \langle 2,\mathit{false}\rangle,\ \ldots,\ \langle 0,\mathit{true}\rangle,\ \langle 1,\mathit{true}\rangle,\ \langle 2,\mathit{true}\rangle,\ \ldots\}$$

If A and B are finite, then $|A \times B| = |A| \cdot |B|$.

The product notion extends to families of sets. If $A_1,\ \ldots\ ,\ A_n$ is a family of sets, then their product (written $A_1 \times A_2 \times \ldots \times A_n$ or $\prod_{i=1}^{n} A_i$) is the set of all n tuples $\langle a_1,\ a_2,\ \ldots,\ a_n\rangle$ such that $a_i \in A_i$. The notation A^n $(= \prod_{i=1}^{n} A)$ stands for the n-fold product of the set A.

A.1.3 Relations

A **binary relation** on A is a subset of $A \times A$.[2] For example, the less-than relation, $<_{Nat}$, on natural numbers is the subset of $Nat \times Nat$ consisting of all pairs of numbers $\langle n, m \rangle$ such that n is less than m:

$$< \;=\; \{\langle 0,1\rangle,\ \langle 0,2\rangle,\ \langle 0,3\rangle,\ \ldots\ ,\ \langle 1,2\rangle,\ \langle 1,3\rangle,\ \ldots\ ,\ \langle 2,3\rangle,\ \ldots\}$$

For a binary relation R on A, the notation $a_1\ R\ a_2$ is shorthand for $\langle a_1, a_2\rangle \in R$. Similarly, the notation $a_1\ \not{R}\ a_2$ means that $\langle a_1, a_2\rangle \notin R$. Thus, the assertion $1 < 2$ is really just another way of saying $\langle 1,2\rangle \in <$, and $3 \not< 2$ is another way of saying $\langle 3,2\rangle \notin <$.

Binary relations are often classified by certain properties. Let R be a binary relation on a set A. Then:

- R is **reflexive** if, for all $a \in A$, $a\ R\ a$.
- R is **symmetric** if, for all $a_1, a_2 \in A$, $a_1\ R\ a_2$ implies $a_2\ R\ a_1$.
- R is **transitive** if, for all $a_1, a_2, a_3 \in A$, $a_1\ R\ a_2$ and $a_2\ R\ a_3$ imply $a_1\ R\ a_3$.
- R is **anti-symmetric** if, for all $a_1, a_2 \in A$, $a_1\ R\ a_2$ and $a_2\ R\ a_1$ implies $a_1 = a_2$. (This assumes the existence of a reflexive, symmetric, and transitive equality relation $=$ on A.)

For example, the $<$ relation on integers is anti-symmetric and transitive, the "is a divisor of" relation on natural numbers is reflexive and transitive, and the $=$ relation on integers is reflexive, symmetric, and transitive.

A binary relation that is reflexive, symmetric and transitive is called an **equivalence relation**. An equivalence relation R on A uniquely **partitions** the elements of A into disjoint **equivalence classes** A_i whose union is A and that satisfy the following: $a_1\ R\ a_2$ if and only if a_1 and a_2 are elements of the same A_i. For example, let $=_{mod3}$ be the "has the same remainder modulo 3" relation on natural numbers. Then it's easy to show that $=_{mod3}$ satisfies the criteria for an equivalence relation. It partitions Nat into three equivalence classes:

$$\begin{aligned} Nat_0 &= \{0,\ 3,\ 6,\ 9,\ \ldots\} \\ Nat_1 &= \{1,\ 4,\ 7,\ 10,\ \ldots\} \\ Nat_2 &= \{2,\ 5,\ 8,\ 11,\ \ldots\} \end{aligned}$$

[2] The notion of a relation can be generalized to arbitrary products, but binary relations are sufficient for our purposes.

The **quotient** of a set A by an equivalence relation R (written A/R) is the set of equivalence classes into which R partitions A. Thus,

$$(Nat \,/\, =_{mod3}) = \{Nat_0,\ Nat_1,\ Nat_2\}$$

There are a number of operations on binary relations that produce new relations. The **n-fold composition** of a binary relation R, written R^n, is the unique relation such that $a_{left}\ R^n\ a_{right}$ if and only if there exist a_i, $1 \leq i \leq n+1$, such that $a_1 = a_{left}$, $a_{n+1} = a_{right}$, and for each i, $a_i\ R\ a_{i+1}$. The **closure** of a binary relation R on A over a specified property P is the smallest relation R_P such that $R \subseteq R_P$ and R_P satisfies the property P. The most important kind of closure we will consider is the **transitive closure** of a relation R, written R^*: $a_{left}\ R^*\ a_{right}$ if and only if $a_{left}\ R^n\ a_{right}$ for some natural number n. For example, the transitive closure of the "is one less than" relation on integers is the "is less than" relation on integers.

A.2 Functions

Functions are a crucial component of our metalanguage. We will devote a fair amount of care to explaining what they are and developing notations to express them.

A.2.1 Definition

Informally, a function is a mapping from an argument to a result. More formally, a function f is a triple of three components:[3]

1. The **source** S of the function (written *src(f)*) — the set from which the argument is taken.

2. The **target** T of the function (written *tgt(f)*) — the set from which the result is taken.

3. The **graph** of a function (written *gph(f)*) — a subset G of $S \times T$ such that each $s \in S$ appears as the first component in no more than one pair $\langle s, t\rangle \in G$.

[3]What we call source and target are commonly called **domain** and **codomain**, respectively. We use different names so as not to cause confusion with the meaning of the term **domain** introduced in Section A.3.

For example, the increment function inc_{Int} on the integers can be defined as

$$inc_{Int} = \langle Int, Int, G_{inc} \rangle$$

where G_{inc} is the set of all pairs $\langle i, i+1 \rangle$ such that $i \in Int$. That is,

$$G_{inc} = \{\ldots\ ,\ \langle -3,-2\rangle,\ \langle -2,-1\rangle,\ \langle -1,0\rangle,\ \langle 0,1\rangle,\ \langle 1,2\rangle,\ \langle 2,3\rangle,\ \ldots\}$$

Note that $src(inc_{Int}) = Int$, $tgt(inc_{Int}) = Int$, and $gph(inc_{Int}) = G_{inc}$.

The **type**[4] of a function specifies its source and target. The type of a function with source S and target T is written $S \to T$. For example, the type of inc_{Int} is $Int \to Int$. The notation $f : S \to T$ means that f has type $S \to T$.

Two functions are **equal** if they are equal as triples — i.e., if their sources, targets, and graphs are respectively equal. In particular, it is not sufficient for their graphs to be equal — they must have the same type as well. For example, consider the following two functions

$$abs_1 = \langle Int, Int, G_{abs} \rangle$$
$$abs_2 = \langle Int, Nat, G_{abs} \rangle$$

where G_{abs} is the set of all pairs $\langle i, i_{abs} \rangle$ such that i is an integer and i_{abs} is the absolute value of i. Then even though abs_1 and abs_2 have the same graph, they are not equal as functions because the type of abs_1, $Int \to Int$, is different from the type of abs_2, $Int \to Nat$.

Many programming languages use the term "function" to refer to a kind of subroutine. To avoid confusion, we will use the term **procedure** for a programming language subroutine, and will reserve the term **function** for the mathematical notion. We wish to carefully distinguish them because they differ in some important respects:

- We often think of procedures as methods, or sometimes even agents, for computing an output from an input. A function doesn't use any method or perform any computation; it doesn't *do* anything. It simply *is* a structure that contains the source, the target, and all input/output pairs.

- We typically view procedures as taking multiple arguments or returning multiple results. But a function always has exactly one argument and exactly one result. However, we will see shortly how these procedural notions can be simulated with functions.

[4] The type of a function is sometimes called its **signature**.

- In addition to returning a value, procedures often have a side-effect — e.g., changing the state of the memory or the status of a screen. There is no equivalent notion of side-effect for a function. However, we will see in Chapter 8 how to use functions to model side-effects in a programming language.

- When viewed in terms of their input/output behavior, procedures can only specify a subset of functions known as the **computable functions**. The most famous example of a non-computable function is the halting function, which maps the text of a program to a boolean that indicates whether or not the program will halt when executed.

The above points do not necessarily apply to the procedural entities in all languages. In particular, the subroutines in so-called **functional programming languages** are very close in spirit to mathematical functions.

A.2.2 Application

The primary operation involving a function is the **application** of the function to an **argument**, an element in its source. The function is called the **operator** of the application, while the argument is called the **operand** of the application. The result of applying an operator f to an operand s is the unique element t in the target of f such that $\langle s, t\rangle$ is in the graph of f. If there is no pair $\langle s, t\rangle$ in the graph of f, then the application of f to s is said to be **undefined**.

A **total function** is one for which application is defined for all elements of its source. If there are source elements for which the function is undefined, the function is said to be **partial**. Most familiar numerical functions are total, but some are partial. The reciprocal function on rationals is partial because it is not defined at 0. And a square root function defined as

$$sqrt = \langle Int, Int, \{\langle i^2, i\rangle \mid i \in Int\}\rangle$$

is partial because it is defined only at perfect squares. For any function f, we use the notation $dom(f)$ to stand for the the source elements at which f is defined. That is,

$$dom(f) = \{s \mid \langle s, t\rangle \in gph(f)\}.$$

For example, $dom(sqrt)$ is the set of perfect squares. A function f is total if $dom(f) = src(f)$ and otherwise is partial. We will use the type notation $S \to T$ to designate the class of total functions and the special notation $S \rightharpoonup T$ to designate the class of partial functions.

It is always possible to turn a partial function into a total function by adding a distinguished element to the target that represents "undefined" and altering the graph to map all previously unmapped members of the source to this "undefined" value. By convention, this element is called **bottom** and is written $\perp$. Using this element, we can define a total reciprocal function whose type is $Rat \rightarrow (Rat \cup \{\perp\})$ and whose graph is:

$$\{\langle 0, \perp\rangle\} \cup \{\langle q, 1/q\rangle \mid q \in Rat, q \neq 0\}$$

Bottom plays a crucial role in the explanation of fixed points in Chapter 5.

We use the juxtaposition $f\ s$ to denote the application of a function f to an element s.[5] For instance, the increment of 3 is written $inc_{Int}\ 3$. Parentheses are used to structure nested applications. Thus,

$$inc_{Int}\ \ (inc_{Int}\ \ 3)$$

expresses the increment of the increment of 3. In the metalanguage, parentheses that don't affect the application structure can always be added without changing the meaning of an expression.[6] The following is equivalent to the above:

$$((inc_{Int})\ \ (inc_{Int}\ \ (3)))$$

By default, function application associates to the left, so that the expression $a\ b\ c\ d$ parses as $(((a\ b)\ \ c)\ \ d)$.

The **type** of an application is the type of the target of the operator of the application. For example, if $sqr : Int \rightarrow Nat$, then $(sqr\ \ -3) : Nat$ (pronounced "$(sqr\ \ -3)$ has type Nat"). An application is **well-typed** only when the type of the operand is the same as the source of the operator type. (The type of a number like 3 depends on context: it can be considered a natural, an integer, a rational, etc.) For example, if f is a function with type $Nat \rightarrow Int$, then the application $(f\ \ -3)$ is not well-typed because $-3 \notin Nat$. However, the application $(f\ \ (sqr\ \ -3))$ is well-typed. In our metalanguage, an application is only legal if it is well-typed.

[5] The reader may find it strange that we depart from the more traditional notation for application, which is written $f(s)$ for single arguments, and $f(s_1, s_2, \ldots, s_n)$ for multiple arguments. The reason is that in the traditional notation, f is usually restricted to be a *function name*, whereas we will want to allow the function position of an application to be any metalanguage expression that stands for a function. Application by juxtaposition is a superior notation for handling this more general case because it visually distinguishes less between the function position and the argument position.

[6] This contrasts with s-expression grammars, as in Lisp-like programming languages, in which no parentheses are optional.

A.2.3 More Function Terminology

For any set A, there is an **identity** function id_A that maps every element of A to itself:

$$id_A = \langle A, A, \{\langle a, a\rangle \mid a \in A\}\rangle$$

For each element a of a set A, there is a **constant** function $const_a$ that maps every element of A to a:

$$const_a = \langle A, A, \{\langle a', a\rangle \mid a' \in A\}\rangle$$

For any set B such that $B \subseteq A$, there is an **inclusion** function $B{\hookrightarrow}A$ that maps every element of B to the same element in the larger set:

$$B \hookrightarrow A = \langle B, A, \{\langle b, b\rangle \mid b \in B\}\rangle$$

Inclusion functions are handy for making a metalanguage expression the "right type." For example, if *sqr* has type $Int \to Nat$, then the expression

$$(sqr\ \ (sqr\ \ -3))$$

is not well-typed, but the expression

$$(sqr\ \ (Nat \hookrightarrow Int\ \ (sqr\ \ -3)))$$

is well-typed.

If $f : A \to B$ and $g : B \to C$, then the **composition** of g and f, written $g \circ f$, is a function of type $A \to C$ defined as follows:

$$(g\ \circ\ f)\ a = (g\ \ (f\ \ a))\,, \text{ for all } a \in A$$

The composition function[7] is associative, so that

$$f\ \circ\ g\ \circ\ h = (f\ \circ\ g)\ \circ\ h = f\ \circ\ (g\ \circ\ h)$$

If $f : A \to A$ then the **n-fold composition** of f, written f^n, is

$$\underbrace{f\ \circ\ f\ \circ\ \ldots\ \circ\ f}_{n \text{ times}}$$

f^0 is defined to be the identity function on A. Because of the associativity of composition, $f^n\ \circ\ f^m = f^{n+m}$.

[7] There is not a single composition function, but really a family of composition functions indexed by the types of their operands.

The **image** of a function is that subset of the target that the function actually maps to. That is, for $f : S \to T$, the image of f is

$$\{t \mid \text{ there exists an } s \text{ such that } (f\ s) = t\}$$

A function is **injective** when no two elements of the source map to the same target element, i.e., when $(f\ d_1) = (f\ d_2)$ implies $d_1 = d_2$. A function is **surjective** when every element in the target is the result of some application, i.e., when the image is equal to the target. A function is **bijective** if it is both injective and surjective. Two sets A and B are said to be in a **one-to-one correspondence** if there exists a bijective function with type $A \to B$.

A.2.4 Higher-Order Functions

The sources and targets of functions are not limited to familiar sets like numbers, but may be sets of sets, sets of tuples, or even sets of functions. Functions whose sources or targets themselves include functions are called **higher-order functions**.

As a natural example of a function that returns a function, consider a function *make-expt* that, given a power, returns a function that raises numbers to that power. The type of *make-expt* is

$$Nat \to (Nat \to Nat)$$

That is, the source of *make-expt* is *Nat*, and the target of *make-expt* is the set of all functions with type $Nat \to Nat$. The graph of *make-expt* is:

$$\begin{array}{l}\{\langle 0, \langle Nat, Nat, \{\langle 0,1\rangle,\ \langle 1,1\rangle,\ \langle 2,1\rangle,\ \langle 3,1\rangle,\ \langle 4,1\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 1, \langle Nat, Nat, \{\langle 0,0\rangle,\ \langle 1,1\rangle,\ \langle 2,2\rangle,\ \langle 3,3\rangle,\ \langle 4,4\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 2, \langle Nat, Nat, \{\langle 0,0\rangle,\ \langle 1,1\rangle,\ \langle 2,4\rangle,\ \langle 3,9\rangle,\ \langle 4,16\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 3, \langle Nat, Nat, \{\langle 0,0\rangle,\ \langle 1,1\rangle,\ \langle 2,8\rangle,\ \langle 3,27\rangle,\ \langle 4,64\rangle,\ \ldots\}\rangle\rangle,\\ \ \ldots\ \}\end{array}$$

That is, (*make-expt* 0) denotes a function that maps every number to 1, (*make-expt* 1) denotes the identity function on natural numbers, (*make-expt* 2) denotes the squaring function, (*make-expt* 3) denotes the cubing function, and so on.

As an example of a function that takes functions as arguments, consider the function *apply-to-five* that takes a function between natural numbers and returns the value of this function applied to 5. The type of *apply-to-five* is

$$(Nat \to Nat) \to Nat$$

and its graph is

$$\{\langle id_{Nat}, 5\rangle,\ \langle inc_{Nat}, 6\rangle,\ \langle dec_{Nat}, 4\rangle,\ \langle square_{Nat}, 25\rangle,\ \langle cube_{Nat}, 125\rangle,\ \ldots,\ \langle\langle Nat, Nat, \{\ldots, \langle 5, n\rangle, \ldots\}\rangle, n\rangle,\ \ldots\}$$

where inc_{Nat}, dec_{Nat}, $square_{Nat}$, and $cube_{Nat}$ denote, respectively, the incrementing function, decrementing function, squaring function, and cubing function on natural numbers.

We make extensive use of higher order functions throughout this book.

A.2.5 Multiple Arguments and Results

We noted before that every mathematical function has a single argument and a single result. Yet, as programmers, we are used to thinking that many familiar procedures, like addition and multiplication, have multiple arguments. Sometimes we think of procedures as returning multiple results; for instance, a division procedure can profitably be viewed as returning both a quotient and a remainder. How can we translate these programming language notions into the world of mathematical functions?

A.2.5.1 Multiple Arguments

There are two common approaches for handling multiple arguments:

1. The multiple arguments can be boxed up into a single argument tuple. For instance, under this approach, the binary addition function $+_{Nat}$ on natural numbers would have type

   $$(Nat \times Nat) \rightarrow Nat$$

   and would have the following graph:

   $$\{\langle\langle 0,0\rangle, 0\rangle,\ \langle\langle 0,1\rangle, 1\rangle,\ \langle\langle 0,2\rangle, 2\rangle,\ \langle\langle 0,3\rangle, 3\rangle,\ \ldots,$$
   $$\langle\langle 1,0\rangle, 1\rangle,\ \langle\langle 1,1\rangle, 2\rangle,\ \langle\langle 1,2\rangle, 3\rangle,\ \langle\langle 1,3\rangle, 4\rangle,\ \ldots,$$
   $$\langle\langle 2,0\rangle, 2\rangle,\ \langle\langle 2,1\rangle, 3\rangle,\ \langle\langle 2,2\rangle, 4\rangle,\ \langle\langle 2,3\rangle, 5\rangle,\ \ldots,$$
   $$\ldots\}$$

   Then an application of the addition function to 3 and 5, say, would be written as $(+_{Nat}\ \langle 3, 5\rangle)$.

2. A function of multiple arguments can be represented as a higher-order function that takes the first argument and returns a function that takes the rest of the arguments. This approach is named **currying**, after its inventor, Haskell Curry. Under this approach, the binary addition function $+_{Nat}$ on natural numbers would have type

   $$Nat \rightarrow (Nat \rightarrow Nat)$$

and would have the following graph:

$$\begin{array}{l}\langle\langle 0, \langle Nat, Nat, \{\langle 0,0\rangle,\ \langle 1,1\rangle,\ \langle 2,2\rangle,\ \langle 3,3\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 1, \langle Nat, Nat, \{\langle 0,1\rangle,\ \langle 1,2\rangle,\ \langle 2,3\rangle,\ \langle 3,4\rangle,\ \ldots\}\rangle\rangle,\\ \ \langle 2, \langle Nat, Nat, \{\langle 0,2\rangle,\ \langle 1,3\rangle,\ \langle 2,4\rangle,\ \langle 3,5\rangle,\ \ldots\}\rangle\rangle,\\ \ \ldots\ \rangle\end{array}$$

When $+_{Nat}$ is applied to n, the resulting value is the increment-by-n function. So, given 0, it returns the identity function on natural numbers; given 1, it returns the increment-by-one function; given 2, it returns the increment-by-two function; and so on. With currying, the application of $+_{Nat}$ to 3 and 5 is written as $((+_{Nat}\ 3)\ 5)$ or as $(+_{Nat}\ 3\ 5)$, (relying on the left-associativity of application).

In the currying approach, functions like $+_{Nat}$ or *make-expt* can be viewed differently according to the context in which they are used. Sometimes, we may like to think of them as functions that "take two arguments." Other times, it is helpful to view them as functions that take a single argument and return a function. Of course, they are exactly the same function in both cases; the only difference is the glasses through which we're viewing them.

Throughout this book, we will use the second approach, currying, as our standard method of handling multiple arguments. We will assume that standard binary numerical function and predicate names, such as $+$, $-$, $\times$, $/$, $<$, $=$, $>$, denote curried functions with type $N \to (N \to N)$ or $N \to (N \to Bool)$, where N is a numerical set like the naturals, integers, or rationals. When disambiguation is necessary, the name of the function or predicate will be subscripted with an indication of what numerical source is intended. So, $+_{Nat}$ is addition on the naturals, while $+_{Int}$ is addition on the integers, etc. For example, $(\times_{Int}\ 2)$ denotes a doubling function on integers.

Since infix notation for standard binary functions is so much more familiar than the curried prefix form, we will typically use infix notation when both arguments are present. Thus, the expression $(3 +_{Int} 4)$ is synonymous with $(+_{Int}\ 3\ 4)$.

We will also assume the existence of a curried three-argument conditional function if_S with type

$$Bool \to (S \to (S \to S))$$

that returns the second argument if the first argument is *true*, and returns the third argument if the first argument is *false*. E.g.,

$$\begin{array}{l}(if_{Nat}\ \ (1 =_{Nat} 1)\ \ 3\ \ 4) = 3\\ (if_{Nat}\ \ (1 =_{Nat} 2)\ \ 3\ \ 4) = 4\end{array}$$

A.2.5.2 Multiple Results

The handling of multiple return values parallels the handling of multiple arguments. Again, there are two common approaches:

1. Return a tuple of the results. Under this approach, a quotient-and-remainder function *quot&rem* on natural numbers would have type

$$Nat \to (Nat \to (Nat \times Nat)).$$

   Some sample applications using this approach:

$$\begin{array}{l}(quot\&rem\ 14\ 4) = \langle 3, 2\rangle \\ (quot\&rem\ 21\ 5) = \langle 4, 1\rangle\end{array}$$

2. Suppose the goal is to define a function f of k arguments that "returns" n results. Instead define a function f' that accepts the k arguments that f would, but in addition also takes a special extra argument called a **receiver**. The value returned by f' is the result of applying the receiver to the n values we want f to return. The receiver indicates how the n returned values can be combined into a single value. For example:

$$\begin{array}{l}(quot\&rem\ 14\ 4\ -_{Int}) = (3 -_{Int} 2) = 1 \\ (quot\&rem\ 14\ 4\ \times_{Int}) = (3 \times_{Int} 2) = 6\end{array} \quad l$$

   In these examples the type of *quot&rem* is

$$Int \to (Int \to ((Int \to (Int \to Int)) \to Int))$$

   In general, the notation

$$(f'\ a_1\ \ldots\ a_k\ r)$$

   can be pronounced "Apply r to the n results of the application of f to $a_1 \ldots a_k$." Note how this pronunciation mentions the f upon which f' is based.

We will use both of these approaches for returning multiple values. The second approach probably seems mighty obscure and bizarre at first reading, but it will prove to be a surprisingly useful technique in many situations. In fact, it is just a special case of a more general technique called **continuation-passing style** that is studied in Chapters 9 and 17.

A.2.6 Lambda Notation

Up to this point, the only notation we've had to express new functions is a combination of tuple notation and set builder notation. For example, the squaring function on natural numbers can be expressed by the notation:

$$square = \langle Nat, Nat, \{\langle n, n^2\rangle \mid n \in Nat\}\rangle$$

This notation is cumbersome for all but the simplest of functions.

For our metalanguage, we will instead adopt **lambda notation** as a more compact and direct notation for expressing functions. The lambda notation version of the above *square* function is:

$$square : Nat \to Nat \;=\; \lambda n\,.\;(n \times_{Nat} n)$$

Here, the source and target of the function are encoded in the type that is attached to the function name. The Greek lambda symbol, λ, introduces an **abstraction** that specifies the graph of the function, i.e., how the function maps its argument to a result. An abstraction has the form

$$\lambda\ formal\,.\ body$$

where *formal* is a **formal parameter** variable that ranges over the source of the function, and *body* is a metalanguage expression, possibly referring to the formal parameter, that specifies a result in the target of the function. The abstraction $\lambda\ formal\,.\ body$ is pronounced "A function that maps *formal* to *body*."

For a function with type $A \to B$, an abstraction defines the graph of the function to be the following subset of $A \times B$:

$$\{\langle formal, body\rangle \mid formal \in A \text{ and } body \text{ is defined}\}$$

Thus, the abstraction $\lambda n\,.\;(n \times_{Nat} n)$ specifies the graph:

$$\{\langle n, (n \times_{Nat} n)\rangle\}$$

The condition that *body* be defined (i.e., is not undefined) handles the fact that the function defined by the abstraction may be partial. For example, consider a reciprocal function defined as:

$$recip : Rat \to Rat \;=\; \lambda q\,.\,(1\,/_{Rat}\,q)$$

The graph of *recip* defined by the abstraction contains no pair of the form $\langle 0, i_0\rangle$ because $(1\,/_{Rat}\,0)$ is undefined.

An important advantage of lambda notation is that it facilitates the expression of higher-order procedures. For example, suppose that *expt* is a binary exponentiation function on natural numbers. Then the *make-expt* function from Section A.2.4 can be expressed succinctly as:

$$\textit{make-expt} : \textit{Nat} \to (\textit{Nat} \to \textit{Nat}) \;=\; \lambda n_1 \,.\, (\lambda n_2 \,.\, (\textit{expt}\;\; n_2 \;\; n_1))$$

The abstraction $\lambda n_1 \,.\, \ldots$ can be read as "The function that maps n_1 to an exponentiating function that raises its argument to the n_1 power." Similarly, the *apply-to-five* function can be concisely written as:

$$\textit{apply-to-five} : (\textit{Nat} \to \textit{Nat}) \to \textit{Nat} \;=\; \lambda f \,.\, (f \;\; 5)$$

By the type of *apply-to-five*, the argument f is constrained to range over functions with the type $\textit{Nat} \to \textit{Nat}$. The lambda notation says that such a function f should map to the result of applying f to 5.

Like applications, all abstractions in our metalanguage must be **well-typed**. An abstraction is well-typed if there is a unique way to assign a type to its formal parameter such that its body expression is well-typed. If the type of body is T when the formal parameter is assumed to have type S, then the abstraction has type $S \to T$.

The type of an abstraction is often explicitly specified, as in the above definitions of *square*, *make-expt*, and *apply-to-five*. If the type of an abstraction has been explicitly specified to be $S \to T$, then the type of the formal parameter must be S. For example, in the definition

$$\textit{square} : \textit{Nat} \to \textit{Nat} \;=\; \lambda n \,.\, (n \times_{\textit{Nat}} n)$$

the formal parameter n has type *Nat* within the body expression $(n \times_{\textit{Nat}} n)$. This body expression has type *Nat* and so is well-typed. On the other hand, the definition

$$\textit{dec} : \textit{Nat} \to \textit{Nat} \;=\; \lambda n \,.\, (-1 +_{\textit{Nat}} n)$$

is not well-typed because in the body of the abstraction $+_{\textit{Nat}}$ is applied to an argument, -1, that is not of type *Nat*.

The type of a formal parameter can always be extracted from a type explicitly specified for an abstraction. However, even when the type of the abstraction is not supplied, it is often possible to determine the type of a formal parameter based on constraints implied by the body expression. For example, in the abstraction

$$\lambda x \,.\, (1 +_{\textit{Int}} x)$$

the formal parameter x must be of type *Int* because the application involving $+_{Int}$ is only legal when both arguments are elements of type *Int*. However, sometimes there aren't enough constraints to unambiguously reconstruct the types of formal parameters. For example, in

$$\lambda f \,.\, (f\ \ 5)$$

the type of the formal parameter f must be of the form $N \to T$, where N is a numeric type; but there are many choices for N, and T is totally unconstrained. This is a case where an explicit type must be given to the abstraction.

An abstraction of type $S \to T$ can appear anywhere that an expression of type $S \to T$ is allowed. For example, an application of the squaring function to the result of adding 2 and 3 is written:

$$((\lambda n \,.\, (n \times_{Nat} n))\ \ (2 +_{Nat} 3))$$

Such an application can be simplified by any manipulation that maintains the meaning of the expression. For instance:

$$\begin{aligned}&((\lambda n \,.\, (n \times_{Nat} n))\ \ (2 +_{Nat} 3))\\ &= ((\lambda n \,.\, (n \times_{Nat} n))\ \ 5)\\ &= (5 \times_{Nat} 5)\\ &= 25\end{aligned}$$

In the next to last step above, the number 5 was substituted for the formal n in the body expression $(n \times_{Nat} n)$. This step is justified by the meaning of application in conjunction with the function graph specified by the abstraction. As another sample application, consider:

$$\begin{aligned}&(\textit{make-expt}\ \ 3)\\ &= ((\lambda n_1 \,.\, (\lambda n_2 \,.\, (\textit{expt}\ \ n_2\ \ n_1)))\ \ 3)\\ &= \lambda n_2 \,.\, (\textit{expt}\ \ n_2\ \ 3)\end{aligned}$$

In this case, the result of the application is a cubing function.

Often the same abstraction can be used to define many different functions. For example, the expression $\lambda a \,.\, a$ can be used to define the graph of any identity or inclusion function. Because the variable a ranges over the source, though, the resulting graphs are different for each source. A family of functions defined by the same abstraction is said to be a **polymorphic function**. We will often parameterize such functions over a type or types to take advantage of their common structure. Thus, we can define the polymorphic identity function as

$$\textit{identity}_A : A \to A \ = \ \lambda a \,.\, a$$

where the subscript A means that $identity_A$ defines a family of functions indexed by the type A. We specify a particular member of the family by fixing the subscript to be a known type. So $identity_{Int}$ is the identity function on integers, and $identity_{Bool}$ is the identity function on booleans.

There are several conventions that are used to make lambda notation more compact:

- It is common to abbreviate nested abstractions by collecting all the formal parameters and putting them between a single λ and dot. Thus,

  $$\lambda a_1 \,.\, \lambda a_2 \,.\, \ldots \lambda a_n \,.\, body$$

  can also be written as

  $$\lambda a_1 a_2 \ldots a_n \,.\, body$$

  This abbreviation promotes the view that curried functions accept "multiple arguments": $\lambda a_1 a_2 \ldots a_n \,.\, body$ can be considered a specification for a function that "takes n arguments."

- Formal parameter names are almost always single characters, perhaps annotated with a subscript or prime. This means that whitespace separating such names can be removed without resulting in any ambiguity. In combination with the left-associativity of application, these conventions allow $\lambda a\ b\ c\,.\,((b\ c)\ \ a)$ to be written as $\lambda abc\,.\,bca$.

- Nested abstractions are potentially ambiguous since it's not always apparent where the body of each abstraction ends. For example, the abstraction $\lambda x\,.\,\lambda y\,.\,yx$ could be parsed either as $\lambda x\,.\,\lambda y\,.\,(yx)$ or as $\lambda x\,.\,(\lambda y\,.\,y)x$. The following disambiguating convention is used in such cases: the body of an abstraction is assumed to extend as far right as explicit parentheses allow. By this convention, $\lambda x\,.\,\lambda y\,.\,yx$ means $\lambda x\,.\,(\lambda y\,.\,(yx))$.

A.2.7 Recursion

Using lambda notation, it is possible to write **recursive** function specifications: functions that are directly or indirectly defined in terms of themselves. For example, the factorial function *fact* on natural numbers can be defined as:

$$fact : Nat \to Nat \ = \ \lambda n\,.\,(if_{Nat}\ \ (n =_{Nat} 0)\ \ 1\ \ (n \times_{Nat}\ (fact\ \ (n -_{Nat} 1))))$$

We can argue that *fact* is defined on all natural numbers based on the principle of mathematical induction. That is, for the base case of an argument equal to 0, the definition clearly specifies the value of *fact* to be 1. For the inductive

case, assume that *fact* is defined for the argument m. Then, according to the definition, the value of $(fact\ (m+1))$ is $((m+1) \times_{Nat} (fact\ m))$. But by the assumption that $(fact\ m)$ is defined, this expression has a clear meaning. So $(fact\ (m+1))$ is also defined. By induction, *fact* is defined on every element of *Nat*, so the above definition determines a unique total function.

There are many recursive definitions for which the above kind of inductive argument fails. Consider the definition of the *strange* function given below:

$$strange : Nat \to Nat = \lambda n\ .\ (if_{Nat}\ \ (even?_{Nat}\ \ n)\ \ 0\ \ (strange\ \ (n +_{Nat} 2)))$$

(Assume that $even?_{Nat}$ is a predicate that tests whether its argument is even.) Clearly the function *strange* maps every even number to 0. But what does it map odd numbers to? Induction does not help us because the argument never gets smaller. If we think in terms of function graphs, then we see that for any natural number c, the above definition is consistent with a graph of the form

$$\{\langle 2n, 0\rangle \mid n \in Nat\} \cup \{\langle 2n+1, c\rangle \mid n \in Nat\}$$

So the specification for *strange* is ambiguous; it designates any of an infinite number of function graphs!

The *strange* example illustrates that recursive definitions need to be handled with extreme care. For now, we will assume that the only case in which a recursive definition has a well-defined meaning is one for which it is possible to construct an inductive argument of the sort used for *fact*. Chapter 5 presents a technique for determining the meaning of a broad class of recursive definitions that includes functions like *strange*.

A.2.8 Lambda Notation is not Lisp!

Those familiar with a dialect of the Lisp programming language may notice a variety of similarities between lambda notation and Lisp. (Those unfamiliar with Lisp may safely skip this section.) Although Lisp is in many ways related to our metalanguage, we emphasize that there are some crucial differences:

- Our metalanguage requires all expressions to be well-typed. In particular, source and target types must be provided for every abstraction. Most dialects of Lisp, on the other hand, have no notion of a well-typed expression, and they provide no mechanism for specifying argument and result types for procedures.[8]

[8] The FX language [GJSO92] is a notable exception.

- Most Lisp-like languages support procedures that handle multiple arguments. Because abstractions specify mathematical functions, they always take a single argument. However, the notion of multiple arguments can be simulated by currying or tupling.

- Every parenthesis in a Lisp expression is required, but parentheses are only strictly necessary in our lambda notation to override the default way in which an expression is parsed. Of course, extra parentheses may be added to clarify a metalanguage expression.

- Lisp dialects are characterized by evaluation strategies that determine details like which subexpressions of a conditional are evaluated and when argument expressions are evaluated relative to the evaluation of a procedure body. Our metalanguage, on the other hand, is not associated with any notion of a dynamic evaluation strategy. Rather, it is just a notation to describe the graph of a function, i.e., a set of argument/result pairs. Any reasoning about an abstraction is based on the structure of the graph it denotes.

  For example, compare the metalanguage abstraction

  $$\lambda a\ .\ (\mathit{if}_{Nat}\ \ (\mathit{even?}_{Nat}\ \ a)\ \ (a +_{Nat} 1)\ \ (a \times_{Nat} 2))$$

  with the similar Lisp expression

  ```
  (lambda (a) (if (even? a) (+ a 1) (* a 2)))
  ```

  In the case of Lisp, only one branch of the conditional is evaluated for any given argument `a`; if `a` is even, then `(+ a 1)` is evaluated, and if it's odd, `(* a 2)` is evaluated. In the case of the metalanguage, the value of the function for any argument a is the result of applying the *if* function to the three arguments $(\mathit{even?}_{Nat}\ \ a)$, $(a +_{Nat} 1)$, and $(a \times_{Nat} 2)$. Here there is no notion of evaluation, no notion that some event does or does not happen, and no notion of time. The expression simply designates the mathematical function:

  $$\langle Nat, Nat, \{\langle 0,1\rangle,\ \langle 1,2\rangle,\ \langle 2,3\rangle,\ \langle 3,6\rangle,\ \langle 4,5\rangle,\ \langle 5,10\rangle,\ \ldots\}\rangle$$

  In fact, a metalanguage abstraction can be viewed as simply a structured name for a particular function.

Although there are many differences between Lisp and lambda notation, the two obviously share some important similarities. Some **functional programming languages** have features that are even more closely patterned after

lambda notation. (The FL language presented in Chapter 6 is an example.) However, our purpose for introducing lambda notation here is to have a convenient notation for expressing mathematical functions, not for writing programs. The relationship between mathematical functions and programs is the essence of semantics, which is studied in the main text of the book.

A.3 Domains

A.3.1 Motivation

Sets and set-theoretic functions have too simple a structure to model some important aspects of the semantics of programming languages. Yet, we would like to proceed with the simplifying assumption that sets are adequate for our purposes until the need for more structure arises. And when we do augment sets with more structure (see Chapter 5), we would prefer not to throw away all of the concepts and notations developed up to that point and start from scratch.

To protect against such a disaster, we will use the same technique that good programmers use to guarantee that their code can be easily modified: **abstraction**.[9] The essence of abstraction is constructing an **abstraction barrier** or **interface** that clearly separates behavior from implementation. In programming, an interface usually consists of a collection of procedures that manipulate elements of an abstract data type. The data type is abstract in the sense that it can only be manipulated by the procedures in the interface; its internal representation details are hidden. The power of abstraction is that changes to the representation of a data type are limited to the implementation side of the barrier; as long as the interface specification is maintained, no client of the abstraction needs to be modified.

We introduce an abstract structure called a **domain** that will serve as our basic entity for modeling programming languages. Domains are set-like structures that have constituent elements, but may have other structure as well. The interface to domains is specified by a collection of **domain constructors** introduced below. In our initial naïve implementation, domains are sets. In Chapter 5, however, we will change this implementation by extending the sets with additional structure.

Together, domains and domain constructors define a simple domain language. The language comes equipped with a collection of fundamental building blocks called **primitive domains**. These cannot be decomposed into simpler

[9] Note that this use of the term "abstraction" is different from that used in the previous section, where it meant a metalanguage expression that begins with a proc.

domains. Domain constructors build more complex domains from simpler ones. The resulting **compound domains** can be decomposed into the parts out of which they were made.

In the naïve implementation of domains, primitive domains are sets whose elements have no structure. That is, the elements of primitive domains may be items like numbers, truth values, or the unit value; but they may not be tuples, sets, or functions. Examples of primitive domains include *Unit*, *Bool*, *Int*, *Nat*, and *Rat*.

Compound domains are built by four domain constructors: $\times$, $+$, *, and $\rightarrow$. We shall study these in turn.

A.3.2 Product Domains

The product of two domains, written $D_1 \times D_2$, is the domain version of a Cartesian product. Elements of a compound domain are created by an appropriate **constructor** function. In the case of products, the constructor *tuple* creates elements of the product domain, which are called **tuples**. We will extend the type notation $d : D$ to indicate that d is an element of the domain D. If $d_1 : D_1$ and $d_2 : D_2$ then

$$\big(tuple_{D_1,D_2}\ d_1\ d_2\big) : D_1 \times D_2$$

The subscripts on *tuple* emphasize that it is really a family of functions indexed by the component domains. For example, $tuple_{Nat,Bool}$ and $tuple_{Int,Int}$ both serve to pair elements, but the fact that they have different sources, targets, and graphs makes them different functions.

We will abbreviate $\big(tuple_{D_1,D_2}\ d_1\ d_2\big)$ as $\langle d_1,\ d_2\rangle_{D_1,D_2}$, and will drop the subscripts when they are clear from context. For example, the product of *Nat* and *Bool* technically is

$$\begin{aligned} &Nat \times Bool = \\ &\quad \{\big(tuple_{Nat,Bool}\ 0\ false\big),\ \big(tuple_{Nat,Bool}\ 1\ false\big),\ \big(tuple_{Nat,Bool}\ 2\ false\big),\ \ldots, \\ &\quad\ \big(tuple_{Nat,Bool}\ 0\ true\big),\ \big(tuple_{Nat,Bool}\ 1\ true\big),\ \big(tuple_{Nat,Bool}\ 2\ true\big),\ \ldots\} \end{aligned}$$

but we will usually write it as

$$\begin{aligned} Nat \times Bool = \{&\langle 0, false\rangle,\ \langle 1, false\rangle,\ \langle 2, false\rangle,\ \ldots, \\ &\langle 0, true\rangle,\ \langle 1, true\rangle,\ \langle 2, true\rangle,\ \ldots\} \end{aligned}$$

Domains of n-tuples (known as n-ary products) are written

$$\prod_{i=1}^{n} D_i = D_1 \times D_2 \times \cdots \times D_n = \{\langle d_1, d_2, \ldots, d_n\rangle_{D_1,D_2,\ldots,D_n} \mid d_i : D_i\}$$

The notation D^n stands for the product of n copies of D.

Every product domain $\prod_{i=1}^{n} D_i$ comes equipped with n **projection functions**

$$Proji_{D_1,\ldots,D_n} : (D_1 \times \ldots \times D_n) \to D_i$$

to extract the ith element from an n-tuple:

$$Proji_{D_1,\ldots,D_n}\ \langle d_1,\ldots,d_n\rangle_{D_1,\ldots,D_n} = d_i,\ 1 \le i \le n$$

For example,

$$\begin{aligned} Proj1_{Nat,Bool}\ \langle 19,\ true\rangle &= 19 \\ Proj2_{Nat,Bool}\ \langle 19,\ true\rangle &= true \end{aligned}$$

Again, the subscripts indicate that for each i, $Proji$ is a family of functions indexed by the component domains of the tuple being operated on. They will be omitted when they are clear from context.

Notice that we have overloaded the notation $\langle\ldots\rangle$, which may now denote either a set-theoretic tuple or a domain-theoretic one. We have done this because in the simple implementation of domains as sets, product domains simply *are* set-theoretic Cartesian products, and set-theoretic tuples *are* tuples. However, thinking in terms of a concrete implementation for domains can be somewhat dangerous. Product domains are really defined only by the behavior of *tuple* and $Proji$, which must satisfy the following two properties:

1. $Proji_{D_1,\ldots,D_n}\ \left(tuple_{D_1,\ldots,D_n}\ d_1\ \ldots\ d_n\right) = d_i,\ 1 \le i \le n$
2. $tuple_{D_1,\ldots,D_n}\ \left(Proj1_{D_1,\ldots,D_n}\ d\right)\ \ldots\ \left(Projn_{D_1,\ldots,D_n}\ d\right) = d,$
   $d : \prod_{i=1}^{n} D_i$

Any implementation of *tuple* and $Proji$ that satisfies these two properties is a valid implementation of products for domains. For example, it's perfectly legitimate to define $tuple_{Nat,Bool}$ by

$$tuple_{Nat,Bool}\ n\ b = \langle b, n\rangle,$$

where the order of elements in the concrete (set-theoretic) representation is reversed, as long as $Proji_{Nat,Bool}$ is defined consistently:

$$\begin{aligned} Proj1_{Nat,Bool}\ \langle b, n\rangle &= n \\ Proj2_{Nat,Bool}\ \langle b, n\rangle &= b \end{aligned}$$

From here on, and in the body of the text, the $\langle\ldots\rangle$ notation will by default denote domain-theoretic tuples rather than set-theoretic tuples.

Since writing out compound domains in full can be cumbersome, it is common to introduce synonyms for them via a **domain definition** of the form

$$name = compound\text{-}domain$$

For example, the domain definitions

$$Vector = Int \times Int$$
$$Circle = Vector \times Int \times Bool$$

introduces the name *Vector* as a synonym for a domain of pairs of integers and the name *Circle* as a synonym for a domain of triples whose components represent the state of a graphical circle object: the position of its center (a pair of integers), its radius (an integer), and a flag indicated whether or not it is filled (a boolean). Domain definitions are often used merely to introduce more mnemonic names for domains. The following set of domain definitions is equivalent to the set above:

$$Vector = X\text{-}coord \times Y\text{-}coord$$
$$X\text{-}coord = Int$$
$$Y\text{-}coord = Int$$
$$Circle = Position \times Radius \times Filled?$$
$$Position = Vector$$
$$Radius = Int$$
$$Filled? = Bool$$

Domain equality is purely structural and has nothing to do with names. Thus, the assertion $Position = (Int \times Int)$ is true because both descriptions designate the domain of pairs of integers.[10]

A.3.3 Sum Domains

Sum domains are analogous to variant records and unions in programming languages. The sum of two domains, written $D_1 + D_2$, is a domain that is the **disjoint union** of the two domains. A disjoint union differs from the usual set union in that each element of the union is effectively "tagged" to indicate which component set it comes from. An element of a sum domain, which we will call a **oneof**, is built by an **injection function**

$$Inj\, i_{D_1,D_2} : D_i \rightarrow (D_1 + D_2)$$

Here, i, which can be either 1 or 2, indicates which component domain the element is from.

[10] It may seem confusing that the equality symbol, $=$, is used both to test domains for equality and to define new domain names. But this confusion is standard in mathematics. In the first case, it is assumed that the meaning of all names is known, and $=$ asserts that the left and right hand sides are equal. In the second case, it is assumed that the meaning of the left hand names are unknown, and the equations are solved to make the $=$ assertions true. In the examples above, the equations are trivial to solve, but domain equations with recursion can be difficult to solve (see Chapter 5).

A sum domain contains all oneofs that can be constructed from its component domains. For example,

$$\begin{aligned} Nat + Int = \{&\left(Inj\,1_{Nat,Int}\ \ 0\right), \left(Inj\,1_{Nat,Int}\ \ 1\right), \left(Inj\,1_{Nat,Int}\ \ 2\right), \ldots\ , \\ &\left(Inj\,2_{Nat,Int}\ -2\right), \left(Inj\,2_{Nat,Int}\ -1\right), \left(Inj\,2_{Nat,Int}\ \ 0\right), \left(Inj\,2_{Nat,Int}\ \ 1\right), \ldots\} \end{aligned}$$

If the familiar set-theoretic union were performed on the domains *Nat* and *Int*, it would be impossible to determine the source domain for any $n \geq 0$ in the union.

The notion of sum naturally extends to n-ary sums, which are constructed by the notation:

$$\sum_{i=1}^{n} D_i = D_1 + D_2 + \ \ldots\ + D_n = \{\left(Inj\,i_{D_1,\ldots,D_n}\ \ d_i\right) \mid d_i\ :\ D_i\}$$

When $S = D_1 + \ \ldots\ + D_n$ and all the domain names D_i are distinct, we write $D_i \mapsto S$ as a synonym for $Inj\,i_{D_1,\ldots,D_n}$. For example, since the *Bool* domain contains only two elements, we can represent it as the sum of two *Unit* domains:

$$\begin{aligned} &Bool = True\ + False \\ &True = Unit \\ &False = Unit \end{aligned}$$

Then the value *true* would be a synonym for $(True \mapsto Bool\ unit)$ and the value *false* would be a synonym for $(False \mapsto Bool\ unit)$. If *Bool* were instead described as the sum *Unit* + *Unit*, the mnemonic injection functions could not be used because the name $Unit \mapsto Bool$ would be ambiguous.[11]

Elements of a sum domain are detagged by a **matching** $_{S,D}$ construct that maps an element of the domain S into an element of the domain D. A **matching** $_{S,D}$ construct has one clause for each possible summand domain in S. If $S = \sum_{i=1}^{n} D_i$ and $s : S$, then the form of this construct is:

matching $_{S,D}$ s
▷ $(D_1 \mapsto S\ \ I_1)$ ‖ E_1
▷ $(D_2 \mapsto S\ \ I_2)$ ‖ E_2
...
▷ $(D_n \mapsto S\ \ I_n)$ ‖ E_n
endmatching

where each E_i is a metalanguage expression of type $D_.$ The subscripts on **matching** will be omitted when they are clear from context.

[11] Note that the alternative injection notation is one place where the name, not the structure of a domain, matters. So even though $True = Unit$, the injection function $True \mapsto Bool$ is *not* the same as the $Unit \mapsto Bool$.

In this notation, s is called the **discriminant**, and lines of the form

$$\triangleright\,(D_i \mapsto S\ \ I_i)[\![\ E_i$$

are called **clauses**. The part of the clause between the $\triangleright$ and the $[\![$ is called the **head** of the clause, and the part of the clause after the $[\![$ is called the **body** of the clause. This notation is pronounced "If the discriminant is the oneof $\big(Inj\,i_{D_1,\ldots,D_n}\ \ d_i\big)$, then the value of the **matching** $_{S,D}$ expression is the value of E_i in a context where the identifier I_i stands for d_i."

The head of a clause is treated as a pattern that can potentially be matched by the discriminant. That is, if the discriminant could have been injected into the sum domain by the expression $(D_i \mapsto S\ \ I_i)$, then in this expression I_i must denote a value from D_i. When such a match is successful, the body is evaluated assuming I_i has this value.

For example, the value of

$$\begin{array}{l}\textbf{matching}\ \big(Inj\,1_{Nat,Int}\ \ 3\big)\\ \triangleright\,((Nat \mapsto Nat + Int)\ \ I_{nat})\ [\![\ (Nat \hookrightarrow Int\ \ (I_{nat} +_{Nat} 1))\\ \triangleright\,((Int \mapsto Nat + Int)\ \ I_{int})\ [\![\ (I_{int} \times_{Int} I_{int})\\ \textbf{endmatching}\end{array}$$

is 4, because the element $\big(Inj\,1_{Nat,Int}\ \ 3\big)$ matches the head of the first clause, and when I_{nat} is 3, the value of $(I_{nat} +_{Nat} 1)$ is 4. Similarly, the value of

$$\begin{array}{l}\textbf{matching}\ \big(Inj\,2_{Nat,Int}\ \ 3\big)\\ \triangleright\,((Nat \mapsto Nat + Int)\ \ I_{nat})\ [\![\ (Nat \hookrightarrow Int\ \ (I_{nat} +_{Nat} 1))\\ \triangleright\,((Int \mapsto Nat + Int)\ \ I_{int})\ [\![\ (I_{int} \times_{Int} I_{int})\\ \textbf{endmatching}\end{array}$$

is 9. Note that the inclusion function $Nat \hookrightarrow Int$ is necessary to guarantee that body expression of the first clause has type *Int*.

Since a **matching** construct has one clause for each summand, there is exactly one clause that matches the discriminant. However, for convenience, the distinguished clause head $\triangleright$ **else** may be used as a catch-all to handle all tags unmatched by previous clauses.

When the expression E_{test} denotes a boolean truth value, the notation

$$\textbf{if}\ _D\ E_{test}\ \textbf{then}\ E_{true}\ \textbf{else}\ E_{false}\ \textbf{fi}$$

is an abbreviation for the case expression

$$\begin{array}{l}\textbf{matching}\ _{Bool,D}\ E_{test}\\ \triangleright\,(True \mapsto Bool\ I_{ignore})\ [\![\ E_{true}\\ \triangleright\,(False \mapsto Bool\ I_{ignore})\ [\![\ E_{false}\\ \textbf{endmatching}\,.\end{array}$$

This abbreviation treats the *Bool* domain as the sum of two *Unit* domains (see page 794). The **if** is subscripted with the domain D of the result, but we will omit it when it is clear from context. Here, the identifier I_{ignore} should be an identifier that does not appear in either E_{true} or E_{false}.[12]

Like products, the sums are abstractions defined only by the behavior of injection functions and the **matching** construct. In particular, these must satisfy the following two properties:

1. $$\begin{array}{|l} \textbf{matching}\ (D_i \mapsto S\ \ d_i) \\ \triangleright\, (D_1 \mapsto S\ \ I_1)\ [\![\ \ I_1 \\ \quad \vdots \\ \triangleright\, (D_n \mapsto S\ \ I_n)\ [\![\ \ I_n \\ \textbf{endmatching} \end{array} = d_i,\ \ 1 \le i \le n$$

2. $$\begin{array}{|l} \textbf{matching}\ d \\ \triangleright\, (D_1 \mapsto S\ \ I_1)\ [\![\ \ (D_1 \mapsto S\ \ I_1) \\ \quad \vdots \\ \triangleright\, (D_n \mapsto S\ \ I_n)\ [\![\ \ (D_n \mapsto S\ \ I_n) \\ \textbf{endmatching} \end{array} = d,\ \ 1 \le i \le n$$

Any implementation of sums in which the injection functions and **matching** satisfy these two properties is a legal implementation of sums.

A.3.4 Sequence Domains

Sequence domains model finite sequences of elements all taken from the same domain. They are built by the * domain constructor; a sequence domain whose sequences contain elements from domain D is written D^*. An element of a sequence domain is simply called a **sequence**. A sequence is characterized by its length n and its ordered elements, which are indexed from 1 to n.

A length-n sequence over the domain D is constructed by the function

$$sequence_{n,D} : D^n \to D^*.$$

Thus $sequence_{3,Int}\langle -5, 7, -3\rangle$ is a sequence of length three with -5 at index 1, 7 at index 2, and -3 at index 3. We will abbreviate $\big(sequence_{n,D}\ \ d_1 \ldots d_n\big).$ as $[d_1, \ldots, d_n]_D$. So the sample sequence above could also be written $[-5, 7, -3]_{Int}$, and the empty sequence of integers would be written $[\,]_{Int}$.[13] As elsewhere, we will omit the subscripts when they can be inferred from context.

[12] This restriction prevents the variable capture problems discussed in Section 6.3.

[13] The empty sequence is created using a 0-tuple.

Every sequence domain D^* is equipped with the following constructor, predicate, and selectors:

- $cons_D : D \to (D^* \to D^*)$
  If $d : D$ and s is a length-n sequence over D^*, then $(cons_D\ d\ s)$ is a length-$n+1$ sequence whose first element is d and whose ith element is the $i-1$th element of s, $2 \leq i \leq n+1$.

- $empty?_D : D^* \to Bool$
  $\big(empty?_D\ s\big)$ is *true* if $s = [\,]_D$ and *false* otherwise.

- $head_D : D^* \to D$
  If $s : D^*$ is nonempty, $(head_D\ s)$ is the first element of s. Defining the *head* of an empty sequence is somewhat problematic. One approach is to treat $(head_D\ [\,]_D)$ as undefined, in which case *head* is only a partial function. An alternative approach that treats *head* as a total function is to define $(head_D\ [\,]_D)$ as a particular element of D.

- $tail_D : D^* \to D^*$
  If $s : D^*$ is nonempty, $(tail_D\ s)$ is the subsequence of the sequence s that consists of all elements but the first element. If s is empty, $(tail_D\ s)$ is defined as $[\,]_D$.

Other useful functions can be defined in terms of the above functions:

$$
\begin{array}{l}
length_D : D^* \to Nat \\
= \lambda d^*\,.\ \textbf{if}\ \big(empty?_D\ d^*\big)\ \textbf{then}\ 0\ \textbf{else}\ \big(1 +_{Nat}\ \big(length_D\ \ (tail_D\ d^*)\big)\big)\ \textbf{fi}
\end{array}
$$

$$
\begin{array}{l}
nth_D : Pos \to D^* \to D \\
= \lambda p d^*\,.\ \textbf{if}\ (p =_{Pos} 1)\ \textbf{then}\ (head_D\ d^*)\ \textbf{else}\ (nth_D\ \ (p -_{Pos} 1)\ \ (tail_D\ d^*))\ \textbf{fi}
\end{array}
$$

$$
\begin{array}{l}
append_D : D^* \to D^* \to D^* \\
= \lambda d_1{}^* d_2{}^*\,.\ \textbf{if}\ \big(empty?_D\ d_1{}^*\big)\ \textbf{then}\ d_2{}^* \\
\qquad\qquad \textbf{else}\ \big(cons_D\ \ (head_D\ d_1{}^*)\ \ \big(append_D\ \ (tail_D\ d_1{}^*)\ \ d_2{}^*\big)\big)\ \textbf{fi}
\end{array}
$$

$$
\begin{array}{l}
map_{D_1,D_2} : (D_1 \to D_2) \to D_1{}^* \to D_2{}^* \\
= \lambda f\, d^*\,.\ \textbf{if}\ \big(empty?_{D_1}\ d^*\big)\ \textbf{then}\ [\,]_{D_2} \\
\qquad\qquad \textbf{else}\ \big(cons_{D_2}\ \ (f\ \ (head_{D_1}\ d^*))\ \ \big(map_{D_1,D_2}\ f\ \ (tail_{D_1}\ d^*)\big)\big)\ \textbf{fi}
\end{array}
$$

$length_D$ returns the length of a sequence. nth_D returns the element of the given sequence at the given index. $append_D$ concatenates a length-m sequence and a length-n sequence to form a length-$m+n$ sequence. Give a $(D_1 \to D_2)$ function f and a length-n sequence of D_1 elements $[d_1, \ldots, d_n]$ map_{D_1,D_2} returns a length-n sequence of D_2 elements $[\,(f\ \ d_1), \ldots, (f\ \ d_n)]$

In the above definitions, we use the convention that if d is a variable ranging over the domain D, d^* is a variable ranging over the domain D^*. All of the above function definitions exhibit a simple form of recursion in which the size of the first argument is reduced at every recursive call; by the principle of mathematical induction, all of the functions are therefore well-defined.

The *cons* and *append* functions are common enough to warrant some convenient abbreviations:

- $d\ .\ d^*$ is an abbreviation of $(cons\ d\ d^*)$. The dot (".") is an infix binary function that naturally associates to the right. Thus, $d_1\ .\ d_2\ .\ d^*$ is parsed as $d_1\ .\ (d_2\ .\ d^*)$.
- $d_1{}^*\ @\ d_2{}^*$ is an abbreviation of $(append\ d_1{}^*\ d_2{}^*)$. The at sign, @, is an associative infix binary operator.

As with products and sums, sequences are defined purely in terms of their abstract behavior. A legal implementation of sequence domains is one which satisfies the following properties for all domains D, all $d : D$ and $d^* : D^*$

1. $\left(empty?_D\ [\,]_D\right) = true$
2. $\left(empty?_D\ (d\ .\ d^*)\right) = false$
3. $(head_D\ (d\ .\ d^*)) = d$
4. $(head_D\ [\,]_D) = d_{emptyHead}$, where $d_{emptyHead}$ is a particular element of D chosen for this purpose.
5. $(tail_D\ (d\ .\ d^*)) = d^*$
6. $(tail_D\ [\,]_D) = [\,]_D$
7. $(cons_D\ (head_D\ d^*)\ (tail_D\ d^*)) = d^*$

A.3.5 Function Domains

The final constructor we will consider is the binary infix function domain constructor, $\rightarrow$. In the naïve implementation of domains as sets, $D_1 \rightarrow D_2$ is the domain of all total functions with D_1 as their source and D_2 as their target. Elements of a function domain are called **functions**. As with tuples, there is the possibility for confusion between set-theoretic functions and domain-theoretic functions. These are the same in the naïve implementation, but differ when we change the implementation of domains. In the body of the text, "function" means domain-theoretic function; we explicitly refer to "set-theoretic functions" when necessary. The same holds for arrow notation, which refers to the function domain constructor unless otherwise specified.

The arrow notation meshes nicely with the use of arrows already familiar from set-theoretic function types. Thus, the notation $f : \mathit{Int} \to \mathit{Bool}$ can now be interpreted as "f is an element of the function domain $\mathit{Int} \to \mathit{Bool}$." Elements of this domain are predicates on the integers, such as functions for testing whether an integer is even or odd, or for testing whether an integer is positive or negative. Similarly, the domain $\mathit{Int} \to (\mathit{Int} \to \mathit{Int})$ is the domain of partial functions on two (curried) integer arguments that return an integer. The (curried) binary integer addition, multiplication, etc., functions are all elements of this domain.

The $\to$ constructor is right-associative:

$$D_1 \to D_2 \to \cdots \to D_{n-1} \to D_n \text{ means } (D_1 \to (D_2 \to \cdots (D_{n-1} \to D_n) \cdots))$$

The right-associativity of $\to$ interacts nicely with the left associativity of application in lambda notation. That is, if $a : A$, $b : B$, and $f : (A \to B \to C)$, then $(f\,a) : B \to C$, so that $(f\ a\ b) : C\ = ((f\ a)\ b) : C$, just as we'd like.

We write particular elements of a function domain using lambda notation. Thus

$$(\lambda n\,.\ (n \times_{\mathit{Nat}} n)) \ :\ \mathit{Nat} \to \mathit{Nat}$$

is the squaring function on natural numbers, and

$$(\lambda i\,.\ (i >_{\mathit{Int}} 0)) \ :\ \mathit{Int} \to \mathit{Bool}$$

is a predicate for testing whether an integer is positive.

As before, we require all abstractions to be well-typed. We can always specify the type of an abstraction by giving it an explicit type. So

$$\lambda x\,.\,x \ :\ \mathit{Int} \to \mathit{Int}$$

specifies the identity function on integers, while

$$\lambda x\,.\,x \ :\ \mathit{Bool} \to \mathit{Bool}$$

specifies the identity function on booleans.

However, to enhance the readability of abstractions, we will use a convention in which each domain of interest has associated with it a **domain variable** that ranges over elements of the domain. For example, consider the following domain definitions:

$b \in \mathit{Bool}$
$n \in \mathit{Nat}$
$p \in \mathit{Nat\text{-}Pred} = \mathit{Nat} \to \mathit{Bool}$

The domain variable b ranges over the *Bool* domain, the domain variable n ranges over the *Nat* domain, and the domain variable p ranges over the function domain $Nat \to Bool$.

Domain variables, possibly in subscripted or primed form, are used in metalanguage expressions to indicate that they denote only entities from their associated domain. Thus $(\lambda b\,.\,b)$ and $(\lambda b_1\,.\,b_1)$ unambiguously denote the identity function in the domain $Bool \to Bool$, $(\lambda n\,.\,n)$ and $(\lambda n'\,.\,n')$ both denote the identity function in the domain $Nat \to Nat$, and $(\lambda p\,.\,p)$ denotes the identity function in the domain

$$Nat - Pred \to Nat - Pred \;=\; (Nat \to Bool) \to (Nat \to Bool)\;.$$

As another example, the expression $(\lambda n\,.\,\lambda p\,.\,pn)$ is an element of the function domain

$$Nat \to Nat - Pred \to Bool = Nat \to (Nat \to Bool) \to Bool\;.$$

In practice, we will use both explicit and implicit typing of domain elements. When we define a value named v from a domain D, we will first write a type of the form $v : D$ that specifies that v names an element from D. Then we will give a definition for the name that uses domain variables where appropriate. So an integer identity function is written

$$\textit{integer-identity} : Int \to Int \;=\; \lambda i\,.\,i$$

and the notation for an identity parameterized over a domain D is:

$$\textit{identity}_D : D \to D \;=\; \lambda d\,.\,d$$

In fact, we have already used this notation to describe the operations on a sequence domain in Section A.3.4.

Our description of function domains in this section has a different flavor than the description of product, sum, and sequence domains. With the other domains, elements of the compound domain were abstractly defined by assembly functions that had to satisfy certain properties with respect to disassembly functions. But with function domains, we concretely specify the elements as set-theoretic functions designated by lambda notation. Is there a more abstract approach to defining function domains? Yes, but it is rather abstract and not important to our current line of development. See Exercise A.3.

▷ **Exercise A.1** It is natural to represent a oneof in $\sum_{i=1}^{n} D_i$ as a set-theoretic pair containing the tag i and an element d_i of D_i:

$$\big(Inj\, i_{D_1,\ldots,D_n}\;\; d_i\big) = \langle i, d_i\rangle$$

Assuming that oneofs are represented as pairs, use lambda notation to construct a set-theoretic function of a oneof argument $s \in D_1 + D_2$ that has the same meaning as the following **matching** expression:

matching s
$\triangleright\, (D_1 \mapsto S\ \ I_1)\ [\![\ \ E_1$
$\triangleright\, (D_2 \mapsto S\ \ I_2)\ [\![\ \ E_2$
endmatching

(Use the three argument if_T function on page 782 rather than the **if** abbreviation on page 795, which itself is implemented in terms of a **matching** expression. Assume E_1 and E_2 are of type T.) $\triangleleft$

$\triangleright$ **Exercise A.2** Suppose that A, B, C, and D are any domains. Extend the notation $\times$ so that it defines a binary infix operator on functions with the following signature:

$$((A \to B) \times (C \to D)) \to ((A \times C) \to (B \times D))$$

If $f : A \to B$, $g : C \to D$, $a : A$, and $c : C$, then $f \times g : (A \times C) \to (B \times D)$ is defined by:

$$\langle f, g\rangle\langle a, c\rangle = \langle (f\ \ a)\,, (g\ \ c)\rangle \qquad f \times g = \langle f \circ Proj1,\ g \circ Proj2\rangle$$

Suppose that $h : (A \times C) \to (B \times D)$ is a set-theoretic function. Show that

$$h = (Proj1\ \ h) \times (Proj2\ \ h) \qquad \triangleleft$$

$\triangleright$ **Exercise A.3** This exercise explores some further properties of function domains. Consider the following two functions:

- $apply_{A,B} : ((A \to B) \times A) \to B$ If $f : A \to B$ and $a : A$, then $\big(apply_{A,B}\ \ \langle f, a\rangle\big)$ denotes the result of applying f to a.
- $curry_{A,B,C} : ((A \times B) \to C) \to (A \to (B \to C))$ If $f : (A \times B) \to C$, then $\big(curry_{A,B,C}\ \ f\big)$ denotes a curried version of f — i.e., it denotes a function g such that $(g\ \ a\ \ b) = (f\ \ \langle a, b\rangle)$ for all $a \in A$ and $b \in B$.

a. Use lambda notation to define set-theoretic versions of *apply* and *curry*.

b. Using your definitions from above, show that if $f : (A \times B) \to C$, then

$$f = apply_{B,C} \circ \big(\big(curry_{A,B,C}\ \ f\big) \times id_B\big)$$

The meaning of $\times$ on functions is defined in Exercise A.2. Recall that id_D is the identity function on domain D.

c. Using your definitions from above, show that if $g : A \to (B \to C)$, then

$$g = \big(curry_{A,B,C}\ \ \big(apply_{B,C} \circ (g \times id_B)\big)\big)$$

It turns out that any domain implementation with an *apply* and a *curry* function that satisfy the above properties is a legal implementation of a function domain. This is the abstract view of function domains alluded to above. ◁

A.4 Metalanguage Summary

So far we've introduced many pieces of the metalanguage. The goal of this section is to put all of the pieces together. We'll summarize the metalanguage notation introduced so far, and introduce a few more handy notations.

In the study of programming languages, it is often useful to break up the description of a language into two parts: the core of the language, called the **kernel**, and various extensions that can be expressed in terms of the core, called the **syntactic sugar**. We shall use this approach to summarize the metalanguage. (See Section 6.2 for an example of using this approach to specify a programming language.)

A.4.1 The Metalanguage Kernel

The entities manipulated by the metalanguage are domains and their elements. Domains are either primitive, in which case they can be viewed as sets of unstructured elements, or compound, in which case they are built out of component domains. Domains are denoted by **domain expressions**. Domain expressions are either domain names (such as *Bool*, *Nat*, etc.) or are the application of the domain operators $\times$, $+$, *, and $\rightarrow$ to other domain expressions. New names can be given to domains via **domain definitions**. Domain definitions can also introduce domain variables that range over elements of the domains.

Domain elements are denoted by **element expressions**. The kernel element expressions are summarized in Figure A.1.

Constants are names for primitive domain elements and functions; these include numbers, booleans, and functions. We will assume that the domain of every constant is evident from context. Variables are names introduced as formal parameters in abstractions or as the defined name of a definition. Every variable ranges over a particular domain. If a variable is the domain variable introduced by some domain definition, it is assumed to range over the specified domain; otherwise, the type of the variable should be explicitly provided to indicate what domain it ranges over.

Applications are compound expressions in which an operator is applied to an operand. The operator expression must denote an element of a function domain $S \rightarrow T$, and the operand expression must denote an element of the domain S;

- **constants**: e.g., *0*, *true*, $+_{Nat}$, $tuple_{Nat,Bool}$, $Proji_{Nat,Bool}$
- **variables**: e.g., a, b', c_2, *fact*
- **applications**: e.g, $(fact\ 5)$, $((+_{Nat}\ 2)\ 3)$, $((\lambda a\,.\,a)\ 1)$
- **abstractions**: e.g., $(\lambda a\,.\,a)$, $(\lambda b\,.\,1)$, $(\lambda a\,.\,\lambda b\,.\,\lambda c\,.\,(ca)b)$
- **case analysis**:

matching s
$\triangleright\,(D_1 \mapsto S\ I_1) \parallel E_1$
$\triangleright\,(D_2 \mapsto S\ I_2) \parallel E_2$
...
$\triangleright\,(D_n \mapsto S\ I_n) \parallel E_n$
endmatching

or

matching s
$\triangleright\,(D_1 \mapsto S\ I_1) \parallel E_1$
$\triangleright\,(D_2 \mapsto S\ I_2) \parallel E_2$
...
$\triangleright$ **else** $\parallel E_{else}$
endmatching

Figure A.1: The kernel element expressions.

in this case, the application denotes an element of type T. Applications with multiple operands are usually expressed by currying. Elements of primitive domains are often the operands to functions (such as arithmetic and logical functions) associated with the domain. Elements of product, sum, and sequence domains can be built by the application of constructor functions ($tuple_{D_1,\ldots,D_n}$, $D_i \mapsto S$, or $sequence_{n,D}$, respectively) to the appropriate arguments. Compound domains are equipped with many other useful functions that operate on elements of the domain.

Abstractions are compound expressions that denote the elements of function domains. Structurally, an abstraction consists of a formal parameter variable and a body element expression. An abstraction $(\lambda I\,.\,E_{body})$ specifies the function graph containing all pairs $\langle I, t\rangle$ where I ranges over the source domain of the function and t is the target domain element that is the value of the body expression E_{body} for the given I. The type of the abstraction should either be given explicitly or should be inferable from the structure of the parts of the abstraction.

While the parts of products and sequences are extracted by function application, elements of a sum domain are disassembled by the **matching** construct. A **matching** construct consists of a discriminant and a set of clauses, each of which has a pattern and a body. There must be one clause to handle each summand in the domain of the discriminant. All body expressions must denote elements of the same domain so that the domain of the value denoted by the **matching** expression is clear.

The element expressions in Figure A.1 are often used in conjunction with

definitions to specify a domain element. A definition has the form

$$name : type = expression$$

where *name* is the name of the element being defined, *type* is a domain expression that denotes the domain to which the defined element belongs, and *expression* is a metalanguage expression that specifies the element. Definitions may only be recursive in the case where it can be shown that they define a unique element in the domain specified by the type. One way to do this is to use induction; another way is to use the iterative fixed point technique developed in Chapter 5.

A.4.2 The Metalanguage Sugar

It is *possible* to write all element expressions using the kernel element expressions, but it is not always *convenient* to do so. We have introduced various notational conventions to make the metalanguage more readable and concise. We review those notations here, and introduce a few more.

Figure A.2 summarizes the syntactic sugar for element expressions. Applications and abstractions are simplified by various conventions. The default left-associativity of application simplifies the expression of multi-argument applications; thus, $(expt\ 2\ 5)$ is an abbreviation for $((expt\ 2)\ 5)$. This default can be overridden by explicit parenthesization. Applications of familiar functions like $+_{Nat}$ are often written in infix style to enhance readability. For example, $(2 +_{Nat} 3)$ is an abbreviation for $((+_{Nat}\ 2)\ 3)$. The formal parameters of nested abstractions are often coalesced into a single abstraction. For instance, $\lambda abc\,.\,ca$ is shorthand for $\lambda a\,.\,\lambda b\,.\,\lambda c\,.\,ca$.

The construction of elements in product, sum, and sequence domains is aided by special notation. Thus,

$$\begin{array}{rll} \langle 1, true\rangle & \text{is shorthand for} & \big(tuple_{Nat,Bool}\ 1\ true\big) \\ ((Nat \mapsto Nat + Int)\ 3) & \text{is shorthand for} & \big(Inj\,1_{Nat,Int}\ 3\big) \\ [5,3,2,7] & \text{is shorthand for} & sequence_{4,Nat}\ \langle 5,3,2,7\rangle \end{array}$$

(We have assumed in all these examples that the numbers are elements of *Nat* rather than of some other numerical domain.) The notations $d\,.\,d^*$ and $d_1{}^* \mathbin{@} d_2{}^*$ are abbreviations for *cons* and *append* respectively, so that the following notations all denote the same sequence of natural numbers:

$$[5,3,2,7] = 5\ .\ [3,2,7] = [5,3]\ @\ [2,7]$$

The **if** conditional expression

$$\textbf{if}\ E_{bool}\ \textbf{then}\ E_{if-true}\ \textbf{else}\ E_{if-false}\ \textbf{fi}$$

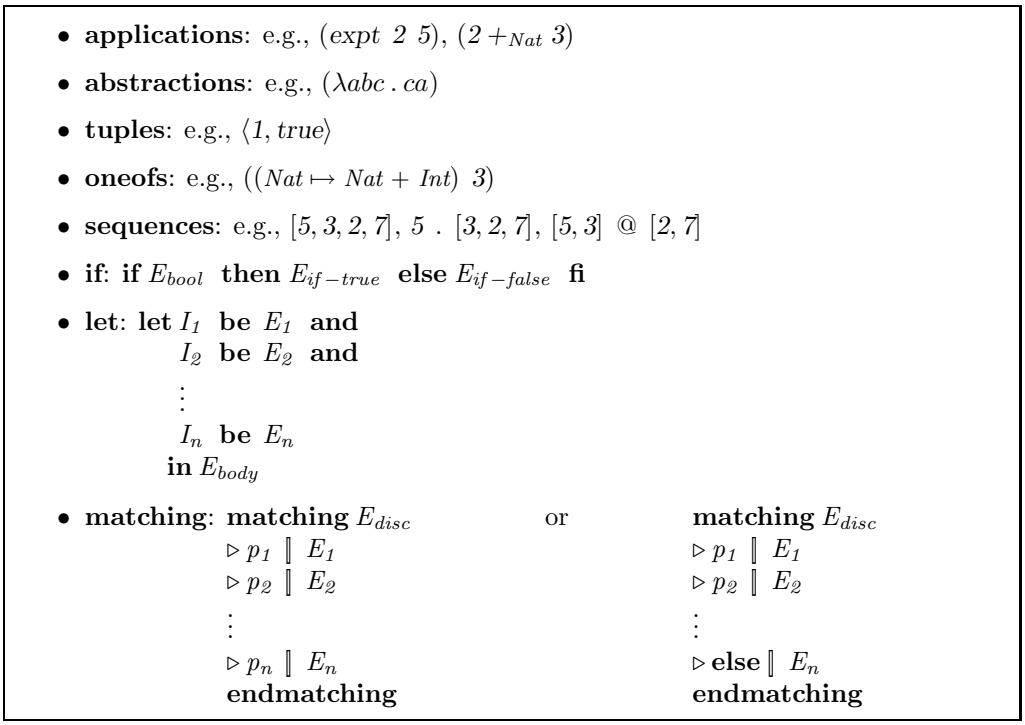


Figure A.2: Sugar for element expressions.

is an abbreviation for the following case analysis:

$$
\begin{array}{l}
\textbf{matching } E_{bool} \\
\triangleright\, (\mathit{True} \mapsto \mathit{Bool}\ I_{ignore}) \parallel E_{if-true} \\
\triangleright\, (\mathit{False} \mapsto \mathit{Bool}\ I_{ignore}) \parallel E_{if-false} \\
\textbf{endmatching}
\end{array}
$$

where E_{bool} is an expression that denotes an element of the domain *Bool* and $E_{if-true}$ and $E_{if-false}$ denote elements from the same domain. The variable I_{ignore} can be any variable that does not appear in $E_{if-true}$ or $E_{if-false}$. This notation assumes that the *Bool* domain is represented as a sum of two *Unit* domains.

The **let** expression is new:

$$
\begin{array}{l}
\textbf{let } I_1 \textbf{ be } E_1 \textbf{ and} \\
\quad I_2 \textbf{ be } E_2 \textbf{ and} \\
\quad \vdots \\
\quad I_n \textbf{ be } E_n \\
\textbf{in } E_{body}
\end{array}
$$

is pronounced "Let I_1 be the value of E_1 and I_2 be the value of E_2 ... and I_n be the value of E_n in the expression E_{body}." The **let** expression is used to name intermediate results that can then be referenced by name in the body expression. The value of a **let** expression is the value of its body in a context where the specified bindings are in effect. The **let** expression is just a more readable form of an application of a manifest abstraction:

$$((\lambda I_1\, I_2 \ldots I_n\, .\, E_{body})\ \ E_1\ \ E_2\ \ \ldots\ \ E_n)$$

The **matching** expression is extended to simplify the extraction of tuple and sequence components:

$$
\begin{array}{l}
\textbf{matching } E_{disc} \\
\triangleright\, p_1 \parallel E_1 \\
\triangleright\, p_2 \parallel E_2 \\
\vdots \\
\triangleright\, p_n \parallel E_n \\
\textbf{endmatching}
\end{array}
$$

As before, a **matching** expression consists of a discriminant and a number of clauses. The two parts of a **matching** clause are called the **pattern** and the **body**. A pattern is composed out of constants, variables, and tuple and sequence constructors; for example, the following are typical patterns:

$\langle n, 1\rangle$,
$\langle\langle w, x\rangle, y, z\rangle$,
$[i_1, -3, i_2]$,
$n\ .\ n^*$,

A pattern is said to **match** a value v if it is possible to assign values to the variables such that the pattern would denote v if it were interpreted as an element expression with the assignments in effect. Thus, the pattern $\langle n, 1\rangle$ matches the value $\langle 2, 1\rangle$ with $n = 2$, but it does not match the values $\langle 3, 4\rangle$ or $\langle 2, 1, 3\rangle$. Similarly, $n\ .\ n^*$ matches $[3, 7, 4]$ with $n = 3$ and $n^* = [7, 4]$, but it does not match $[\,]$.

The value of the **matching** expression is determined by the first clause (reading top down) whose pattern matches the discriminant. In this case, the value of the **matching** expression is the value of the clause body in a context where all the variables introduced by the pattern are assumed to denote the value determined by the match. For example, consider the following expression, where $d : Nat \times Nat$:

matching d
$\triangleright\ \langle n, 1\rangle \ [\!]\ (n -_{Nat} 1)$
$\triangleright\ \langle n, 2\rangle \ [\!]\ (n \times_{Nat} n)$
$\triangleright\ \langle n_1, n_2\rangle \ [\!]\ (n_1 +_{Nat} n_2)$
endmatching

If the second component of d is 1, then the value of the **matching** expression is one less than the first component; if the second component is 2, then the value of the **matching** expression is the square of the first component; otherwise, the value of the **matching** expression is the sum of the two components. As before, the last clause of the **matching** expression can have an $\triangleright$ **else** pattern that handles any discriminant that did not successfully match the preceding patterns. A **matching** expression is ill-formed if no pattern matches the discriminant.

A **matching** expression can always be rewritten in terms of conditional expressions and explicit component extraction functions. Thus, the **matching** clause above is equivalent to:

if $(Proj2_{Nat,Nat}\ d) =_{Nat} 1$
then $(Proj1_{Nat,Nat}\ d) -_{Nat} 1$
else if $(Proj2_{Nat,Nat}\ d) =_{Nat} 2$
then $(Proj1_{Nat,Nat}\ d) \times_{Nat} (Proj1_{Nat,Nat}\ d)$
else $(Proj1_{Nat,Nat}\ d) +_{Nat} (Proj2_{Nat,Nat}\ d)$
fi
fi

In this case, the **matching** expression is more concise and more readable than the desugared form.

In fact, the pattern matching approach is such a powerful notational tool that we shall extend many of our other notations to use implicit pattern matching. For example, we shall allow formal parameters to an abstraction to be patterns rather than just variables. Thus, the abstraction

$$\lambda\langle n_1, n_2\rangle \,.\, (n_1 +_{Nat} n_2)$$

specifies a function with type $(Nat \times Nat) \rightarrow Nat$ that is shorthand for

$$\begin{array}{l}\lambda d \,.\ \textbf{matching}\ d\\ \quad \triangleright \langle n_1, n_2\rangle \ [\!] \ (n_1 +_{Nat} n_2)\\ \quad \textbf{endmatching}\end{array}$$

where d is assumed to range over $Nat \times Nat$. Similarly, we will allow the variable positions of **let** expressions to be filled by general pattern expressions.

The great flexibility of patterns in **matching** are also useful in defining functions. Throughout the book, we will often avoid a very long (even multi-page) **matching** construct by using patterns to define a function by cases. For example, we could write a function that maps sequences of identifiers to the length of the sequence:

$$\begin{array}{l}\mathit{length\text{-}example} : \text{Identifier}^* \rightarrow \mathit{Nat}\\ \mathit{length\text{-}example}\ [\,] \ = \ 0\\ \mathit{length\text{-}example}\ (I_{\mathit{fist}} \ . \ {I_{\mathit{rest}}}^*) \ = \ 1 + (\mathit{length\text{-}example}\ {I_{\mathit{rest}}}^*)\end{array}$$

which is equivalent to:

$$\begin{array}{l}\mathit{length\text{-}example} : \text{Identifier}^* \rightarrow \mathit{Nat}\\ = \lambda I^* \,.\ \textbf{matching}\ I^*\\ \qquad \triangleright [\,] \ [\!] \ 0\\ \qquad \triangleright I_{\mathit{fist}} \ . \ {I_{\mathit{rest}}}^* \ [\!] \ 1 + (\mathit{length\text{-}example}\ {I_{\mathit{rest}}}^*)\\ \qquad \textbf{endmatching}\end{array}$$

This notation is especially helpful when we define functions that operate over programs, where each clause defines the function for a particular type of program expression.

Reading

The concept of domains introduced in this appendix is refined in Chapter 5. See the references there for reading on domain theory.

Defining products, sums, and functions in an abstract way is at the heart of **category theory**. [Pie91] and [BW90] are accessible introductions to category theory aimed at computer scientists.

For coverage of computability issues, we recommend [HU79], [Min67], and [Hof80].

Bibliography

[AF00] Andrew W. Appel and Amy Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th Symposium on the Principles of Programming Languages (POPL)*, pages 243–253, Boston, January 2000.

[AJ88] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. Technical Report CS-TR-183-88, Princeton University Department of Computer Science, July (revised September) 1988.

[AM87] Andrew W. Appel and David B. MacQueen. *Proceedings of the Converence on Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, Portland, September 1987.

[AN89] Arvind and Rishiyur S. Nikhil. A dataflow approach to general-purpose parallel computing. Computation Structure Group Memo 302, MIT Laboratory for Computer Science, July 1989.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, pages 598–632, October 1989.

[AP02] Andrew Appel and Jens Palsberg. *Modern Compiler Implementation In Java*. Cambridge University Press, second edition, 2002.

[Ape89] Andrew W. Apel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[App90] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App98a] Andrew Appel. *Modern Compiler Implementation In C*. Cambridge University Press, 1998.

[App98b] Andrew Appel. *Modern Compiler Implementation In ML*. Cambridge University Press, 1998.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press and McGraw-Hill, second edition, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bac78] John Backus. Can programming be liberated from the von Neuman style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):245–264, August 1978.

[Bar92a] H[enrik] P[ieter] Barendregt. Lambda calculi with types. In S[amson] Abramsky, Dov M. Gabbay, and T[homas] S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[Bar92b] Paul S. Barth. Atomic data structures for parallel computing. Technical Report MIT/LCS/TR-532, MIT Laboratory for Computer Science, March 1992.

[BCT94] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[BDD80] H. Boehm, A. Demers, and J. Donahue. An informal description of russell. Technical Report TR80-430, Cornell University, Department of Computer Science, 1980.

[Bir89] Andrew Birrel. An introduction to programming with threads. SRC Report 35, Digital Equipment Corporation, January 1989.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[BL84] R. Burstall and B. W. Lampson. *A Kernel Language for Abstract Data Types and Modules*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Ble92] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, Carnegie-Mellon University Computer Science Department, January 1992.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bri77] P. Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computer Scientists*. Prentice-Hall, 1990.

[BWD95] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Conference on Programming Language Design and Implementation*. ACM SIGPLAN, June 1995.

[BWW+89] J. Backus, J. H. Williams, E. L. Wimmers, P. Lucas, and A. Aiken. FL language manual, parts 1 and 2. Technical Report RJ 7100 (67163), IBM Research, 1989.

[BWW90] J. Backus, J. H. Williams, and E. L. Wimmers. An introduction to the programming language FL. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, Reading, MA, 1990.

[CAC+81] G. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[Car89] Luca Cardelli. Typeful programming. In *IFIP Advanced Seminar on Formal Description of Programming Concepts*, 1989.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[Cha82] G. J. Chaitin. Register allocation and spilling via coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. Proceedings of the ACM SIGPLAN Symposium on Compiler Construction.

[CHD01] K. Crary, R. Harper, and D. Dreyer. A type system for higher-order modules, 2001.

[CHP99] Karl Crary, Robert Robert Harper, and Sidd Puri. What is a recursive module? In *Programming Language Design and Implementation (PLDI)*, June 1999.

[CJW00] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *9th European Symposium on Programming*, pages 56–71, Berlin, Germany, March 2000.

[Cli82] William Clinger. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 226–234, Pittsburgh, PA, 1982.

[CM90] Eric Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon University Computer Science Department, December 1990.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 193–242. MIT Press/Elsevier, 1990.

[CT03] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2003.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[DF96] Paolo Di Blasio and Kathleen Fisher. A calculus for concurrent objects. In *Seventh International Conference on Concurrency Theory (CONCUR96)*, volume 637 of *Lecture Notes in Computer Science*, pages 655–670, Pisa, August 1996.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages (NATO Advanced Study Institute)*, pages 43–112. London: Academic Press, 1968.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. MIT Press/Elsevier, 1990.

[DJG92] V. Dornic, P. Jouvelot, and D. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1:33–45, 1992.

[DWM+01] Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and J. B. Wells. Functioning without closure: Type-safe customized function representations for Standard ML. In *6th International Conference on Functional Programming*, pages 14–25, Firenze, Italy, September 2001. ACM.

[FF86] Matthias Felleisen and Daniel Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts — III*, pages 193–219. North-Holland, 1986.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software – Practice and Experience*, 30(3):199–232, 2000.

[For91] Alessandro Forin. Futures. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 219–241. MIT Press, 1991.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation*, pages 238–247. ACM, 1993. (SIGPLAN Notices, Volume 28, Number 6, June 1993).

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, second edition, 2001.

[GJ90] David Gelernter and Suresh Jagannathan. *Programming Linguistics*. MIT Press, 1990.

[GJSO92] David Gifford, Pierre Jouvelot, Mark A. Sheldon, and James O'Toole. Report on the FX-91 programming language. Technical Report MIT/LCS/TR-531, MIT Laboratory for Computer Science, February 1992.

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[GS90] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 633–674. MIT Press/Elsevier, 1990.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, pages 501–528, October 1985.

[Har86] Robert Harper. Modules and persistence in standard ml. Technical Report ECS-LFCS-86-11, University of Edinburgh, Laboratory for Foundations of Computer Science, September 1986.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, pages 323–364, 1977.

[HH86] James G. Hook and Douglas J. Howe. Impredicative strong existential equivalent to type:type. Technical Report TR 86-760, Department of Computer Science, Cornell University, Ithaca, New York, June 1986.

[HJW+92] Paul Hudak, Simon Petyon Jones, Philip Wadler, et al. Report on the programming language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Converence Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hof80] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Vintage Books, 1980.

[Hor95] Ellis Horowitz. *Programming Languages: A Grand Tour*. W H Freeman & Co., third edition, 1995.

[HS86] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12), 1986.

[HU79] John E. Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley, 1979.

[Hue90] Gerard Huet, editor. *Logical Foundations of Functional Programming*. Addison-Wesley, 1990.

[Hug82] R. J. M. Hughes. Super-combinators: A new implementation technique for applicative languages. In *Symposium on Lisp and Functional Programming*, pages 1–10, August 1982.

[JG89] P. Jouvelot and D. Gifford. Reasoning about continuations with control effects. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 218–226, 1989.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. ACM, 1991.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In *Procedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988. ACM.

[JM97] Simon Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *1997 ACM SIGPLAN Workshop on Types in Compilation*, Amsterdam, The Netherlands, June 1997. Boston College Computer Science Department Technical Report BCCS-97-03.

[Joh75] S. C. Johnson. Yacc — yet another compiler compiler. Computing Science Technical Report 32, Bell Laboratories, Murray Hill, N.J., 1975.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, September 1985.

[Jon96] Simon Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proceedings of the European Symposium on Programming*. Springer, 1996.

[JW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th Symposium on Principles of Programming Languages*. ACM, 1993.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of STACS '87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Kam90] Samuel Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, 1990.

[Kel89] Richard Kelsey. *Compilation by Program Transformation*. PhD thesis, Yale University, 1989.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Principles of Programming Languages*, pages 281–292. ACM, 1989.

[KKR+86] David Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: an optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[L+79] Barbara Liskov et al. CLU reference manual. Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, October 1979.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, pages 308–320, January 1964.

[Lea99] Douglas Lea. *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Addison-Wesley, Boston, 1999.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd Symposium on Principles of Programming Languages*. ACM, 1995.

[Les75] M. E. Lesk. Lex — a lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories, Murray Hill, N.J., 1975.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 47–57, 1988.

[Mac84] David MacQueen. Modules for standard ML. In *Proceedings ACM Symposium on Lisp and Functional Programming*, 1984.

[Mac86] D. B. MacQueen. Using dependent types to express modular structure. In *Symposium on Principles of Programming Languages*. ACM, 1986.

[Mac88] David MacQueen. An implementation of standard ml modules. Part of the SMLNJ Distribution, March 1988.

[Mac99] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Oxford University Press, third edition, 1999.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[McC62] John McCarthy. Towards a mathematical science of computation. In *Information Processing*, pages 21–28, Amsterdam, 1962. North Holland.

[McC67] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holland, Amsterdam, 1967.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, pages 348–375, 1978.

[Mil87] James S. Miller. MultiScheme: A parallel processing system based on MIT Scheme. Technical Report MIT/LCS/TR-402, MIT Laboratory for Computer Science, September 1987.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Min67] Marvin Minsky. *Finite and Infinite Machines*. Prentice-Hall, 1967.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996.

[Mit03] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[ML86] Michael Marcotty and Henry Ledgard. *The World of Programming Languages*. Springer-Verlag, 1986.

[MMS78] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual (version 4.0). System development division, Xerox Palo Alto Research Center, May 1978.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[Mos90] Peter Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 575–631. MIT Press/Elsevier, 1990.

[MP84] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *Principles of Programming Languages*, pages 37–51. ACM, 1984.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, MA, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, June 1998. ACM.

[NNH98] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1998.

[NO93] Scott M. Nettles and James W. O'Toole. Real-time replication garbage collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*. ACM, June 1993.

[NOG93] Scott Nettles, James O'Toole, and David Gifford. Concurrent garbage collection of persistent heaps. Technical Report MIT/LCS/TR-569, MIT Laboratory for Computer Science, June 1993.

[NOPH92] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In *Proceedings of the SIGPLAN International Workshop on Memory Management*, pages 357–364. ACM, Springer-Verlag, September 1992.

[occ95] *occam 2.1 reference manual*. SGS-Thomson Microeclectronics Limited, May 1995.

[OG89] James William O'Toole, Jr. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–217, Portland, Oregon, June 1989. ACM.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, September 1981.

[Plu02] Mike Plusch. *Water: Simplified Web Services and XML Programming*. John Wiley & Sons, Hoboken, NJ, December 2002. ISBN: 0764525360.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation: An International Journal*, 6:233–247, 1993. A history of continuations.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[RG94] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *1994 ACM Conference on Lisp and Functional Programming*, pages 65–78. ACM, June 1994.

[Roz84] Guillermo J. Rozas. Liar, an Algol-like compiler for Scheme. Master's thesis, EECS Department, MIT, January 1984.

[Sab88] Gary W. Sabot. *The Paralation Model*. MIT Press, 1988.

[Sch86a] David Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sch86b] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, MA, 1986.

[Sch94] David Schmidt. *The Structure of Typed Programming Languages*. MIT Press, 1994.

[Sco77] Dana S. Scott. Logic and programming languages. *Communications of the ACM*, 20(9):634–641, September 1977. Turing Award Lecture.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Prgramming*, pages 288–298. ACM, 1992.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3–4):289–360, 1993.

[SG90] Mark A. Sheldon and David K. Gifford. Static dependent types for first class modules. In *Symposium on Lisp and Functional Programming*. ACM, 1990.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, 1997.

[SS76] Guy L. Steele Jr. and Gerald Jay Sussman. LAMBDA: The Ultimate Imperative. Technical Report AIM-353, MIT Artificial Intelligence Laboratory, March 1976.

[Ste77] Guy L. Steele Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or LAMBDA, the Ultimate Goto. Technical Report AIM-443, MIT Artificial Intelligence Laboratory, October 1977.

[Ste78] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Sto85] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1985.

[SW74] C. Strachey and C. Wadsworth. Continuations: A mathematical semantics which can deal with full jumps. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1974.

[SW97] Paul Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, January 1997.

[SW00] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics which can deal with full jumps. *Higher-Order and Symbolic Computation*, 13(1–2):135–152, 2000.

[Ten76] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8), August 1976.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation*. ACM, 1996.

[TO98] Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Muchnik Steve, editor, *Proceedings of the 14th Symposium on Principles of Programming Languages*, Munich, Germany, January 1987. ACM. Revised March 1987.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Wel99] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999. Supersedes [**?**].

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St. Malo, France, September 1992.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[WS97] Mitchell Wand and Gregory T. Sullivan. Denotational semantics using an operationally-based term model. In *24th Symposium on the Principles of Programming Languages (POPL)*, pages 386–399, 1997.

[You81] Richard M. Young. The machine inside the machine: Users' models of pocket calculators. *International Journal of Man-Machine Studies*, 15:51–85, 1981.

Index