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 $[1 \ldots 10]$ 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	$=$	$\{\texttt{#t}, \texttt{#f}\}$
N	$\in$	Intlit	$=$	$\{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
L	$\in$	Lit		
O	$\in$	Primop	$=$	Usual FL primitives except list ops and predicates.

P ::= $\texttt{(flx ((}I_{formal}\ T\texttt{)*)}\ E_{body}\ D_{definitions}\texttt{*)}$	[Program]
D ::= $\texttt{(define}\ I_{name}\ T_{type}\ E_{defn}\texttt{)}$	[Value Definition]
$\mid$ $\texttt{(define-type}\ I_{name}\ T_{defn}\texttt{)}$	[Type Definition]
E ::= L	[Literal]
$\mid$ I	[Variable Reference]
$\mid$ $\texttt{(if}\ E_{test}\ E_{consequent}\ E_{alternate}\texttt{)}$	[Branch]
$\mid$ $\texttt{(lambda ((}I_{formal}\ T\texttt{)*)}\ E_{body}\texttt{)}$	[Abstraction]
$\mid$ $\texttt{(}E_{rator}\ E_{rand}\texttt{*)}$	[Application]
$\mid$ $\texttt{(let ((}I_{name}\ E_{defn}\texttt{)*)}\ E_{body}\texttt{)}$	[Local Value Binding]
$\mid$ $\texttt{(letrec ((}I_{name}\ T_{type}\ E_{defn}\texttt{)*)}\ E_{body}\texttt{)}$	[Local Value Recursion]
$\mid$ $\texttt{(primop}\ O_{name}\ E_{arg}\texttt{*)}$	[Primitive Application]
$\mid$ $\texttt{(error}\ I_{message}\ T\texttt{)}$	[Error]
$\mid$ $\texttt{(the}\ T\ E\texttt{)}$	[Type Ascription]
$\mid$ $\texttt{(tlet ((}I_{name}\ T_{defn}\texttt{)*)}\ E_{body}\texttt{)}$	[Local Type Binding]
L ::= $\texttt{#u}$	[Unit Literal]
$\mid$ B	[Boolean Literal]
$\mid$ N	[Integer Literal]
$\mid$ $\texttt{(symbol}\ I\texttt{)}$	[Symbol Literal]
T ::= $\texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{sym}$	[Base Types]
$\mid$ $\texttt{(-> (}T\texttt{*)}\ T_{body}\texttt{)}$	[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:

$$A \in \textit{Type-Environment} = \text{Identifier} \rightharpoonup \text{Type}$$

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 $dom(A_2) = \{\mathrm{b}, \mathrm{f}, \mathrm{x}, \mathrm{z}\}$ 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

$$\frac{\textit{Premise}_1; \ldots; \textit{Premise}_n}{\textit{Conclusion}} \qquad [\textit{name-of-rule}]$$

where *Conclusion* and each $\textit{Premise}_i$ 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:

$$\frac{\{\} \vdash \texttt{1} : \texttt{bool} \quad ; \quad \{\} \vdash \texttt{2} : \texttt{bool} \quad ; \quad \{\} \vdash \texttt{3} : \texttt{bool}}{\{\} \vdash \texttt{(if 1 2 3)} : \texttt{bool}}$$

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

$\vdash$ `#u` : `unit` [*unit*] $\vdash N$: `int` [*int*] $\vdash B$: `bool` [*bool*]

$\vdash$ `(symbol` I`)` : `sym` [*sym*] $\vdash$ `(error` $I\ T$`)` : T [*error*]

$$A \vdash I : A(I),\ \text{where } I \in dom(A) \qquad [\textit{var}]$$

$$\frac{A \vdash E_{test} : \texttt{bool} \ ;\ A \vdash E_{con} : T \ ;\ A \vdash E_{alt} : T}{A \vdash \texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)} : T} \qquad [\textit{if}]$$

$$\frac{A[I_1 : T_1,\ \ldots,\ I_n : T_n] \vdash E_{body} : T_{body}}{\begin{array}{c}A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}\ E_{body}\texttt{)}\\ : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}\end{array}} \qquad [\texttt{->}\textit{-intro}]$$

$$\frac{\begin{array}{c}A \vdash E_{rator} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{result}\texttt{)}\\ \forall_{i=1}^{n} \,.\, A \vdash E_i : T_i\end{array}}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)} : T_{result}} \qquad [\texttt{->}\textit{-elim}]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n} \,.\, A \vdash E_i : T_i\\ A[I_1 : T_1,\ \ldots,\ I_n : T_n] \vdash E_{body} : T_{body}\end{array}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [\textit{let}]$$

$$\frac{\begin{array}{c}\forall_{i=1}^{n} \,.\, A' \vdash E_i : T_i\\ A' \vdash E_{body} : T_{body}\end{array}}{A \vdash \texttt{(letrec ((}I_1\ T_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\ E_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [\textit{letrec}]$$

where $A' = A[I_1 : T_1,\ \ldots,\ I_n : T_n]$

$$\frac{\begin{array}{c}A_{std} \vdash O_{name} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{result}\texttt{)}\\ \forall_{i=1}^{n} \,.\ A \vdash E_i : T_i\end{array}}{A \vdash \texttt{(primop } O_{name}\ E_1\ \ldots\ E_n\texttt{)} : T_{result}} \qquad [\textit{primop}]$$

$$\frac{A \vdash E : T}{A \vdash \texttt{(the } T\ E\texttt{)} : T} \qquad [\textit{the}]$$

$$\frac{A \vdash ([T_i/I_i]_{i=1}^{n})E_{body} : T_{body}}{A \vdash \texttt{(tlet ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [\textit{tlet}]$$

$$\frac{A[I_1 : T_1,\ \ldots,\ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(flx ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}\ E_{body}\texttt{)} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\textit{prog}]$$

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.

$$\texttt{((lambda (}I_1 \ \ldots I_n\texttt{)}\ E_{body}\texttt{)}\ E_1 \ \ldots E_n\texttt{)} \Rightarrow ([E_i/I_i]_{i=1}^{n})E_{body} \qquad [\text{FL*-}\mathit{apply}]$$

$$\frac{E_{rator} \Rightarrow E_{rator}{}'}{\texttt{(}E_{rator}\ E_1 \ \ldots E_n\texttt{)} \Rightarrow \texttt{(}E_{rator}{}'\ E_1 \ \ldots E_n\texttt{)}} \qquad [\text{FL*-}\mathit{rator}]$$

$$\begin{array}{l}\texttt{(letrec ((}I_1\ E_1\texttt{)} \ \ldots \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} \\ \Rightarrow ([\texttt{(letrec ((}I_1\ E_1\texttt{)} \ \ldots \texttt{(}I_n\ E_n\texttt{))}\ E_i\texttt{)}/I_i]_{i=1}^{n})E_{body}\end{array} \qquad [\text{FL*-}\mathit{letrec}]$$

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.*

$$\lceil . \rceil : \text{Exp}_{FL/X} \rightarrow \text{Exp}_{FL*}$$

$$\lceil L \rceil = L$$

$$\lceil I \rceil = I$$

$$\lceil \texttt{(if } E_{test}\ E_{consequent}\ E_{alternate}\texttt{)} \rceil = \texttt{(if } \lceil E_{test} \rceil\ \lceil E_{consequent} \rceil\ \lceil E_{alternate} \rceil \texttt{)}$$

$$\lceil \texttt{(lambda ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ E_{body}\texttt{)} \rceil = \texttt{(lambda (}I_1 \ldots I_n\texttt{)}\ \lceil E_{body} \rceil \texttt{)}$$

$$\lceil \texttt{(}E_{rator}\ E_{rand1} \ldots E_{randn}\texttt{)} \rceil = \texttt{(}\lceil E_{rator} \rceil\ \lceil E_{rand1} \rceil \ldots \lceil E_{randn} \rceil \texttt{)}$$

$$\lceil \texttt{(let ((}I_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} \rceil = \\ \texttt{((lambda (}I_1 \ldots I_n\texttt{)}\ \lceil E_{body} \rceil \texttt{)}\ \lceil E_1 \rceil \ldots \lceil E_n \rceil \texttt{)}$$

$$\lceil \texttt{(letrec ((}I_1\ T_1\ E_1\texttt{)} \ldots \texttt{(}I_n\ T_1\ E_n\texttt{))}\ E_{body}\texttt{)} \rceil = \\ \texttt{(letrec ((}I_1\ \lceil E_1 \rceil\texttt{)} \ldots \texttt{(}I_n\ \lceil E_n \rceil\texttt{))}\ \lceil E_{body} \rceil \texttt{)}$$

$$\lceil \texttt{(primop } O_{name}\ E_1 \ldots E_n\texttt{)} \rceil = \texttt{(primop } O_{name}\ \lceil E_1 \rceil \ldots \lceil E_n \rceil \texttt{)}$$

$$\lceil \texttt{(tlet ((}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}\ E_{body}\texttt{)} \rceil = \lceil E_{body} \rceil$$

$$\lceil \texttt{(the } T\ E\texttt{)} \rceil = \lceil E \rceil$$

$$\lceil \texttt{(error } I_{message}\ T\texttt{)} \rceil = \texttt{(error } I_{message}\texttt{)}$$

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

$E ::= \ldots$	$\texttt{(pair } E_{fst} \ E_{snd}\texttt{)}$	[Pair Intro]
	$\texttt{(fst } E_{pair}\texttt{)}$	[Pair Elim First]
	$\texttt{(snd } E_{pair}\texttt{)}$	[Pair Elim Second]
$T ::= \ldots$	$\texttt{(pairof } T_{fst} \ T_{snd}\texttt{)}$	[Pair Type]

Type Rules

$$\frac{A \vdash E_f : T_f \ ; \ A \vdash E_s : T_s}{A \vdash \texttt{(pair } E_f \ E_s\texttt{)} : \texttt{(pairof } T_f \ T_s\texttt{)}} \qquad [\textit{pairof-intro}]$$

$$\frac{A \vdash E_{pair} : \texttt{(pairof } T_f \ T_s\texttt{)}}{A \vdash \texttt{(fst}E_{pair}\texttt{)} : T_f} \qquad [\textit{pairof-elim-F}]$$

$$\frac{A \vdash E_{pair} : \texttt{(pairof } T_f \ T_s\texttt{)}}{A \vdash \texttt{(snd } E_{pair}\texttt{)} : T_s} \qquad [\textit{pairof-elim-S}]$$

Figure 12.7: Handling pairs in FL/X.

Syntax

$E ::= \ldots$	$\texttt{(product } E^*\texttt{)}$	[Product Intro]
	$\texttt{(proj } N_{index} \ E_{prod}\texttt{)}$	[Product Elim]
$T ::= \ldots$	$\texttt{(productof } T^*\texttt{)}$	[Product Type]

Type Rules

$$\frac{\forall_{i=1}^{n} \ . \ A \vdash E_i : T_i}{A \vdash \texttt{(product } E_1 \ \ldots \ E_n\texttt{)} : \texttt{(productof } T_1 \ \ldots \ T_n\texttt{)}} \qquad [\textit{productof-intro}]$$

$$\frac{A \vdash E_{prod} : \texttt{(productof } T_1 \ \ldots \ T_n\texttt{)}}{A \vdash \texttt{(proj } N_{index} \ E_{prod}\texttt{)} : T_{N_{index}}}, \text{ where } 1 \le N_{index} \le n \qquad [\textit{productof-elim}]$$

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

$E ::= \ldots$ | `(record (`I E`)*)`	[Record Intro]
| `(select` I E`)`	[Record Elim]
$T ::= \ldots$ | `(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:

$$\frac{A \vdash E_{test} : T_{test} \quad ; \quad A \vdash E_{con} : T_{con} \quad ; \quad A \vdash E_{alt} : T_{alt}}{A \vdash \texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)} : T_{result}} \qquad [if]$$

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.

$$T = T \qquad [\textit{reflexive-=}]$$

$$\frac{T_1 = T_2}{T_2 = T_1} \qquad [\textit{symmetric-=}]$$

$$\frac{T_1 = T_2 \quad ; \quad T_2 = T_3}{T_1 = T_3} \qquad [\textit{transitive-=}]$$

$$\frac{\forall_{i=1}^{n} \;.\; S_i = T_i \quad ; \quad S_{body} = T_{body}}{\texttt{(-> (}S_1 \;\ldots\; S_n\texttt{)}\; S_{body}\texttt{)} = \texttt{(-> (}T_1 \;\ldots\; T_n\texttt{)}\; T_{body}\texttt{)}} \qquad [\texttt{->}\textit{-=}]$$

$$\frac{\forall_{i=1}^{2} \;.\; S_i = T_i}{\texttt{(pairof}\; S_1 \;\; S_2\texttt{)} = \texttt{(pairof}\; T_1 \;\; T_2\texttt{)}} \qquad [\textit{pairof-=}]$$

$$\frac{\forall_{i=1}^{n} \;.\; S_i = T_i}{\texttt{(productof}\; S_1 \;\ldots\; S_n\texttt{)} = \texttt{(productof}\; T_1 \;\ldots\; T_n\texttt{)}} \qquad [\textit{productof-=}]$$

$$\begin{array}{c}\texttt{(recordof (}J_1 \;\; S_1\texttt{)} \;\ldots\; \texttt{(}J_n \;\; S_n\texttt{))} \\ = \texttt{(recordof (}I_1 \;\; T_1\texttt{)} \;\ldots\; \texttt{(}I_n \;\; T_n\texttt{))}\end{array} \qquad [\textit{recordof-=}]$$

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

$E ::= \ldots \mid$ `(begin` E_1 E_2`)`	[Sequential Execution]
$\mid$ `(cell` E`)`	[Cell Creation]
$\mid$ `(cell-ref` E`)`	[Cell Get]
$\mid$ `(cell-set!` E_{cell} E_{val}`)`	[Cell Set]
$T ::= \ldots \mid$ `(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

$$\begin{array}{ll}
E ::= \ldots \mid (\texttt{cons}\ E_{head}\ \ E_{tail}) & \text{[List Creation]} \\
\quad\mid (\texttt{car}\ E_{list}) & \text{[List Head]} \\
\quad\mid (\texttt{cdr}\ E_{list}) & \text{[List Tail]} \\
\quad\mid (\texttt{null}\ T) & \text{[Empty List]} \\
\quad\mid (\texttt{null?}\ \ E_{list}) & \text{[Empty List Test]} \\
 & \\
T ::= \ldots \mid (\texttt{listof}\ T) & \text{[List Type]}
\end{array}$$

Type Rules

$$\frac{\begin{array}{c} A \vdash E_{head} : T \\ A \vdash E_{tail} : (\texttt{listof}\ T) \end{array}}{A \vdash (\texttt{cons}\ E_{head}\ \ E_{tail}) : T_{(listof T)}} \qquad [\textit{cons}]$$

$$\frac{A \vdash E_{list} : (\texttt{listof}\ T)}{A \vdash (\texttt{car}\ E_{list}) : T} \qquad [\textit{car}]$$

$$\frac{A \vdash E_{list} : (\texttt{listof}\ T)}{A \vdash (\texttt{cdr}\ E_{list}) : (\texttt{listof}\ T)} \qquad [\textit{cdr}]$$

$$A \vdash (\texttt{null}\ T) : (\texttt{listof}\ T) \qquad [\textit{null}]$$

$$\frac{A \vdash E_{list} : (\texttt{listof}\ T)}{A \vdash (\texttt{null?}\ \ E_{list}) : \texttt{bool}} \qquad [\textit{null?}]$$

Type Equality

$$\frac{T_1 = T_2}{(\texttt{listof}\ T_1) = (\texttt{listof}\ T_2)} \qquad [\textit{listof-=}]$$

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.

$$\texttt{(recof } I\ T\texttt{)} = \texttt{(recof } I_{new}\ [I_{new}/I]T\texttt{)},\ \text{where } I_{new} \notin FreeIds[\![T]\!] \qquad [recof\text{-}\alpha]$$

$$\texttt{(recof } I\ T\texttt{)} = [\texttt{(recof } I\ T\texttt{)}/I]T \qquad [recof\text{-}\beta]$$

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?

◁

```
Tit1 = (recof it
         (oneof (leaf int)
                (node (recordof (left it)
                                (right it))))

Tit2 = (recof int-tree
         (oneof (leaf int)
                (node (recordof (left int-tree)
                                (right int-tree)))))

Tit3 = (recof it
         (oneof
           (leaf int)
           (node (recordof (left (recof it
                                   (oneof (leaf int)
                                          (node (recordof (left it)
                                                          (right it))))))
                           (right it)))))

Tit4 = (recof it
         (oneof
           (leaf int)
           (node (recordof (left it)
                           (right (recof it
                                    (oneof (leaf int)
                                           (node (recordof (left it)
                                                           (right it))))))))))


Tit5 = (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].