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 FL/R$_{\textsc{Tortoise}}$ 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 FL/R$_{\textsc{Tortoise}}$ and Silk later in this section. The Silk$_{tgt}$ dialect is described in Section 17.12.

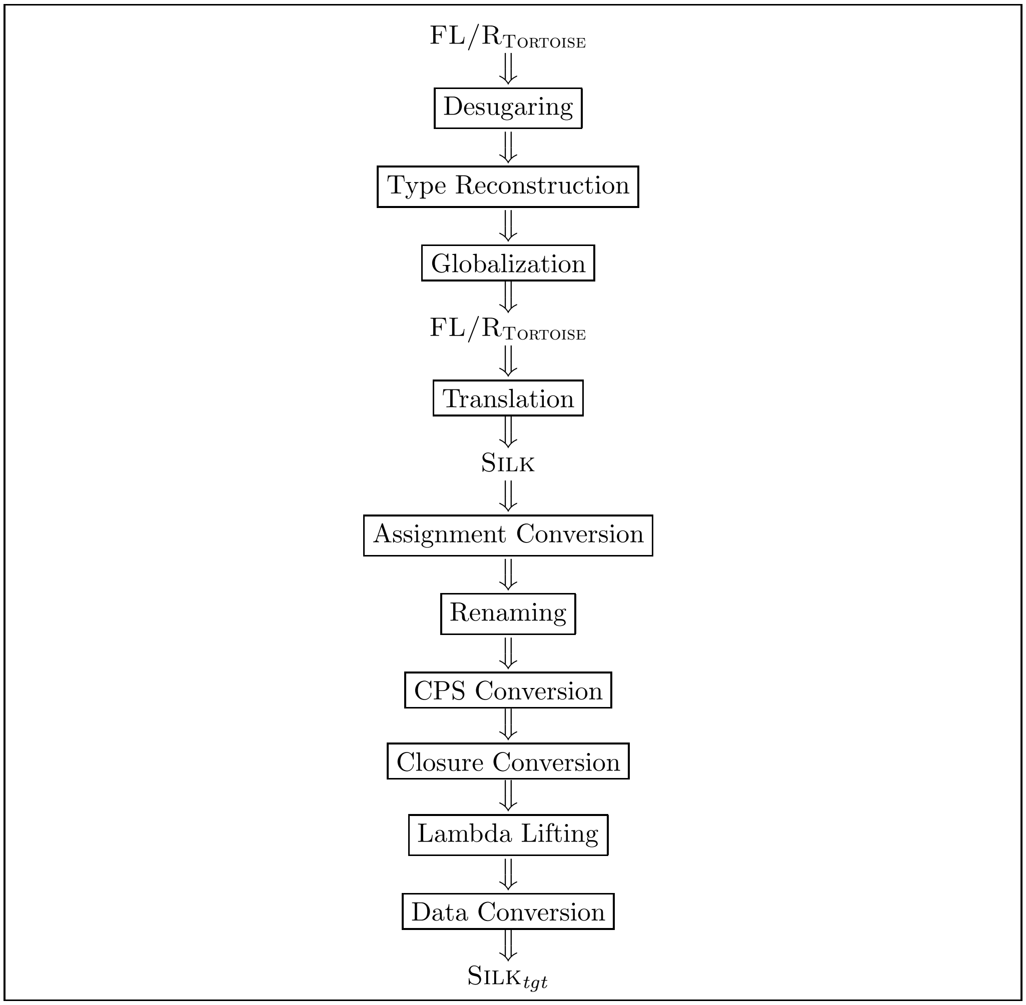

Figure 17.1: Organization of the Tortoise compiler. After desugaring, type reconstruction, and globalization, a FL/R$_{\text{Tortoise}}$ source program is translated into the Silk intermediate language, and the Silk program is gradually transformed into a form that resembles register machine code.

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: $\mathrm{FL/R}_{\textbf{Tortoise}}$

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

$$\begin{array}{ll} P \in \text{Program}_{FL/R} & I \in \text{Identifier}_{FL/R} = \text{usual identifiers}\\ E \in \text{Exp}_{FL/R} & B \in \text{Boollit}_{FL/R} = \{\texttt{\#t}, \texttt{\#f}\}\\ AB \in \text{Abstraction}_{FL/R} & N \in \text{Intlit}_{FL/R} = \{\ldots,\texttt{-2},\texttt{-1},\texttt{0},\texttt{1},\texttt{2},\ldots\}\\ L \in \text{Lit}_{FL/R} & O \in \text{Primop}_{FL/R} \end{array}$$

$$\begin{array}{rcl} P & ::= & \texttt{(flr (}I_{fml}*\texttt{)}\ E_{body}\texttt{)} \\ E & ::= & L \mid I \mid AB \mid \texttt{(}E_{rator}\ E_{rand}*\texttt{)} \mid \texttt{(primop}\ O_{op}\ E_{arg}*\texttt{)}\\ & & \mid \texttt{(if}\ E_{test}\ E_{then}\ E_{else}\texttt{)} \mid \texttt{(set!}\ I_{var}\ E_{rhs}\texttt{)} \mid \texttt{(error}\ I\texttt{)}\\ & & \mid \texttt{(let ((}I_{name}\ E_{defn}\texttt{)}*\texttt{)}\ E_{body}\texttt{)} \mid \texttt{(funrec ((}I_{name}\ AB_{defn}\texttt{)}*\texttt{)}\ E_{body}\texttt{)}\\ AB & ::= & \texttt{(lambda (}I_{fml}*\texttt{)}\ E_{body}\texttt{)}\\ L & ::= & \texttt{\#u} \mid B \mid N \end{array}$$

$$\begin{array}{rcll} O_{FL/R} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} & \text{[Arithmetic ops]}\\ & & \mid \texttt{<=} \mid \texttt{<} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{>} \mid \texttt{>=} & \text{[Relational ops]}\\ & & \mid \texttt{not} \mid \texttt{band} \mid \texttt{bor} & \text{[Logical ops]}\\ & & \mid \texttt{cell} \mid \texttt{\^{}} \mid \texttt{:=} & \text{[Mutable cell ops]}\\ & & \mid \texttt{pair} \mid \texttt{fst} \mid \texttt{snd} & \text{[Pair ops]}\\ & & \mid \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{null} \mid \texttt{null?} & \text{[List ops]} \end{array}$$

Syntactic Sugar

$\texttt{(begin)} \xrightarrow{desugar} \texttt{\#u}$

$\texttt{(begin}\ E\texttt{)} \xrightarrow{desugar} E$

$\texttt{(begin}\ E_1\ E_{rest}*\texttt{)} \xrightarrow{desugar} \texttt{(let ((}I\ E_1\texttt{)) (begin}\ E_{rest}*\texttt{))}$, where I is fresh

$\texttt{(let* ()}\ E_{body}\texttt{)} \xrightarrow{desugar} E_{body}$

$\texttt{(let* ((}I_1\ E_1\texttt{)}\ IE*\texttt{)}\ E_{body}\texttt{)} \xrightarrow{desugar} \texttt{(let ((}I_1\ E_1\texttt{)) (let* (}IE*\texttt{)}\ E_{body}\texttt{))}$
where IE ranges over bindings of the form $\texttt{(}I\ E\texttt{)}$.

$\texttt{(recur}\ I_{fcn}\ \texttt{((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}$
$\xrightarrow{desugar} \texttt{(funrec ((}I_{fcn}\ \texttt{(lambda (}I_1\ \ldots I_n\texttt{)}\ E_{body}\texttt{)))}$
$\texttt{(}I_{fcn}\ E_1\ \ldots\ E_n\texttt{))}$

$\texttt{(scand)} \xrightarrow{desugar} \texttt{\#t}$

$\texttt{(scand}\ E_1\ E_{rest}*\texttt{)} \xrightarrow{desugar} \texttt{(if}\ E_1\ \texttt{(scand}\ E_{rest}*\texttt{) \#f)}$

$\texttt{(scor)} \xrightarrow{desugar} \texttt{\#f}$

$\texttt{(scor}\ E_1\ E_{rest}*\texttt{)} \xrightarrow{desugar} \texttt{(if}\ E_1\ \texttt{\#t (scor}\ E_{rest}*\texttt{))}$

$\texttt{(list)} \xrightarrow{desugar} \texttt{(primop null)}$

$\texttt{(list}\ E_1\ E_{rest}*\texttt{)} \xrightarrow{desugar} \texttt{(primop cons}\ E_1\ \texttt{(list}\ E_{rest}*\texttt{))}$

Standard Identifiers

$I_{std} ::= O$

Figure 17.2: Syntax for FL/R

```
(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

$P \in \text{Program}_{Silk}$ $\qquad$ $I \in \text{Identifier} = \text{usual identifiers}$

$E \in \text{Exp}_{Silk}$ $\qquad$ $B \in \text{Boollit} = \{\texttt{\#t}, \texttt{\#f}\}$

$BV \in \text{BindingValue}_{Silk}$ $\qquad$ $N \in \text{Intlit} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$

$DV \in \text{DataValue}_{Silk}$ $\qquad$ $S \in \text{Poslit} = \{\texttt{1}, \texttt{2}, \texttt{3}, \ldots\}$

$AB \in \text{Abstraction}_{Silk}$ $\qquad$ $O \in \text{Primop}_{Silk}$

$L \in \text{Lit}_{Silk}$

$P ::= \texttt{(silk (}I_{fml}\texttt{*)}\ E_{body}\texttt{)}$

$E ::= L \mid I \mid AB \mid \texttt{(if}\ E_{test}\ E_{then}\ E_{else}\texttt{)} \mid \texttt{(set!}\ I_{var}\ E_{rhs}\texttt{)}$

$\quad \mid \texttt{(call}\ E_{rator}\ E_{rand}\texttt{*)} \mid \texttt{(primop}\ O_{op}\ E_{arg}\texttt{*)}$

$\quad \mid \texttt{(let ((}I_{name}\ E_{defn}\texttt{)*)}\ E_{body}\texttt{)} \mid \texttt{(cycrec ((}I_{name}\ BV_{defn}\texttt{)*)}\ E_{body}\texttt{)}$

$\quad \mid \mid \texttt{(error}\ I\texttt{)}$

$AB ::= \texttt{(lambda (}I_{fml}\texttt{*)}\ E_{body}\texttt{)}$

$BV ::= L \mid AB \mid \texttt{(primop mprod}\ DV\texttt{*)}$

$DV ::= L \mid I$

$L ::= \texttt{\#u} \mid B \mid N$

$O ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$ [Arithmetic ops]

$\quad \mid \texttt{<=} \mid \texttt{<} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{>} \mid \texttt{>=}$ [Relational ops]

$\quad \mid \texttt{not} \mid \texttt{band} \mid \texttt{bor}$ [Logical ops]

$\quad \mid \texttt{mprod} \mid \texttt{(mget}\ S\texttt{)} \mid \texttt{(mset!}\ S\texttt{)}$ [Mutable tuples]

$\quad \mid \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{null} \mid \texttt{null?}$ [List ops]

$\quad \mid$ *... add data conversion bit ops here ...*

Syntactic Sugar

$\texttt{(@mget}\ S\ E_{mp}\texttt{)} \xrightarrow{desugar} \texttt{(primop (mget}\ S\texttt{)}\ E_{mp}\texttt{)}$

$\texttt{(@mset!}\ S\ E_{mp}\ E_{new}\texttt{)} \xrightarrow{desugar} \texttt{(primop (mset!}\ S\texttt{)}\ E_{mp}\ E_{new}\texttt{)}$

$\texttt{(@}O\ E_1\ \ldots\ E_n\texttt{)} \xrightarrow{desugar} \texttt{(primop}\ O\ E_1\ \ldots\ E_n\texttt{)}$, where $O \not\in \{\texttt{mget}, \texttt{mset!}\}$

$\texttt{(let* ()}\ E_{body}\texttt{)} \xrightarrow{desugar} E_{body}$

$\texttt{(let* ((}I_1\ E_1\texttt{)}\ IE\texttt{*)}\ E_{body}\texttt{)} \xrightarrow{desugar} \texttt{(let ((}I_1\ E_1\texttt{)) (let* (}IE\texttt{*)}\ E_{body}\texttt{))}$

where IE ranges over bindings of the form $\texttt{(}I\ E\texttt{)}$.

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

$$\texttt{(let ()}\ E_{body}\texttt{)} \xrightarrow{simp} E_{body} \qquad [\textit{empty-let}]$$

$$\texttt{(cycrec ()}\ E_{body}\texttt{)} \xrightarrow{simp} E_{body} \qquad [\textit{empty-cycrec}]$$

$$\begin{aligned}&\texttt{(call (lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\ E_1\ \ldots\ E_n\texttt{)}\\ &\quad\xrightarrow{simp}\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}\end{aligned} \qquad [\textit{implicit-let}]$$

$$\begin{aligned}&\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) (call}\ E_{rator}\ I_1\ \ldots\ I_n\texttt{))} \xrightarrow{simp} E_{rator},\\ &\quad\text{where } \bullet\ E_{rator} \text{ is a variable or abstraction;}\\ &\quad\phantom{\text{where }} \bullet\ \mathit{FreeIds}[\![E_{rator}]\!] \cap \{I_1,\ldots,I_n\} = \{\}.\end{aligned} \qquad [\textit{eta-lambda}]$$

$$\begin{aligned}&\texttt{(let ((}I\ I'\texttt{))}\ E_{body}\texttt{)} \xrightarrow{simp} [I'/I]E_{body},\\ &\text{where there are no assignments to } I \text{ or } I' \text{ in the program.}\end{aligned} \qquad [\textit{copy-prop}]$$

$$\begin{aligned}&\texttt{(cycrec ((}I_1\ BV_1\texttt{)}\ \ldots\ \texttt{(}I_m\ BV_m\texttt{))}\\ &\quad\texttt{(cycrec ((}I_1{'}\ BV_1{'}\texttt{)}\ \ldots\ \texttt{(}I_n{'}\ BV_n{'}\texttt{))}\\ &\qquad E_{body}\texttt{))}\\ &\xrightarrow{simp}\texttt{(cycrec ((}I_1\ BV_1\texttt{)}\ \ldots\ \texttt{(}I_m\ BV_m\texttt{)}\\ &\qquad\qquad\texttt{(}I_1{'}\ BV_1{'}\texttt{)}\ \ldots\ \texttt{(}I_n{'}\ BV_n{'}\texttt{))}\\ &\qquad E_{body}\texttt{)}\\ &\text{where } (\{I_1,\ldots,I_m\} \cup_{i=1}^{m} \mathit{FreeIds}[\![BV_i]\!]) \cap \{I_1{'},\ldots,I_n{'}\} = \{\}\end{aligned}\ , \qquad [\textit{combine-cycrecs}]$$

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.

```
(cycrec ((a 17)
         (b (lambda (x) (@mprod x d)))
         (c (@mprod d))
         (d (@mprod a b c d)))
  d)
```

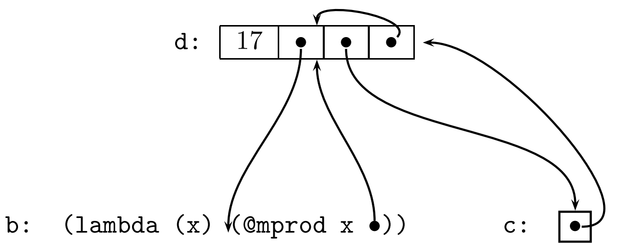

Figure 17.6: A sample `cycrec` expression and the cyclic structure it denotes.

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

$$\big(init\ [in_{1_{fix}}, \ldots, in_{n_{fix}}]\ \ e_{fix}\ \ s_{fix}\big)\,,$$

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

$$
\begin{array}{rl}
c \in & \mathit{Computation} = \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store}) \\
\delta \in & \mathit{Denotable} = \mathit{Location} \\
\sigma \in & \mathit{Storable} = \mathit{Value} \\
mt \in & \mathit{MProd} = \mathit{Location}* \\
p \in & \mathit{Procedure} = \mathit{Denotable}* \to \mathit{Computation} \\
v \in & \mathit{Value} = \mathit{Procedure} + \mathit{MProd} + \ldots \\
in \in & \mathit{Initializer} = \mathit{Environment} \to \mathit{Store} \to \mathit{Store} \\
vi \in & \mathit{VI} = \mathit{Value} \times \mathit{Initializer}
\end{array}
$$

$$
\begin{array}{l}
\mathit{extend}* : \text{Identifier}* \to \mathit{Denotable}* \to \mathit{Environment} \to \mathit{Environment} \\
\mathit{extend}* \ [\,]_{\text{Identifier}} \ [\,]_{\mathit{Denotable}} \ \ e \ = e \\
\mathit{extend}* \ \ (I_1 \ . \ I_{rest}*) \ \ (\delta_1 \ . \ \delta_{rest}*) \ \ e \ = \text{extend}* \ I_{rest}* \ \delta_{rest}* \ \ ([\delta : e]I)
\end{array}
$$

$$
\begin{array}{l}
\mathit{assign}* : \mathit{Location}* \to \mathit{Storable}* \to \mathit{Store} \to \mathit{Store} \\
\mathit{assign}* \ [\,]_{\mathit{Location}} \ [\,]_{\mathit{Storable}} \ \ s \ = s \\
\mathit{assign}* \ \ (l_1 \ . \ l_{rest}*) \ \ (\sigma_1 \ . \ \sigma_{rest}*) \ \ s \ = \mathit{assign}* \ l_{rest}* \ \sigma_{rest}* \ \ (\mathit{assign} \ l \ \sigma \ \ s)
\end{array}
$$

$$
\begin{array}{l}
\mathit{init} : \mathit{Initializer}* \to \mathit{Environment} \to \mathit{Store} \to \mathit{Store} \\
\mathit{init} \ [\,]_{\mathit{Initializer}} \ \ e \ \ s \ = s \\
\mathit{init} \ \ (in_1 \ . \ in_{rest}*) \ \ e \ \ s \ = \text{init} \ in_{rest}* \ \ e \ \ (in \ \ e \ \ s)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{BV} : \text{BindingValue} \to \mathit{Environment} \to \mathit{Store} \to (\mathit{Value} \times \mathit{Store} \times \mathit{Initializer}) \\
\mathcal{BV}[\![L]\!] \ \ e \ \ s \ = \langle \mathcal{L}[\![L]\!], s, \lambda e's' \,.\, s' \rangle \\
\mathcal{BV}[\![\texttt{(lambda (}I*\texttt{)} \ \ E_{body}\texttt{)}]\!] \ \ e \ \ s \ = \\
\quad \langle (\mathit{Procedure} \mapsto \mathit{Value} \ \ (\lambda\delta* \,.\, \mathcal{E}[\![E_{body}]\!](\text{extend}* \ \ I* \ \ \delta* \ \ e))), s, \lambda e's' \,.\, s' \rangle \\
\mathcal{BV}[\![\texttt{(primop mprod} \ DV_1 \ \ \ldots \ \ DV_n\texttt{)}]\!] \ \ e \ \ s \ = \\
\quad \mathbf{let} \ \langle l*, s' \rangle \ \ \mathbf{be} \ \ (\mathit{fresh\text{-}locs} \ \ n \ \ s) \\
\quad\ \mathbf{in} \ \langle (\mathit{MProd} \mapsto \mathit{Value} \ \ l*), \ s', \\
\qquad\quad \lambda e' \,. \ \ \mathit{with\text{-}values} \ \ (\mathcal{E}*[\![[DV_1, \ldots, DV_n]]\!] \ \ e') \ \ (\text{assign}* \ \ l*) \ \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{BV}* : \text{BindingValue}* \to \mathit{Environment} \to \mathit{Store} \to ((\mathit{Value} \times \mathit{Initializer})* \times \mathit{Store}) \\
\mathcal{BV}* \ \ [\,]_{\text{BindingValue}} \ \ e \ \ s \ = s \\
\mathcal{BV}* \ \ (BV_1 \ \ . \ \ BV_{rest}*) \ \ e \ \ s \ = \\
\quad \mathbf{let} \ \langle v_1, s_1, in_1 \rangle \ \ \mathbf{be} \ \mathcal{BV} \ \ BV_1 \ \ e \ \ s \\
\quad\ \mathbf{in\,let} \ \langle vi_{rest}, s_n \rangle \ \ \mathbf{be} \ \mathcal{BV}* \ \ BV_{rest} \ \ e \ \ s_1 \ \ \mathbf{in} \ \langle \langle v_1, in_1 \rangle \ \ . \ \ vi_{rest}, s_n \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(cycrec ((}I_1 \ \ BV_1\texttt{)} \ \ldots\texttt{(}I_n \ \ BV_n\texttt{))} \ \ E_{body}\texttt{)}]\!] \ = \\
\quad \lambda es \,. \ \ \mathbf{let} \ \langle e_{\mathit{fix}}, s_{\mathit{fix}}, \langle in_{1_{\mathit{fix}}}, \ldots, in_{n_{\mathit{fix}}} \rangle \rangle \ \ \mathbf{be} \\
\qquad\qquad \mathbf{fix}_{\mathit{Environment} \times \mathit{Store} \times (\mathit{Initializer}^n)} \\
\qquad\qquad\quad (\lambda \langle e_{\mathit{fix}}, s_{\mathit{fix}}, \langle in_{1_{\mathit{fix}}}, \ldots, in_{n_{\mathit{fix}}} \rangle \rangle \,. \\
\qquad\qquad\qquad \mathbf{let} \ \langle [\langle v_1, in_1 \rangle, \ldots, \langle v_n, in_n \rangle], s' \rangle \ \ \mathbf{be} \ \ (\mathcal{BV}* \ \ [BV_1, \ldots, BV_n] \ \ e_{\mathit{fix}} \ \ s) \\
\qquad\qquad\qquad\ \mathbf{in\,let} \ \langle [l_1, \ldots, l_n], s'' \rangle \ \ \mathbf{be} \ \ (\mathit{fresh\text{-}locs} \ \ n \ \ s') \\
\qquad\qquad\qquad\quad \mathbf{in} \ \langle [I_1 : l_1] \ldots [I_n : l_n]e, \\
\qquad\qquad\qquad\qquad \text{assign}* \ \ [l_1, \ldots, l_n] \ \ [v_1, \ldots, v_n] \ \ s'', \\
\qquad\qquad\qquad\qquad \langle in_1, \ldots, in_n \rangle \rangle \\
\qquad\quad \mathbf{in} \, \mathcal{E}[\![E_{body}]\!] \ \ e_{\mathit{fix}} \ \ \big(\text{init} \ \ [in_{1_{\mathit{fix}}}, \ldots, in_{n_{\mathit{fix}}}] \ \ e_{\mathit{fix}} \ \ s_{\mathit{fix}}\big)
\end{array}
$$

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

$$\begin{aligned} T ::= {} & \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{char} \mid I \mid \nu \\ & \mid \texttt{(listof}\ T\texttt{)} \mid \texttt{(mprodof}\ T^{*}\ [\texttt{...}]\texttt{)} \\ & \mid \texttt{(->}\ \texttt{(}T_{arg}{}^{*}\texttt{)}\ T_{body}\texttt{)} \mid \texttt{(tletrec}\ \texttt{((}I_{name}\ T_{defn}\texttt{)}^{*}\texttt{)}\ T_{body}\texttt{)} \\ & \mid \texttt{(forall}\ \texttt{(}I^{*}\texttt{)}\ T\texttt{)} \mid \texttt{(exists}\ \texttt{(}I^{*}\texttt{)}\ T\texttt{)} \end{aligned}$$

Subtyping

$$\texttt{(mprodof}\ T_1\ \ldots\ T_n\texttt{)} \sqsubseteq \texttt{(mprodof}\ T_1\ \ldots\ T_k\ \texttt{...)}, \text{ where } n \geq k \qquad [\textit{mprod-}\sqsubseteq]$$

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 [∀-*elim*] rule and the [∃-*intro*] rule.

A a value with universal type can be introduced anywhere and then later be implicitly projected at various types. For example, in

$\vdash \texttt{\#u} : \texttt{unit}$ $[\mathit{unit}]$ $\vdash N : \texttt{int}$ $[\mathit{int}]$ $\vdash B : \texttt{bool}$ $[\mathit{bool}]$ $\vdash H : \texttt{char}$ $[\mathit{char}]$

$\vdash \texttt{(error } I\texttt{)} : T$ $[\mathit{error}]$ $A \vdash I : A(I)$, where $I \in \mathit{dom}(A)$ $[\mathit{var}]$

$$\frac{A \vdash E_{rhs} : A(I)}{A \vdash \texttt{(set! } I \ E_{rhs}\texttt{)} : \texttt{unit}}, \text{ where } I \in \mathit{dom}(A) \qquad [\mathit{assign}]$$

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

$$\frac{A[I_1 : T_1, \ \ldots, \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{)} \ E_{body}\texttt{)} : \texttt{(-> (}T_1 \ \ldots \ T_n\texttt{)} \ T_{body}\texttt{)}} \qquad [\texttt{->}\text{-}\mathit{intro}]$$

$$\frac{A \vdash E_{rator} : \texttt{(-> (}T_1 \ \ldots \ T_n\texttt{)} \ T_{result}\texttt{)} \ ; \ \forall_{i=1}^{n} . \ A \vdash E_i : T_i}{A \vdash \texttt{(call } E_{rator} \ E_1 \ \ldots \ E_n\texttt{)} : T_{result}} \qquad [\texttt{->}\text{-}\mathit{elim}]$$

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

$$\frac{\forall_{i=1}^{n} . \ A' \vdash BV_i : T_i \ ; \ A' \vdash E_{body} : T_{body}}{A \vdash \texttt{(cycrec ((}I_1 \ BV_1\texttt{)} \ \ldots \ \texttt{(}I_n \ BV_n\texttt{))} \ E_{body}\texttt{)} : T_{body}} \qquad [\mathit{cycrec}]$$

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

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

$$\frac{A \vdash E : T}{A \vdash E : \texttt{(forall (}I_1 \ \ldots \ I_n\texttt{)} \ T\texttt{)}} \qquad [\forall\text{-}\mathit{intro}]$$

where E is pure *[purity restriction]*
$\{I_1, \ldots, I_k\} \cap \{(FTV \ A(I)) \mid I \in \mathit{FreeIds}[\![E]\!]\} = \{\}$ *[import restriction]*

$$\frac{A \vdash E : \texttt{(forall (}I_1 \ \ldots \ I_n\texttt{)} \ T\texttt{)}}{A \vdash E : ([T_i/I_i]_{i=1}^{n}) \ T} \qquad [\forall\text{-}\mathit{elim}]$$

$$\frac{A \vdash E : ([T_i/I_i]_{i=1}^{n}) \ T}{A \vdash E : \texttt{(exists (}I_1 \ \ldots \ I_n\texttt{)} \ T\texttt{)}} \qquad [\exists\text{-}\mathit{intro}]$$

where $\{I_1, \ldots, I_k\} \cap \{(FTV \ A(I)) \mid I \in \mathit{FreeIds}[\![E]\!]\} = \{\}$ *[import restriction]*

$$\frac{A \vdash E : \texttt{(exists (}I_1 \ \ldots \ I_n\texttt{)} \ T\texttt{)}}{A \vdash E : ([\nu_i/I_i]_{i=1}^{n}) \ T}, \text{ where } \nu_1, \ldots, \nu_n \text{ are fresh nonce types.} \qquad [\exists\text{-}\mathit{elim}]$$

$$\frac{A \vdash E : T}{A \vdash E : T'}, \text{ where } T \sqsubseteq T' \qquad [\mathit{subtype}]$$

$$\frac{\{I_1 : \texttt{int}, \ldots, I_n : \texttt{int}\} \vdash E_{body} : T}{\vdash \texttt{(silk (}I_1 \ \ldots \ I_n\texttt{)} \ E_{body}\texttt{)} : T} \qquad [\mathit{prog}]$$

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 metalanguage 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 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 $\mathit{mapsub}_{\mathit{Silk}}$ 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 $\mathit{mapsub}_{\mathit{Silk}}$.

$$\mathit{tf} \in \mathit{Transform}_{\mathit{Silk}} = \mathrm{Exp}_{\mathit{Silk}} \rightarrow \mathrm{Exp}_{\mathit{Silk}}$$

$$\mathit{mapsub}_{\mathit{Silk}} : \mathrm{Exp}_{\mathit{Silk}} \rightarrow \mathit{Transform}_{\mathit{Silk}} \rightarrow \mathrm{Exp}_{\mathit{Silk}}$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket L \rrbracket\ \mathit{tf} = L$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket I \rrbracket\ \mathit{tf} = I$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(error}\ I_{\mathit{msg}}\texttt{)} \rrbracket\ \mathit{tf} = \texttt{(error}\ I_{\mathit{msg}}\texttt{)}$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(set!}\ I_{\mathit{var}}\ E_{\mathit{rhs}}\texttt{)} \rrbracket\ \mathit{tf} = \texttt{(set!}\ I_{\mathit{var}}\ (\mathit{tf}\ E_{\mathit{rhs}})\texttt{)}$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(if}\ E_{\mathit{test}}\ E_{\mathit{then}}\ E_{\mathit{else}}\texttt{)} \rrbracket\ \mathit{tf} = \texttt{(if}\ (\mathit{tf}\ E_{\mathit{test}})\ (\mathit{tf}\ E_{\mathit{then}})\ (\mathit{tf}\ E_{\mathit{else}})\texttt{)}$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{\mathit{body}}\texttt{)} \rrbracket\ \mathit{tf} = \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ (\mathit{tf}\ E_{\mathit{body}})\texttt{)}$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(call}\ E_{\mathit{rator}}\ E_1\ \ldots\ E_n\texttt{)} \rrbracket\ \mathit{tf} = \texttt{(call}\ (\mathit{tf}\ E_{\mathit{rator}})\ (\mathit{tf}\ E_1)\ \ldots\ (\mathit{tf}\ E_n)\texttt{)}$$

$$\begin{aligned}&\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{\mathit{body}}\texttt{)} \rrbracket\ \mathit{tf}\\ &\quad = \texttt{(let ((}I_1\ (\mathit{tf}\ E_1)\texttt{)}\ \ldots\ \texttt{(}I_n\ (\mathit{tf}\ E_n)\texttt{))}\ (\mathit{tf}\ E_{\mathit{body}})\texttt{)}\end{aligned}$$

$$\begin{aligned}&\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(cycrec ((}I_1\ \mathit{BV}_1\texttt{)}\ \ldots\ \texttt{(}I_n\ \mathit{BV}_n\texttt{))}\ E_{\mathit{body}}\texttt{)} \rrbracket\ \mathit{tf}\\ &\quad = \texttt{(cycrec ((}I_1\ (\mathit{tf}\ \mathit{BV}_1)\texttt{)}\ \ldots\ \texttt{(}I_n\ (\mathit{tf}\ \mathit{BV}_n)\texttt{))}\ (\mathit{tf}\ E_{\mathit{body}})\texttt{)}\end{aligned}$$

$$\mathit{mapsub}_{\mathit{Silk}}\llbracket \texttt{(primop}\ O\ E_1\ \ldots\ E_n\texttt{)} \rrbracket\ \mathit{tf} = \texttt{(primop}\ O\ (\mathit{tf}\ E_1)\ \ldots\ (\mathit{tf}\ E_n)\texttt{)}$$

Figure 17.10: The $\mathit{mapsub}_{\mathit{Silk}}$ function simplifies the specification of purely structural transformations.

In the `cycrec` clause for $\mathit{mapsub}_{\mathit{Silk}}$, we take the liberty of applying the transformation tf directly to the binding values $\mathit{BV}_1 \ldots \mathit{BV}_n$. 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 $(\mathit{tf}\ \mathit{BV}_i)$. If the result is not in the BindingValue domain, then the `cycrec` form is not syntactically well-formed. So whenever $\mathit{mapsub}_{\mathit{Silk}}$ is applied to `cycrec` forms, we must argue that tf maps elements of BindingValue to elements of BindingValue.

As an example of $\mathit{mapsub}_{\mathit{Silk}}$, consider a transformation $\mathcal{IT}$ that rewrites every occurrence of $\texttt{(if (primop not } E_1\texttt{) } E_2 \; E_3\texttt{)}$ to $\texttt{(if } E_1 \; E_3 \; E_2\texttt{)}$. 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 $\mathit{mapsub}_{\mathit{Silk}}$ in the following definition:

$$\mathcal{IT} : \mathrm{Exp}_{\mathit{Silk}} \to \mathrm{Exp}_{\mathit{Silk}}$$

$$\mathcal{IT}\llbracket\texttt{(if (primop not } E_1\texttt{) } E_2 \; E_3\texttt{)}\rrbracket = \texttt{(if } (\mathcal{IT}\llbracket E_1\rrbracket) \; (\mathcal{IT}\llbracket E_3\rrbracket) \; (\mathcal{IT}\llbracket E_2\rrbracket)\texttt{)}$$

$$\mathcal{IT}\llbracket E\rrbracket = \mathit{mapsub}_{\mathit{Silk}}\llbracket E\rrbracket \; \mathcal{IT}, \text{ for all other expressions } E.$$

It is not hard to show that $\mathcal{IT}$ transforms every binding value to a binding value, so $\mathit{mapsub}_{\mathit{Silk}}$ is sensible for `cycrec`.

The $\mathit{mapsub}_{\mathit{Silk}}$ function only works for transforming one Silk expression to another. It is straightforward to define a similar $\mathit{mapsub}_{\mathit{FL/R}}$ 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 $\mathit{subexps}_{\mathit{FL/R}}$ function that returns a sequence of all children expressions of a given FL/R$_{\text{Tortoise}}$ expression. It is straightforward to define a similar $\mathit{subexps}_{\mathit{Silk}}$ function for Silk expressions.

$$\mathit{subexps}_{\mathit{FL/R}} : \mathrm{Exp}_{\mathit{FL/R}} \to \left(\mathrm{Exp}_{\mathit{FL/R}}{}^{*}\right)$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket L\rrbracket = [\,]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket I\rrbracket = [\,]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(error } I_{\mathit{msg}}\texttt{)}\rrbracket = [\,]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(set! } I_{\mathit{var}} \; E_{\mathit{rhs}}\texttt{)}\rrbracket = [E_{\mathit{rhs}}]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(if } E_{\mathit{test}} \; E_{\mathit{then}} \; E_{\mathit{else}}\texttt{)}\rrbracket = [E_{\mathit{test}}, E_{\mathit{then}}, E_{\mathit{else}}]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(lambda (}I_1 \; \ldots \; I_n\texttt{) } E_{\mathit{body}}\texttt{)}\rrbracket = [E_{\mathit{body}}]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(}E_{\mathit{rator}} \; E_1 \; \ldots \; E_n\texttt{)}\rrbracket = [E_{\mathit{rator}}, E_1, \ldots, E_n]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(primop } O \; E_1 \; \ldots \; E_n\texttt{)}\rrbracket = [E_1, \ldots, E_n]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(let ((}I_1 \; E_1\texttt{) } \ldots \texttt{ (}I_n \; E_n\texttt{)) } E_{\mathit{body}}\texttt{)}\rrbracket$$
$$= [E_1, \ldots, E_n, E_{\mathit{body}}]$$

$$\mathit{subexps}_{\mathit{FL/R}}\llbracket\texttt{(cycrec ((}I_1 \; BV_1\texttt{) } \ldots \texttt{ (}I_n \; BV_n\texttt{)) } E_{\mathit{body}}\texttt{)}\rrbracket$$
$$= [BV_1, \ldots, BV_n, E_{\mathit{body}}]$$

Figure 17.11: The $\mathit{subexps}_{\mathit{FL/R}}$ 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$_{\textsc{Tortoise}}$ 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$_{\textsc{Tortoise}}$ 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 conses.

```
(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$_{\textsc{Tortoise}}$ (and Silk) programs are allowed to proceed through the rest of the compiler. Because type reconstruction for FL/R$_{\textsc{Tortoise}}$ 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))))
```

$\mathcal{GW} : \text{Program}_{FL/R} \to \text{Program}_{FL/R}$

$\mathcal{GW}[\![\texttt{(flr (}I_1 \ \ldots \ I_n\texttt{)} \ E_{body}\texttt{)}]\!] = \texttt{(flr (}I_1 \ \ldots \ I_n\texttt{)} \ (wrap[\![E_{body}]\!] \ \texttt{(}\textit{\texttt{FreeIds}}[\![E_{body}]\!]\texttt{)))}$

$wrap : \text{Exp}_{FL/R} \to \mathcal{P}(\text{Identifier}) \to \text{Exp}_{FL/R}$

$wrap[\![E]\!] \ \{O_1, \ldots, O_n\} = \texttt{(let ((}O_1 \ \mathcal{ABS}[\![O_1]\!]\texttt{)} \ \ldots \ \texttt{(}O_1 \ \mathcal{ABS}[\![O_n]\!]\texttt{))} \ E\texttt{)}$

$\mathcal{ABS} : \text{Primop}_{FL/R} \to \text{Abstraction}$

$\mathcal{ABS}[\![O]\!] = \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{) (primop } O \ I_1 \ \ldots \ I_n\texttt{))}$
where $I_1, \ldots, I_n$ are fresh and $(typeof \ [\![O]\!] \ A_{std_{FL/R}}) = \texttt{(-> (}T_1 \ \ldots \ T_n\texttt{)} \ T_{res}\texttt{)}$.

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/$\mathrm{R}_{\mathrm{TORTOISE}}$, 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/$\mathrm{R}_{\mathrm{TORTOISE}}$, 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

$\mathcal{GI}_{prog} : \text{Program}_{FL/R} \rightarrow \text{Program}_{FL/R}$

$\mathcal{GI}_{prog}\llbracket P \rrbracket = \texttt{(flr (}I_1 \ \ldots \ I_n\texttt{)} \ (wrap\llbracket \mathcal{GI}_{exp}\llbracket E_{body} \rrbracket \ IS_{immuts} \rrbracket \ IS_{muts}\texttt{))}$
where $P = \texttt{(flr (}I_1 \ \ldots \ I_n\texttt{)} \ E_{body}\texttt{)}$, $IS_{muts} = MutIds_{prog}\llbracket P \rrbracket$,
$IS_{immuts} = (FreeIds\llbracket P \rrbracket) - IS_{muts}$, *wrap* is defined in Figure 17.14,
and $MutIds_{prog}$ is defined in Figure 17.15.

$\mathcal{GI}_{exp} : \text{Exp}_{FL/R} \rightarrow \text{IdSet} \rightarrow \text{Exp}_{FL/R}$

$$\begin{aligned}
&\mathcal{GI}_{exp}\llbracket \texttt{(}I_{rator} \ E_1 \ \ldots \ E_n\texttt{)} \rrbracket \ IS \\
&\quad = \ \textbf{if} \ I_{rator} \in IS \ \textbf{then} \ \texttt{(primop} \ I_{rator} \ (\mathcal{GI}_{exp}\llbracket E_1 \rrbracket \ IS) \ \ldots \ (\mathcal{GI}_{exp}\llbracket E_n \rrbracket \ IS)\texttt{)} \\
&\qquad\quad \textbf{else} \ \texttt{(}I_{rator} \ (\mathcal{GI}_{exp}\llbracket E_1 \rrbracket \ IS) \ \ldots \ (\mathcal{GI}_{exp}\llbracket E_n \rrbracket \ IS)\texttt{)} \ \textbf{fi}
\end{aligned}$$

$$\mathcal{GI}_{exp}\llbracket I \rrbracket \ IS \ = \textbf{if} \ I \in IS \ \textbf{then} \ \mathcal{ABS}\llbracket I \rrbracket \ \textbf{else} \ I \ \textbf{fi}$$

$$\begin{aligned}
&\mathcal{GI}_{exp}\llbracket \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{)} \ E_{body}\texttt{)} \rrbracket \ IS \\
&\quad = \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{)} \ (\mathcal{GI}_{exp}\llbracket E_{body} \rrbracket \ (IS - \{I_1, \ldots, I_n\}))\texttt{)}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{GI}_{exp}\llbracket \texttt{(let ((}I_1 \ E_1\texttt{)} \ \ldots \ \texttt{(}I_n \ E_n\texttt{))} \ E_{body}\texttt{)} \rrbracket \ IS \\
&\quad = \texttt{(let} \ ((I_1 \ (\mathcal{GI}_{exp}\llbracket E_1 \rrbracket \ IS)) \ldots (I_n \ (\mathcal{GI}_{exp}\llbracket E_n \rrbracket \ IS))) \\
&\qquad\quad (\mathcal{GI}_{exp}\llbracket E_{body} \rrbracket \ (IS - \{I_1, \ldots, I_n\})))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{GI}_{exp}\llbracket \texttt{(funrec ((}I_1 \ AB_1\texttt{)} \ \ldots \ \texttt{(}I_n \ AB_n\texttt{))} \ E_{body}\texttt{)} \rrbracket \ IS \\
&\quad = \texttt{(funrec ((}I_1 \ (\mathcal{GI}_{exp}\llbracket AB_1 \rrbracket \ IS'\texttt{))} \ \ldots \ \texttt{(}I_n \ (\mathcal{GI}_{exp}\llbracket AB_n \rrbracket \ IS'\texttt{)))} \\
&\qquad\quad (\mathcal{GI}_{exp}\llbracket E_{body} \rrbracket \ IS'\texttt{))} \\
&\quad \text{where } IS' \ = \ IS \ - \ \{I_1, \ldots, I_n\}
\end{aligned}$$

$$\mathcal{GI}_{exp}\llbracket E \rrbracket \ IS \ = \ \text{mapsub}_{FL/R}\llbracket E \rrbracket \ (\lambda E_{sub} \,.\, \mathcal{GI}_{exp}\llbracket E_{sub} \rrbracket \ IS)$$

Figure 17.14: The inlining approach to globalization.

$IS \in \text{IdSet} = \mathcal{P}(\text{Identifier})$

$\mathit{MutIds}_{prog} : \text{Program}_{FL/R} \rightarrow \text{IdSet}$

$\mathit{MutIds}_{prog}\llbracket\texttt{(flr (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\rrbracket = (\mathit{MutIds}\llbracket E_{body}\rrbracket) - \{I_1, \ldots, I_n\}$

$\mathit{MutIds} : \text{Exp}_{FL/R} \rightarrow \text{IdSet}$

$\mathit{MutIds}\llbracket\texttt{(set!}\ I\ E\texttt{)}\rrbracket = \{I\} \cup \mathit{MutIds}\llbracket E\rrbracket$

$\mathit{MutIds}\llbracket\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\rrbracket = (\mathit{MutIds}\llbracket E_{body}\rrbracket) - \{I_1, \ldots, I_n\}$

$\mathit{MutIds}\llbracket\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}\rrbracket$
$\quad = (\cup_{i=1}^{n}\mathit{MutIds}\llbracket E_i\rrbracket) \cup \mathit{MutIds}\llbracket E_{body}\rrbracket - \{I_1, \ldots, I_n\}$

$\mathit{MutIds}\llbracket\texttt{(funrec ((}I_1\ AB_1\texttt{)}\ \ldots\ \texttt{(}I_n\ AB_n\texttt{))}\ E_{body}\texttt{)}\rrbracket$
$\quad = (\cup_{i=1}^{n}\mathit{MutIds}\llbracket AB_i\rrbracket \cup \mathit{MutIds}\llbracket E_{body}\rrbracket) - \{I_1, \ldots, I_n\}$

$\mathit{MutIds}\llbracket E\rrbracket = \textbf{let}\ [E_1, \ldots, E_n]\ \textbf{be}\ \mathit{subexps}_{FL/R}\llbracket E\rrbracket\ \textbf{in}\ \cup_{i=1}^{n}\mathit{MutIds}\llbracket E_i\rrbracket$, 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$_{\textsc{Tortoise}}$ 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$_{\textsc{Tortoise}}$ 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$_{\textsc{Tortoise}}$ or Silk) to another expression in the *same* language. But $\mathcal{T}_{\text{exp}}$ translates a FL/R$_{\textsc{Tortoise}}$ expression to a Silk expression.

The precondition for $\mathcal{T}_{\text{prog}}$ requires a closed FL/R$_{\textsc{Tortoise}}$ 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$_{\textsc{Tortoise}}$. This can be formally proved by showing how a FL/R$_{\textsc{Tortoise}}$ 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$_{\textsc{Tortoise}}$ 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/RTortoise program.

Postconditions: The output of $\mathcal{T}_{\text{prog}}$ is a valid kernel Silk program.

$\mathcal{T}_{\text{prog}}\llbracket\texttt{(flr (}I_1 \ \ldots \ I_n\texttt{) }E_{body}\texttt{)}\rrbracket = \texttt{(silk (}I_1 \ \ldots \ I_n\texttt{) (}\mathcal{T}_{\text{exp}}\llbracket E_{body}\rrbracket\texttt{))}$

$\mathcal{T}_{\text{exp}} : \text{Exp}_{FL/R_{\text{Tortoise}}} \rightarrow \text{Exp}_{Silk}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(funrec ((}I_1 \ AB_1\texttt{) ... (}I_n \ AB_n\texttt{)) }E_{body}\texttt{)}\rrbracket$
$= \texttt{(cycrec ((}I_1 \ \mathcal{T}_{\text{exp}}\llbracket AB_1\rrbracket\texttt{) ... (}I_n \ \mathcal{T}_{\text{exp}}\llbracket AB_n\rrbracket\texttt{)) (}\mathcal{T}_{\text{exp}}\llbracket E_{body}\rrbracket\texttt{))}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(primop cell }E_1\texttt{)}\rrbracket = \texttt{(primop mprod }E_1\texttt{)}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(primop \^{} }E_{cell}\texttt{)}\rrbracket = \texttt{(primop (mget 1) }E_{cell}\texttt{)}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(primop := }E_{cell} \ E_{new}\texttt{)}\rrbracket = \texttt{(primop (mset! 1) }E_{cell} \ E_{new}\texttt{)}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(primop pair }E_1 \ E_2\texttt{)}\rrbracket = \texttt{(primop mprod }E_1 \ E_2\texttt{)}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(primop fst }E_{pair}\texttt{)}\rrbracket = \texttt{(primop (mget 1) }E_{pair}\texttt{)}$

$\mathcal{T}_{\text{exp}}\llbracket\texttt{(primop snd }E_{pair}\texttt{)}\rrbracket = \texttt{(primop (mget 2) }E_{pair}\texttt{)}$

All other cases of $\mathcal{T}_{\text{exp}}$ are purely structural.

Figure 17.17: Transformation translating FL/RTortoise 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:

$$\begin{array}{lll} E ::= \ldots & & \\ & \mid \texttt{(one}\ I_{tag}\ \ E\texttt{)} & \text{[Oneof Intro]} \\ & \mid \texttt{(tagcase}\ E_{disc}\ \ I_{val}\ \ \texttt{(}I_{tag}\ \ E_{body}\texttt{)*}\ [\texttt{(else}\ E_{else}\texttt{)}]\texttt{)} & \text{[Oneof Elim]} \end{array}$$

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*`)`,

$\mathcal{AC}_{prog} : \text{Program}_{Silk} \to \text{Program}_{Silk}$

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.

$$\begin{array}{l}
\mathcal{AC}_{prog}\llbracket\texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\rrbracket \\
\quad = \texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ (\textit{wrap-cells}\ \textit{IS}_{muts}\ (\mathcal{AC}_{exp}\llbracket E_{body}\rrbracket\ \textit{IS}_{muts})\texttt{))}
\end{array}$$

where $\textit{IS}_{muts} = \textit{MutIds}\llbracket E_{body}\rrbracket$ and *MutIds* is a version of the function defined in Figure 17.15 adapted to SILK.

$\mathcal{AC}_{exp} : \text{Exp}_{Silk} \to \text{IdSet} \to \text{Exp}_{Silk}$

$\mathcal{AC}_{exp}\llbracket I\rrbracket\ \textit{IS} = \textbf{if}\ I \in \textit{IS}\ \textbf{then}\ \texttt{(@mget 1}\ I\texttt{)}\ \textbf{else}\ I\ \textbf{fi}$

$\mathcal{AC}_{exp}\llbracket\texttt{(set!}\ I\ E\texttt{)}\rrbracket\ \textit{IS} = \texttt{(@mset! 1}\ I\ (\mathcal{AC}_{exp}\llbracket E\rrbracket\ \textit{IS})\texttt{)}$

$$\begin{array}{l}
\mathcal{AC}_{exp}\llbracket\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\rrbracket\ \textit{IS} \\
\quad = \textbf{let}\ \langle \textit{IS}_m, \textit{IS}_i\rangle\ \textbf{be}\ (\textit{partition}\ \{I_1,\ldots,I_n\}\ [E_{body}]) \\
\qquad \textbf{in}\ \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)} \\
\qquad\qquad (\textit{wrap-cells}\ \textit{IS}_m\ (\mathcal{AC}_{exp}\llbracket E_{body}\rrbracket\ ((\textit{IS} \cup \textit{IS}_m) - \textit{IS}_i)))\texttt{)}
\end{array}$$

$$\begin{array}{l}
\mathcal{AC}_{exp}\llbracket\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}\rrbracket\ \textit{IS} \\
\quad = \textbf{let}\ \langle \textit{IS}_m, \textit{IS}_i\rangle\ \textbf{be}\ (\textit{partition}\ \{I_1,\ldots,I_n\}\ [E_{body}]) \\
\qquad \textbf{in}\ \texttt{(let ((}I_1\ (\textit{maybe-cell}\ I_1\ \textit{IS}_m\ (\mathcal{AC}_{exp}\llbracket E_1\rrbracket\ \textit{IS}))\texttt{)} \\
\qquad\qquad\qquad \ldots\ \texttt{(}I_n\ (\textit{maybe-cell}\ I_n\ \textit{IS}_m\ (\mathcal{AC}_{exp}\llbracket E_n\rrbracket\ \textit{IS}))\texttt{))} \\
\qquad\qquad (\mathcal{AC}_{exp}\llbracket E_{body}\rrbracket\ ((\textit{IS} \cup \textit{IS}_m) - \textit{IS}_i))\texttt{)}
\end{array}$$

$$\begin{array}{l}
\mathcal{AC}_{exp}\llbracket\texttt{(cycrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}\rrbracket\ \textit{IS} \\
\quad = \textbf{let}\ \langle \textit{IS}_m, \textit{IS}_i\rangle\ \textbf{be}\ (\textit{partition}\ \{I_1,\ldots,I_n\}\ [E_1,\ldots,E_n,E_{body}]) \\
\qquad \textbf{in}\ \texttt{(cycrec ((}I_1\ (\textit{maybe-cell}\ I_1\ \textit{IS}_m\ (\mathcal{AC}_{exp}\llbracket E_1\rrbracket\ \textit{IS}'))\texttt{)} \\
\qquad\qquad\qquad\quad \ldots\ \texttt{(}I_n\ (\textit{maybe-cell}\ I_n\ \textit{IS}_m\ (\mathcal{AC}_{exp}\llbracket E_n\rrbracket\ \textit{IS}'))\texttt{))} \\
\qquad\qquad (\mathcal{AC}_{exp}\llbracket E_{body}\rrbracket\ \textit{IS}')\texttt{)} \\
\quad \text{where}\ \textit{IS}' = ((\textit{IS} \cup \textit{IS}_m) - \textit{IS}_i).
\end{array}$$

$\mathcal{AC}_{exp}\llbracket E\rrbracket\ \textit{IS} = \textit{mapsub}_{Silk}\llbracket E\rrbracket\ (\lambda E_{sub}\,.\,\mathcal{AC}_{exp}\llbracket E_{sub}\rrbracket\ \textit{IS})$, otherwise.

$\textit{wrap-cells} : \text{IdSet} \to \text{Exp}_{Silk} \to \text{Exp}_{Silk}$

$\textit{wrap-cells}\ \{I_1 \ldots I_n\}\ E = \texttt{(let ((}I_1\ \texttt{(@mprod}\ I_1\texttt{))}\ \ldots\ \texttt{(}I_n\ \texttt{(@mprod}\ I_n\texttt{)))}\ E\texttt{)}$

$\textit{partition} : \text{IdSet} \to {\text{Exp}_{Silk}}^{*} \to (\text{IdSet} \times \text{IdSet})$

$\textit{partition}\ \textit{IS}\ [E_1 \ldots E_n] = \textbf{let}\ \textit{IS}_M\ \textbf{be}\ \cup_{i=1}^{k}(\textit{MutIds}\llbracket E_i\rrbracket)\ \textbf{in}\ \langle \textit{IS} \cup \textit{IS}_M, \textit{IS} - \textit{IS}_M\rangle$

$\textit{maybe-cell} : \text{Identifier} \to \text{IdSet} \to \text{Exp}$

$\textit{maybe-cell}\ I\ \textit{IS}\ E = \textbf{if}\ I \in \textit{IS}\ \textbf{then}\ \texttt{(@mprod}\ E\texttt{)}\ \textbf{else}\ E\ \textbf{fi}$

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 $\mathcal{AC}_{exp}$ functions 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

$re \in \mathit{RenEnv} = \text{Identifier} \to \text{Identifier}$

$\mathit{rbind} : \text{Identifier} \to \text{Identifier} \to \mathit{RenEnv} \to \mathit{RenEnv}$
$\quad = \lambda I_{old}\ I_{new}\ re\ .\ \lambda I_{key}\ .\ \textbf{if}\ (\mathit{same\text{-}identifier?}\ I_{key}\ I_{old})\ \textbf{then}\ I_{new}\ \textbf{else}\ (re\ I_{key})$

$\mathit{rbind}\ I_{old}\ I_{new}\ re$ 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

$\mathcal{R}_{\text{prog}} : \text{Program}_{\mathit{Silk}} \to \text{Program}_{\mathit{Silk}}$

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.

$\mathcal{R}_{\text{prog}}\llbracket\texttt{(silk (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\rrbracket$
$\quad = \texttt{(silk (}I_1{}'\ \ldots\ I_n{}'\texttt{)}\ (\mathcal{R}_{\text{exp}}\llbracket E_{body}\rrbracket\ ([I_1 : I_1{}'] \ldots [I_n : I_n{}']\,(\lambda I\,.\,I)))\texttt{)}$,
where $I_1{}' \ldots I_n{}'$ are fresh.

$\mathcal{R}_{\text{exp}} : \text{Exp}_{\mathit{Silk}} \to \mathit{RenEnv} \to \text{Exp}_{\mathit{Silk}}$

$\mathcal{R}_{\text{exp}}\llbracket I\rrbracket\ re\ =\ (re\ I)$

$\mathcal{R}_{\text{exp}}\llbracket\texttt{(set!}\ I\ E\texttt{)}\rrbracket\ re\ =\ \texttt{(set!}\ (re\ I)\ (\mathcal{R}_{\text{exp}}\llbracket E\rrbracket\ re)\texttt{)}$

$\mathcal{R}_{\text{exp}}\llbracket\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)}\rrbracket\ re$
$\quad = \texttt{(lambda (}I_1{}'\ \ldots\ I_n{}'\texttt{)}\ (\mathcal{R}_{\text{exp}}\llbracket E_{body}\rrbracket\ ([I_1 : I_1{}'] \ldots [I_n : I_n{}']re))\texttt{)}$,
where $I_1{}' \ldots I_n{}'$ are fresh.

$\mathcal{R}_{\text{exp}}\llbracket\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)}\rrbracket\ re$
$\quad = \texttt{(let ((}I_1{}'\ (\mathcal{R}_{\text{exp}}\llbracket E_1\rrbracket\ re)\texttt{)}\ \ldots\ \texttt{(}I_n{}'\ (\mathcal{R}_{\text{exp}}\llbracket E_n\rrbracket\ re)\texttt{))}$
$\qquad (\mathcal{R}_{\text{exp}}\llbracket E_{body}\rrbracket\ ([I_1 : I_1{}'] \ldots [I_n : I_n{}']re))\texttt{)}$,
where $I_1{}' \ldots I_n{}'$ are fresh.

$\mathcal{R}_{\text{exp}}\llbracket\texttt{(cycrec ((}I_1\ BV_1\texttt{)}\ \ldots\ \texttt{(}I_n\ BV_n\texttt{))}\ E_{body}\texttt{)}\rrbracket\ re$
$\quad = \texttt{(cycrec ((}I_1{}'\ (\mathcal{R}_{\text{exp}}\llbracket E_1\rrbracket\ re{}')\texttt{)}\ \ldots\ \texttt{(}I_n{}'\ (\mathcal{R}_{\text{exp}}\llbracket E_n\rrbracket\ re{}')\texttt{))}\ (\mathcal{R}_{\text{exp}}\llbracket E_{body}\rrbracket\ re{}')\texttt{)}$,
where $I_1{}' \ldots I_n{}'$ are fresh and $re{}'\ =\ ([I_1 : I_1{}'] \ldots [I_n : I_n{}']re)$

$\mathcal{R}_{\text{exp}}\llbracket E\rrbracket\ re\ =\ \mathit{mapsub}_{\mathit{Silk}}\llbracket E\rrbracket\ (\lambda E_{sub}\,.\,\mathcal{R}_{\text{exp}}\llbracket E_{sub}\rrbracket\ re)$, 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) ... (I_n E_n) of the `let` form are so-called **parallel bindings** in which the expressions E_1 ... E_n cannot refer to any of the internal variables I_1 ... I_n – i.e., the ones declared in the bindings. Any occurrences of I_1 ... I_n in E_1 ... 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 ... I_{i-1} within E_i refer to the internal variables, but references to I_i ... 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:

$\mathcal{R}_{\text{exp}} : \text{Exp}_{\mathit{Silk}} \rightarrow \mathit{RenEnv} \rightarrow \mathit{Nat} \rightarrow (\text{Exp}_{\mathit{Silk}} \times \mathit{Nat})$

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:

$\mathcal{R}_{\text{exp}} : \text{Exp}_{\mathit{Silk}} \rightarrow \mathit{RenEnv} \rightarrow \mathit{RenameCont} \rightarrow \mathit{Nat} \rightarrow \text{Exp}$

RenameCont is a renaming continuation defined as follows:

$\mathit{rc} \in \mathit{RenameCont} = \text{Exp} \rightarrow \mathit{Nat} \rightarrow \text{Exp}$

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^cps_sos = (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 `sqr`$_{cps}$ procedure invokes its continuation `ksqr` on the square of its input. The `sos`$_{cps}$ procedure first calls `sqr`$_{cps}$ on `a` with a continuation that names the result `asqr`. This continuation then calls `sqr`$_{cps}$ on `b` with a second continuation that names the second result `bsqr`. Finally, `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 ::= □	[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]

$$\begin{array}{rlrl}
P_{cps} \in & \text{Program}_{cps} & L \in & \text{Lit} \\
E_{cps} \in & \text{Exp}_{cps} & I \in & \text{Identifier} = \text{usual identifiers} \\
V_{cps} \in & \text{ValueExp}_{cps} & B \in & \text{Boollit} = \{\texttt{\#t}, \texttt{\#f}\} \\
AB_{cps} \in & \text{Abstraction}_{cps} & N \in & \text{Intlit} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\} \\
LE_{cps} \in & \text{LetableExp}_{cps} & O \in & \text{Primop} = \text{as in full } \textsc{Silk}. \\
BV_{cps} \in & \text{BindingValue}_{cps} & & \\
DV_{cps} \in & \text{DataValue}_{cps} & &
\end{array}$$

$$\begin{array}{rl}
P_{cps} ::= & \texttt{(silk (}I_{fml}{*}\texttt{)}\ E_{cps}\texttt{)} \\
E_{cps} ::= & \texttt{(call}\ V_{cps}\ V_{cps}{*}\texttt{)} \mid \texttt{(if}\ V_{cps}\ E_{cps}\ E_{cps}\texttt{)} \mid \texttt{(error}\ I\texttt{)} \\
& \mid \texttt{(let ((}I\ LE_{cps}\texttt{))}\ E_{cps}\texttt{)} \mid \texttt{(cycrec ((}I\ BV_{cps}\texttt{)}{*}\texttt{)}\ E_{cps}\texttt{)} \\
V_{cps} ::= & L \mid I \\
AB_{cps} ::= & \texttt{(lambda (}I{*}\texttt{)}\ E_{cps}\texttt{)} \\
LE_{cps} ::= & V_{cps} \mid AB_{cps} \mid \texttt{(primop}\ O_{op}\ V_{cps}{*}\texttt{)} \mid \texttt{(set!}\ I\ V\texttt{)} \\
BV_{cps} ::= & L \mid AB_{cps} \mid \texttt{(primop mprod}\ V_{cps}{*}\texttt{)} \\
DV_{cps} ::= & V \mid AB \\
L ::= & \texttt{\#u} \mid B \mid N
\end{array}$$

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 calls, ifs, cycrecs, or other lets. 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, calls and errors 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_{sos}^{cps} 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 $\ldots$ E_n`)`, explicit continuations are used to specify that the rator E_0 and rands $E_1 \ldots E_n$ are evaluated in left to right order before the invocation takes place. The fresh variables $I_0 \ldots I_n$ 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

```
$\mathcal{SCPS}_{prog}$ : Program$_{Silk}$ → Program$_{cps}$

$\mathcal{SCPS}_{prog}$[[(silk ($I_1$ ... $I_n$) $E_{body}$)]] =
  (silk ($I_1$ ... $I_n$ $I_{ktop}$) ; $I_{ktop}$ fresh
    (let (($I_{body}$ ($\mathcal{SCPS}_{exp}$[[$E_{body}$]]))) ; $I_{body}$ fresh
      (call $I_{body}$ $I_{ktop}$)))


$\mathcal{SCPS}_{exp}$ : Exp$_{Silk}$ → Exp$_{cps}$

$\mathcal{SCPS}_{exp}$[[$L$]] = (lambda ($I_k$) (call $I_k$ $L$)) ; $I_k$ fresh

$\mathcal{SCPS}_{exp}$[[$I$]] = (lambda ($I_k$) (call $I_k$ $I$)) ; $I_k$ fresh

$\mathcal{SCPS}_{exp}$[[(lambda ($I_1$ ... $I_n$) $E_{body}$)]] =
  (lambda ($I_k$) ; $I_k$ fresh
    (let (($I_{abs}$ ; $I_{abs}$ fresh
           (lambda ($I_1$ ... $I_n$ $I_{kcall}$) ; $I_{kcall}$ fresh
             (call ($\mathcal{SCPS}_{exp}$[[$E_{body}$]]) $I_{kcall}$))))
      (call $I_k$ $I_{abs}$)))

$\mathcal{SCPS}_{exp}$[[(call $E_0$ ... $E_n$)]] =
  (lambda ($I_k$) ; $I_k$ fresh
    (call ($\mathcal{SCPS}_{exp}$[[$E_0$]])
          (lambda ($I_0$) ; $I_0$ fresh
               ⋱
                (call ($\mathcal{SCPS}_{exp}$[[$E_n$]])
                      (lambda ($I_n$) ; $I_n$ fresh
                        (call $I_0$ ... $I_n$ $I_k$))) ...)))

$\mathcal{SCPS}_{exp}$[[(let (($I_1$ $E_1$) ... ($I_n$ $E_n$)) $E_{body}$)]] =
  (lambda ($I_k$) ; $I_k$ fresh
    (call ($\mathcal{SCPS}_{exp}$[[$E_1$]])
          (lambda ($I_1$)
               ⋱
                (call ($\mathcal{SCPS}_{exp}$[[$E_n$]])
                      (lambda ($I_n$)
                        (call ($\mathcal{SCPS}_{exp}$[[$E_{body}$]]) $I_k$))))))
```

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 ($\mathcal{SCPS}_{exp}[\![E_1]\!]$)
          (lambda ($I_1$) ; $I_1$ fresh
               $\ddots$
                (call ($\mathcal{SCPS}_{exp}[\![E_n]\!]$)
                      (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 (($I_1$ ($\mathcal{SCPS}_{bv}[\![BV_1]\!]$))
                  $\vdots$
             ($I_n$ ($\mathcal{SCPS}_{bv}[\![BV_n]\!]$)))
      (call ($\mathcal{SCPS}_{exp}[\![E_{body}]\!]$) $I_k$)))

$\mathcal{SCPS}_{exp}[\![$(set! $I_{lhs}$ $E_{rhs}$)$]\!]$ =
  (lambda ($I_k$) ; $I_k$ fresh
    (call ($\mathcal{SCPS}_{exp}[\![E_{rhs}]\!]$)
          (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 ($\mathcal{SCPS}_{exp}[\![E_{test}]\!]$)
          (lambda ($I_{test}$) ; $I_{test}$ fresh
            (if $I_{test}$
                (call ($\mathcal{SCPS}_{exp}[\![E_{then}]\!]$) $I_k$)
                (call ($\mathcal{SCPS}_{exp}[\![E_{else}]\!]$) $I_k$)))))

$\mathcal{SCPS}_{exp}[\![$(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 ($\mathcal{SCPS}_{exp}[\![E_{body}]\!]$) $I_{kcall}$))

$\mathcal{SCPS}_{dv}$ : $\text{DataValue}_{Silk} \rightarrow \text{DataValue}_{cps}$

$\mathcal{SCPS}_{dv}[\![V]\!] = V$

$\mathcal{SCPS}_{dv}[\![AB]\!] = \mathcal{SCPS}_{bv}[\![AB]\!]$
```

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 $\textsc{Silk}_{cps}$ by restricting `if` expressions to have the form

$$\texttt{(if } V_{cps} \texttt{ (call } V_{cps} \; V_{cps}{}^{*}\texttt{) } E_{cps}\texttt{)}.$$

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 k_6 be (λV_3 .
             let k_5 be (λV_4 .
                          (let ((ans.1 (@+ V_3 V_4)))
                            (call ktop.0 ans.1)))
              in (k_5 1))
   in (k_6 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$_{\mathit{cps}}$ value expressions to Silk$_{\mathit{cps}}$ expressions. The $\mathit{mc}{\rightarrow}\mathit{exp}$ and $\mathit{exp}{\rightarrow}\mathit{mc}$ functions perform conversions between compile-time meta-continuations and Silk$_{\mathit{cps}}$ 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$_{\mathit{cps}}$ abstraction of the form `(lambda (`I_k`) ...)` into a metalanguage abstraction of the form $(\lambda m.\ \ldots)$;
- transforming every Silk$_{\mathit{cps}}$ continuation of the form `(lambda (`I`) ...)` into a meta-continuation of the form $(\lambda V.\ \ldots)$;
- transforming every Silk$_{\mathit{cps}}$ 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 $\mathit{mc}{\rightarrow}\mathit{exp}$ and $\mathit{exp}{\rightarrow}\mathit{mc}$ 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$_{\mathit{cps}}$ expression context with one hole) is reduced to

$$\mathbb{E}\{[V_{actual}/V_{formal}]E\}$$

```
$E_{cps} \in \mathrm{Exp}_{cps}$
$V_{cps} \in \mathrm{ValueExp}_{cps}$
  $m \in \mathit{MetaCont} = \mathrm{ValueExp}_{cps} \rightarrow \mathrm{Exp}_{cps}$

$\mathrm{mc{\rightarrow}exp} : \mathit{MetaCont} \rightarrow \mathrm{Exp}_{cps} = (\lambda m\,.$ (lambda ($I_{temp}$) $(m\ I_{temp})$))
$\mathrm{exp{\rightarrow}mc} : \mathrm{Exp}_{cps} \rightarrow \mathit{MetaCont} = (\lambda E_{cps}\,.\ (\lambda V_{cps}\,.$ (call $E_{cps}\ V_{cps}$)))

$\mathcal{MCPS}_{prog} : \mathrm{Program}_{Silk} \rightarrow \mathrm{Program}_{cps}$

$\mathcal{MCPS}_{prog}[\![$(silk ($I_1$ ... $I_n$) $E_{body}$)$]\!]$ =
  (silk ($I_1$ ... $I_n$ $I_{ktop}$) ; $I_{ktop}$ fresh
    $(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ (\mathrm{exp{\rightarrow}mc}\ I_{ktop})))$

$\mathcal{MCPS}_{exp} : \mathrm{Exp}_{Silk} \rightarrow \mathit{MetaCont} \rightarrow \mathrm{Exp}_{cps}$

$\mathcal{MCPS}_{exp}[\![L]\!] = (\lambda m\,.\ (m\ I))$

$\mathcal{MCPS}_{exp}[\![I]\!] = (\lambda m\,.\ (m\ L))$

$\mathcal{MCPS}_{exp}[\![$(lambda ($I_1$ ... $I_n$) $E_{body}$)$]\!]$ =
  $(\lambda m\,.$ (let (($I_{abs}$ ; $I_{abs}$ fresh
                (lambda ($I_1$ ... $I_n$ $I_{kcall}$) ; $I_{kcall}$ fresh
                  $(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ (\mathrm{exp{\rightarrow}mc}\ I_{kcall}))$)))
          $(m\ I_{abs})$))

$\mathcal{MCPS}_{exp}[\![$(call $E_0$ ... $E_n$)$]\!]$ =
  $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_0]\!]$
          $(\lambda V_0\,.$
              $\ddots$
                $(\mathcal{MCPS}_{exp}[\![E_n]\!]$
                    $(\lambda V_n\,.$ (call $V_0$ ... $V_n$ $(\mathrm{mc{\rightarrow}exp}\ m)$))) ...)))

$\mathcal{MCPS}_{exp}[\![$(let (($I_1$ $E_1$) ... ($I_n$ $E_n$)) $E_{body}$)$]\!]$ =
  $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
          $(\lambda V_1\,.$
              $\ddots$
                $(\mathcal{MCPS}_{exp}[\![E_n]\!]$
                    $(\lambda V_n\,.$ (let (($I_1$ $V_1$) ... ($I_n$ $V_n$))
                            $(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ m)$))) ... )))
```

Figure 17.30: An efficient CPS transform based on meta-continuations, part 1.

```
$\mathcal{MCPS}_{exp}[\![$(primop $O$ $E_1$ ... $E_n$)$]\!]$ =
  ($\lambda m$ .  ($\mathcal{MCPS}_{exp}[\![E_1]\!]$
              ($\lambda V_1$ .
                  $\ddots$
                     ($\mathcal{MCPS}_{exp}[\![E_n]\!]$
                          ($\lambda V_n$ . (let (($I_{ans}$ (primop $O$ $V_1$ ... $V_n$))) ; $I_{ans}$ fresh
                                    ($m$ $I_{ans}$))) ... )))

$\mathcal{MCPS}_{exp}[\![$(cycrec (($I_1$ $BV_1$) ... ($I_n$ $BV_n$)) $E_{body}$)$]\!]$ =
  ($\lambda m$ . (cycrec (($I_1$ ($\mathcal{MCPS}_{bv}[\![BV_1]\!]$))
                               $\vdots$
                     ($I_n$ ($\mathcal{MCPS}_{bv}[\![BV_n]\!]$)))
           ($\mathcal{MCPS}_{exp}[\![E_{body}]\!]$ $m$))

$\mathcal{MCPS}_{exp}[\![$(set! $I_{lhs}$ $E_{rhs}$)$]\!]$ =
  ($\lambda m$ . ($\mathcal{MCPS}_{exp}[\![E_{rhs}]\!]$
              ($\lambda V_{rhs}$ .  (let (($I_{ans}$ (set! $I_{lhs}$ $V_{rhs}$))) ; $I_{ans}$ fresh
                          ($m$ $I_{ans}$)))))

$\mathcal{MCPS}_{exp}[\![$(if $E_{test}$ $E_{then}$ $E_{else}$)$]\!]$ =
  ($\lambda m$ . ($\mathcal{MCPS}_{exp}[\![E_{test}]\!]$
              ($\lambda V_{test}$ .  (let (($I_{kif}$ (mc→exp $m$)))
                           (if $V_{test}$
                                ($\mathcal{MCPS}_{exp}[\![E_{then}]\!]$ (exp→mc $I_{kif}$))
                                ($\mathcal{MCPS}_{exp}[\![E_{else}]\!]$ (exp→mc $I_{kif}$)))))))

$\mathcal{MCPS}_{exp}[\![$(error $I_{msg}$)$]\!]$ = ($\lambda m$ . (error $I_{msg}$))

$\mathcal{SCPS}_{bv}$ : BindingValue$_{Silk}$ → BindingValue$_{cps}$

$\mathcal{MCPS}_{bv}[\![L]\!]$ = $L$

$\mathcal{MCPS}_{bv}[\![$(@mprod $V_1$ ... $V_n$)$]\!]$ = (@mprod $V_1$ ... $V_n$) ; unchanged

$\mathcal{MCPS}_{bv}[\![$(lambda ($I_1$ ... $I_n$) $E_{body}$)$]\!]$ =
  (lambda ($I_1$ ... $I_n$ $I_{kcall}$) ; $I_{kcall}$ fresh
    ($\mathcal{MCPS}_{exp}[\![E]\!]$ (exp→mc $I_{kcall}$)))

$\mathcal{MCPS}_{dv}$ : DataValue$_{Silk}$ → 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 runtime 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

$$\begin{array}{l}(mc{\rightarrow}exp\ (exp{\rightarrow}mc\ \texttt{k}))\\ =((\lambda m\,.\,\texttt{(lambda (}I_{temp}\texttt{)}\ (m\ I_{temp})\texttt{)})\ (\lambda V\,.\,\texttt{(call k }V\texttt{)}))\\ =\texttt{(lambda (}I_{temp}\texttt{)}\ ((\lambda V\,.\,\texttt{(call k }V\texttt{)})\ I_{temp})\texttt{)}\\ =\texttt{(lambda (}I_{temp}\texttt{) (call k }I_{temp}\texttt{))}.\end{array}$$

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 $\mathcal{MCPS}_{prog}\llbracket\cdot\rrbracket$ To demonstrate this, show that the output of $\mathcal{MCPS}_{prog}\llbracket P_{mnun}\rrbracket$ 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 $\mathcal{MCPS}_{exp}\llbracket\rrbracket$ 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 $\mathcal{MCPS}_{prog}\llbracket\mathcal{AC}_{prog}\llbracket P_{set!}\rrbracket\rrbracket$.

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.

```
$m \in \mathit{MetaCont} = \mathrm{LetableExp}_{cps} \rightarrow \mathrm{Exp}_{cps}$

$mc{\rightarrow}exp : \mathit{MetaCont} \rightarrow \mathrm{Exp}_{cps} = (\lambda m\,.$ (lambda ($I_{temp}$) $(m\ I_{temp})$))
$exp{\rightarrow}mc : \mathrm{Exp}_{cps} \rightarrow \mathit{MetaCont}$
      $= (\lambda E_{cps}\,.\ (\lambda LE_{cps}\,.$ (let (($I_{rand}$ $LE_{cps}$)) (call $E_{cps}$ $I_{rand}$))))

$\mathcal{MCPS}_{exp}[\![$(lambda ($I_1$ ... $I_n$) $E_{body}$)$]\!] =$
   $(\lambda m\,.\ (m$ (lambda ($I_1$ ... $I_n$ $I_{kcall}$) ; $I_{kcall}$ fresh
               $(\mathcal{MCPS}_{exp}[\![E]\!]\ (exp{\rightarrow}mc\ I_{kcall}))$))))

$\mathcal{MCPS}_{exp}[\![$(call $E_0$ ... $E_n$)$]\!] =$
   $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
             $(\lambda LE_1\,.$
                 ⋱
                   $(\mathcal{MCPS}_{exp}[\![E_n]\!]$
                        $(\lambda LE_n\,.$ (let* (($I_1$ $LE_1$) ... ($I_n$ $LE_n$)) ; $I_1$ ... $I_n$ fresh
                                   (call $I_1$ ... $I_n$ $(mc{\rightarrow}exp\ m)$)))) ...)))

$\mathcal{MCPS}_{exp}[\![$(let (($I_1$ $E_1$) ... ($I_n$ $E_n$)) $E_{body}$)$]\!] =$
   $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
             $(\lambda LE_1\,.$
                 ⋱
                   $(\mathcal{MCPS}_{exp}[\![E_n]\!]$
                        $(\lambda LE_n\,.$ (let (($I_1$ $LE_1$) ... ($I_n$ $LE_n$))
                                   $(\mathcal{MCPS}_{exp}[\![E_{body}]\!]\ m)$))) ... )))

$\mathcal{MCPS}_{exp}[\![$(primop $O$ $E_1$ ... $E_n$)$]\!] =$
   $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_1]\!]$
             $(\lambda LE_1\,.$
                 ⋱
                   $(\mathcal{MCPS}_{exp}[\![E_n]\!]$
                        $(\lambda LE_n\,.$ (let* (($I_1$ $LE_1$) ... ($I_n$ $LE_n$)) ; $I_1$ ... $I_n$ fresh
                                   $(m$ (primop $O$ $I_1$ ... $I_n$)))) ... )))

$\mathcal{MCPS}_{exp}[\![$(set! $I_{lhs}$ $E_{rhs}$)$]\!] =$
   $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_{rhs}]\!]$
             $(\lambda LE_{rhs}\,.$ (let (($I_{rhs}$ $LE_{rhs}$)) ; $I_{rhs}$ fresh
                             $(m$ (set! $I_{lhs}$ $I_{rhs}$)))))

$\mathcal{MCPS}_{exp}[\![$(if $E_{test}$ $E_{then}$ $E_{else}$)$]\!] =$
   $(\lambda m\,.\ (\mathcal{MCPS}_{exp}[\![E_{test}]\!]$
             $(\lambda LE_{test}\,.$ (let (($I_{kif}$ $(mc{\rightarrow}exp\ m)$)
                                   ($I_{test}$ $LE_{test}$)) ; $I_{test}$ fresh
                              (if $I_{test}$
                                   $(\mathcal{MCPS}_{exp}[\![E_{then}]\!]\ (exp{\rightarrow}mc\ I_{kif}))$
                                   $(\mathcal{MCPS}_{exp}[\![E_{else}]\!]\ (exp{\rightarrow}mc\ I_{kif}))$)))))
```

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)))) -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:

```
CPS[[E]]  =  (program (define *top* (lambda (v) v))
                      (define *except* (lambda (v)
                                         "throw without catch"))
               (SCPS[[E]] *top* *except*))
SCPS[[I]]  =  (lambda (kn ke) (kn I))
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)) -eval-> 1

(begin (call (lambda (x)
               (call (lambda (y) y) x))
             #u)
       (call-count)) -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, `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 `fg`$_{code}$ has no free variables and so is a closed procedure. The environments `f`$_{env}$ and `g`$_{env}$ are tuples holding the free variable values. The closures `f`$_{clopair}$ and `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 f`$_{clopair}$`) (@mget 2 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 `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 $\{\texttt{c}, \texttt{r}, \texttt{t}\}$ 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.

```
$\mathcal{CL}[\![$(lambda ($I_1$ ... $I_n$) $E_{body}$)$]\!]$
  = let $\{I_{fv_1}, \ldots, I_{fv_k}\}$ be $FreeIds[\![$(lambda ($I_1$ ... $I_n$) $E_{body}$)$]\!]$
      in (@mprod (lambda ($I_{clo}$ $I_1$ ... $I_n$) ; $I_{clo}$ fresh
                   (let* (($I_{fv_1}$ (mget 2 $I_{clo}$))
                              ⋮
                          ($I_{fv_k}$ (mget k+1 $I_{clo}$)))
                     $\mathcal{CL}[\![E_{body}]\!]$))
                 $I_{fv_1}$ ... $I_{fv_k}$)

$\mathcal{CL}[\![$(call $E_{rator}$ $E_1$ ... $E_n$)$]\!]$
  = (let* (($I_{clo}$ $\mathcal{CL}[\![E_{rator}]\!]$) ; $I_{clo}$ fresh
           ($I_{code}$ (mget 1 $I_{clo}$))) ; $I_{code}$ fresh
      (call $I_{code}$ $I_{clo}$ $\mathcal{CL}[\![E_1]\!]$ ... $\mathcal{CL}[\![E_n]\!]$))
```

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 (($I_1$ $E_1$) ... ($I_n$ $E_n$)) ; $I_1$ ... $I_n$ are fresh
  (tagcase $E_{rator}$ $I_{rator}$
    (code (call $I_{rator}$ $I_1$ ... $I_n$))
    (closure (call (@mget 1 $I_{rator}$) $I_{rator}$ $I_1$ ... $I_n$))).
```

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

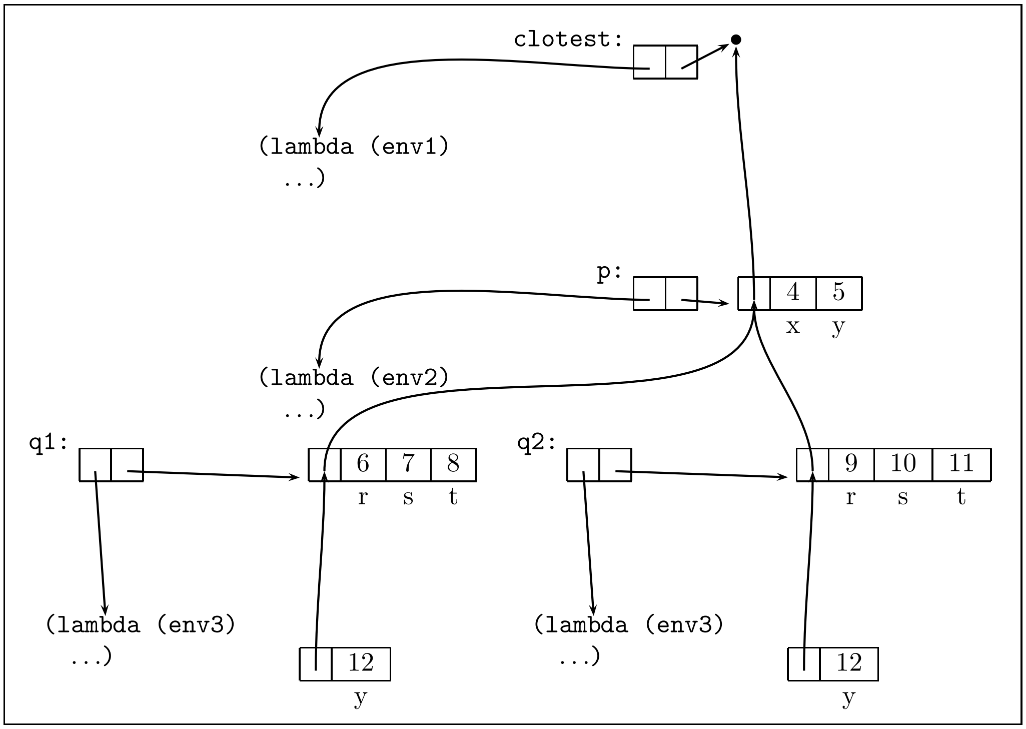

Figure 17.42: Figure depicting the links in the linked `clotest` example. (This figure needs lots of reformatting work!

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 *FreeIds*⟦`(lambda (`I_1 `...` I_n`)` E_{body}`)`⟧ 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 `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

$$\begin{array}{rlrl}
P_{lft} \in \text{Program}_{lft} & & L \in \text{Lit} & \\
E_{lft} \in \text{Exp}_{lft} & & I \in \text{Identifier}_{lft} & = \text{ usual identifiers} \\
BV_{lft} \in \text{BindingValue}_{lft} & & B \in \text{Boollit}_{lft} & = \{\texttt{\#t}, \texttt{\#f}\} \\
LE_{lft} \in \text{LetableExp}_{lft} & & N \in \text{Intlit}_{lft} & = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\} \\
V_{lft} \in \text{ValueExp}_{lft} & & O \in \text{Primop}_{lft} & = \text{ as in full } \textsc{Silk}.
\end{array}$$

$$\begin{array}{rl}
P_{lft} ::= & \texttt{(silk (}I_{fml}{}^{*}\texttt{) (cycrec ((}I\ \texttt{(lambda (}I^{*}\texttt{) }E_{lft}\texttt{))}^{*}\texttt{) }E_{lft}\texttt{))} \\
E_{lft} ::= & \texttt{(call }V_{lft}\ V_{lft}{}^{*}\texttt{)} \mid \texttt{(if }V_{lft}\ E_{lft}\ E_{lft}\texttt{)} \mid \texttt{(error }I\texttt{)} \\
 & \mid \texttt{(let ((}I\ LE_{lft}\texttt{)) }E_{lft}\texttt{)} \mid \texttt{(cycrec ((}I\ BV_{lft}\texttt{)}^{*}\texttt{) }E_{lft}\texttt{)} \\
V_{lft} ::= & L \mid I \\
LE_{lft} ::= & V_{lft} \mid \texttt{(primop }O_{op}\ V_{lft}{}^{*}\texttt{)} \\
BV_{lft} ::= & L \mid \texttt{(primop mprod }V_{lft}{}^{*}\texttt{)} \\
L ::= & \texttt{\#u} \mid B \mid H \mid N
\end{array}$$

Figure 17.43: Grammar for $\textsc{Silk}_{lft}$, 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 $\textsc{Silk}_{lft}$, a restricted form of $\textsc{Silk}_{cps}$ presented in Figure 17.43. The key difference between $\textsc{Silk}_{lft}$ and $\textsc{Silk}_{cps}$ is that $\textsc{Silk}_{lft}$ 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 $\textsc{Silk}_{lft}$ 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

   ```
   (cycrec (($I_{lam1}$  $AB_1{}'$)
                  ⋮
            ($I_{lamn}$  $AB_n{}'$))
     $E_{body}{}'$),
   ```

   where

   - $AB_1{}' \ldots AB_n{}'$ 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
   - $E_{body}{}'$ 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.