Programming Languages and Lambda Calculi

(Summer 2006 Revised Version)

Matthias Felleisen Matthew Flatt

Draft: July 12, 2006

Contents

Preface

These notes are a work in progress, but you may find them useful.

Part I

Models of Languages

Chapter 1: Computing with Text

In this book, we study how a programming language can be defined in a way that is easily understood by people, and also amenable to formal analysis—where the formal analysis should be easily understood by people, too.

One way of defining a language is to write prose paragraphs that explain the kinds of expressions that are allowed in the language and how they are evaluated. This technique has an advantage that a reader can quickly absorb general ideas about the language, but details about the language are typically difficult to extract from prose. Worse, prose does not lend itself to formal analysis.

Another way of defining a language is to implement an interpreter for it in some meta-language. Assuming that a reader is familiar with the meta-language, this technique has the advantage of clearly and completely specifying the details of the language. Assuming that the meta-language has a formal specification, then the interpreter has a formal meaning in that language, and it can be analyzed.

The meta-language used to define another language need not execute efficiently, since its primary purpose is to explain the other language to humans. The meta-language's primitive data constructs need not be defined in terms of bits and bytes. Indeed, for the meta-language we can directly use logic and set theory over a universe of program text. In the same way that computation on physical machines can be described, ultimately, in terms of shuffling bits among memory locations, computation in the abstract can be described in terms of relations on text.

1.1 Defining Sets

When we write a BNF grammar, such as

$$\begin{array}{rcl} B & = & \texttt{t} \\ & | & \texttt{f} \\ & | & (B \bullet B) \end{array}$$

then we actually define a set B of textual strings. The above definition can be expanded to the following constraints on B:

$$\begin{array}{c} \texttt{t} \in B \\ \texttt{f} \in B \\ a \in B \text{ and } b \in B \;\Rightarrow\; (a \bullet b) \in B \end{array}$$

Technically, the set B that we mean is the smallest set that obeys the above constraints.

Notation: we'll sometimes use "B" to mean "the set B", but sometimes "B" will mean "an arbitrary element of B". The meaning is always clear from the context. Sometimes, we use a subscript or a prime on the name of the set to indicate an arbitrary element of the set, such as "B_1" or "B'". Thus, the above constraints might be written as

$$\begin{array}{cl} \texttt{t} \in B & \text{[a]} \\ \texttt{f} \in B & \text{[b]} \\ (B_1 \bullet B_2) \in B & \text{[c]} \end{array}$$

Whether expressed in BNF shorthand or expanded as a set of constraints, set B is defined recursively. Enumerating all of the elements of B in finite space is clearly impossible in this case:

$$B = \{\texttt{t},\ \texttt{f},\ (\texttt{t} \bullet \texttt{t}),\ (\texttt{t} \bullet \texttt{f}),\ \ldots\}$$

Given a particular bit of text that belongs to B, however, we can demonstrate that it is in B by showing how the constraints that define B require the text to be in B. For example, $(\texttt{t} \bullet (\texttt{f} \bullet \texttt{t}))$ is in B:

1.	$\texttt{t} \in B$	by [a]
2.	$\texttt{f} \in B$	by [b]
3.	$\texttt{t} \in B$	by [a]
4.	$(\texttt{f} \bullet \texttt{t}) \in B$	by 2, 3, and [c]
5.	$(\texttt{t} \bullet (\texttt{f} \bullet \texttt{t})) \in B$	by 1, 4, and [c]

We'll typically write such arguments in proof-tree form:

$$\dfrac{\texttt{t} \in B \text{ [a]} \qquad \dfrac{\texttt{f} \in B \text{ [b]} \qquad \texttt{t} \in B \text{ [a]}}{(\texttt{f} \bullet \texttt{t}) \in B} \text{ [c]}}{(\texttt{t} \bullet (\texttt{f} \bullet \texttt{t})) \in B} \text{ [c]}$$

Also, we'll tend to omit the names of deployed rules from the tree, since they're usually obvious:

$$\dfrac{\texttt{t} \in B \qquad \dfrac{\texttt{f} \in B \qquad \texttt{t} \in B}{(\texttt{f} \bullet \texttt{t}) \in B}}{(\texttt{t} \bullet (\texttt{f} \bullet \texttt{t})) \in B}$$

▷ **Exercise 1.1.** Which of the following are in B? For each member of B, provide a proof tree showing that it must be in B.

1. $\texttt{t}$
2. $\bullet$
3. $((\texttt{f} \bullet \texttt{t}) \bullet (\texttt{f} \bullet \texttt{f}))$
4. $((\texttt{f}) \bullet (\texttt{t}))$

1.2 Relations

A **relation** is a set whose elements consist of ordered pairs.[1] For example, we can define the $\equiv$ relation to match each element of B with itself:

$$a \in B \quad \Rightarrow \quad \langle a, a \rangle \in \equiv$$

For binary relations such as $\equiv$, instead of $\langle a, a \rangle \in \equiv$, we usually write $a \equiv a$:

$$a \in B \quad \Rightarrow \quad a \equiv a$$

or even simply

$$B_1 \equiv B_1$$

as long as it is understood as a definition of $\equiv$. As it turns out, the relation $\equiv$ is reflexive, symmetric, and transitive:

a relation $\mathbf{R}$ is **reflexive**	iff	$a \,\mathbf{R}\, a$ (for any a)
a relation $\mathbf{R}$ is **symmetric**	iff	$a \,\mathbf{R}\, b \Rightarrow b \,\mathbf{R}\, a$
a relation $\mathbf{R}$ is **transitive**	iff	$a \,\mathbf{R}\, b$ and $b \,\mathbf{R}\, c \Rightarrow a \,\mathbf{R}\, c$

[1] An ordered pair $\langle x, y \rangle$ can be represented as a set of sets $\{\{x\}, \{x, y\}\}$.

If a relation is reflexive, symmetric, and transitive, then it is an **equivalence**.

The following defines a relation $\mathbf{r}$ that is neither reflexive, symmetric, nor transitive:

$$\begin{array}{rlll} (\texttt{f} \bullet B_1) & \mathbf{r} & B_1 & \text{[a]} \\ (\texttt{t} \bullet B_1) & \mathbf{r} & \texttt{t} & \text{[b]} \end{array}$$

Using $\mathbf{r}$, we could define a new relation $\asymp_{\mathbf{r}}$ by adding the constraint that $\asymp_{\mathbf{r}}$ is reflexive:

$$\begin{array}{rlll} (\texttt{f} \bullet B_1) & \asymp_{\mathbf{r}} & B_1 & \text{[a]} \\ (\texttt{t} \bullet B_1) & \asymp_{\mathbf{r}} & \texttt{t} & \text{[b]} \\ B_1 & \asymp_{\mathbf{r}} & B_1 & \text{[c]} \end{array}$$

The relation $\asymp_{\mathbf{r}}$ is the **reflexive closure** of the $\mathbf{r}$ relation. We could define yet another relation by adding symmetric and transitive constraints:

$$\begin{array}{rlll} (\texttt{f} \bullet B_1) & \approx_{\mathbf{r}} & B_1 & \text{[a]} \\ (\texttt{t} \bullet B_1) & \approx_{\mathbf{r}} & \texttt{t} & \text{[b]} \\ B_1 & \approx_{\mathbf{r}} & B_1 & \text{[c]} \\ B_1 \approx_{\mathbf{r}} B_2 & \Rightarrow & B_2 \approx_{\mathbf{r}} B_1 & \text{[d]} \\ B_1 \approx_{\mathbf{r}} B_2 \text{ and } B_2 \approx_{\mathbf{r}} B_3 & \Rightarrow & B_1 \approx_{\mathbf{r}} B_3 & \text{[e]} \end{array}$$

The $\approx_{\mathbf{r}}$ relation is the **symmetric–transitive closure** of $\asymp_{\mathbf{r}}$, and it is the **reflexive–symmetric–transitive closure** or **equivalence closure** of $\mathbf{r}$.

1.3 Relations as Evaluation

The running example of B and $\mathbf{r}$ should give you an idea of how a programming language can be defined through text and relations on text—or, more specifically, through a set (B) and a relation on the set ($\mathbf{r}$).

In fact, you might begin to suspect that B is a grammar for boolean expressions with $\bullet$ as "or", and $\approx_{\mathbf{r}}$ equates pairs of expressions that have the same boolean value.

Indeed, using the constraints above, we can show that $(\texttt{f} \bullet \texttt{t}) \approx_{\mathbf{r}} (\texttt{t} \bullet \texttt{t})$, just as $\mathsf{false} \vee \mathsf{true} = \mathsf{true} \vee \mathsf{true}$:

$$\dfrac{\dfrac{(\texttt{f} \bullet \texttt{t}) \approx_{\mathbf{r}} \texttt{t}\ \text{[a]} \qquad \dfrac{(\texttt{t} \bullet \texttt{t}) \approx_{\mathbf{r}} \texttt{t}\ \text{[b]}}{\texttt{t} \approx_{\mathbf{r}} (\texttt{t} \bullet \texttt{t})}\ \text{[d]}}{(\texttt{f} \bullet \texttt{t}) \approx_{\mathbf{r}} (\texttt{t} \bullet \texttt{t})}\ \text{[e]}$$

However, it does not follow directly that $\bullet$ is exactly like a boolean "or". Instead, we would have to prove general claims about $\bullet$, such as the fact that $(B_1 \bullet \texttt{t}) \approx_{\mathbf{r}} \texttt{t}$ for any expression B_1. (It turns out that we would not be able to prove the claim, as we'll soon see.)

In other words, there is generally a gap between an interpreter-defined language (even if the interpreter is defined with mathematics) and properties of the language that we might like to guarantee. For various purposes, the properties of a language are as important as the values it computes. For example, if $\bullet$ really acted like "or", then a compiler might safely optimize $(B_1 \bullet \texttt{t})$ as $\texttt{t}$. Similarly, if syntactic rules for a language gurantee that a number can never be added to any value other than another number, then the implementation of the language need not check the arguments of an addition expression to ensure that they are numbers.

Before we can start to address such gaps, however, we must eliminate a meta-gap between the way that we define languages and the way that normal evaluation proceeds.

1.4 Directed Evaluation

The "evaluation" rule $\approx_{\mathbf{r}}$ is not especially evaluation-like. It allows us to prove that certain expressions are equivalent, but it doesn't tell us exactly how to get from an arbitrary B to either $\texttt{t}$ or $\texttt{f}$.

The simpler relation $\mathbf{r}$ was actually more useful in this sense. Both cases in the definition of $\mathbf{r}$ map an expression to a smaller expression. Also, for any expression B, either B is $\mathtt{t}$ or $\mathtt{f}$, or $\mathbf{r}$ relates B to at most one other expression. As a result, we can think of $\mathbf{r}$ as a **single-step reduction**, corresponding to the way that an interpreter might take a single evaluation step in working towards a final value.

We can define $\rightsquigarrow_{\mathbf{r}}$ as the reflexive–transitive closure of $\mathbf{r}$, and we end up with a **multi-step reduction** relation. The multi-step relation $\rightsquigarrow_{\mathbf{r}}$ will map each expression to many other expressions. As it turns out, though, $\rightsquigarrow_{\mathbf{r}}$ maps each expression to at most one of $\mathtt{t}$ or $\mathtt{f}$.

> It's sufficient to define $\rightsquigarrow_{\mathbf{r}}$ as "the reflexive–transitive closure of $\mathbf{r}$". An alternate formulation would be to expand out the constraints, as in the previous section. A third way would be to partially expand the constraints, but use $\mathbf{r}$ in the definition of $\rightsquigarrow_{\mathbf{r}}$:
>
> $$\begin{array}{rcl} B_1 & \rightsquigarrow_{\mathbf{r}} & B_1 \\ B_1 \,\mathbf{r}\, B_2 & \Rightarrow & B_1 \rightsquigarrow_{\mathbf{r}} B_2 \\ B_1 \rightsquigarrow_{\mathbf{r}} B_2 \text{ and } B_2 \rightsquigarrow_{\mathbf{r}} B_3 & \Rightarrow & B_1 \rightsquigarrow_{\mathbf{r}} B_3 \end{array}$$

The relations $\mathbf{r}$ and $\rightsquigarrow_{\mathbf{r}}$ are intentionally asymmetric, corresponding to the fact that evaluation should proceed in a specific direction towards a value. For example, given the expression $(\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f})))$, we can show a **reduction** to $\mathtt{t}$:

$$\begin{array}{lll} (\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f}))) & \mathbf{r} & (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f})) \\ & \mathbf{r} & (\mathtt{t} \bullet \mathtt{f}) \\ & \mathbf{r} & \mathtt{t} \end{array}$$

Each blank line in the left column is implicitly filled by the expression in the right column from the previous line. Each line is then a step in an argument that $(\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f}))) \rightsquigarrow_{\mathbf{r}} \mathtt{t}$.

▷ **Exercise 1.2.** Show that $(\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{f} \bullet \mathtt{f}))) \rightsquigarrow_{\mathbf{r}} \mathtt{f}$ by showing its reduction with the $\mathbf{r}$ one-step relation.

1.5 Evaluation in Context

How does the expression $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ reduce? According to $\mathbf{r}$ or $\rightsquigarrow_{\mathbf{r}}$, it does not reduce at all!

Intuitively, $((\mathtt{f}\bullet\mathtt{t})\bullet\mathtt{f})$ should reduce to $(\mathtt{t}\bullet\mathtt{f})$, by simplifying the first subexpression according to $(\mathtt{f} \bullet \mathtt{t}) \,\mathbf{r}\, \mathtt{f}$. But there is no constraint in the definition of $\mathbf{r}$ that matches $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ as the source expression. We can only reduce expressions of the form $(\mathtt{f} \bullet B)$ and $(\mathtt{t} \bullet B)$. In other words, the expression on the right-hand side of the outermost $\bullet$ is arbitrary, but the expression on the left-hand side must be $\mathtt{f}$ or $\mathtt{t}$.

We can extend the $\mathbf{r}$ relation with $\rightarrow_{\mathbf{r}}$ to support the reduction of subexpressions:

$$\begin{array}{rcll} B_1 \,\mathbf{r}\, B_2 & \Rightarrow & B_1 \rightarrow_{\mathbf{r}} B_2 & \text{[a]} \\ B_1 \rightarrow_{\mathbf{r}} B_1' & \Rightarrow & (B_1 \bullet B_2) \rightarrow_{\mathbf{r}} (B_1' \bullet B_2) & \text{[b]} \\ B_2 \rightarrow_{\mathbf{r}} B_2' & \Rightarrow & (B_1 \bullet B_2) \rightarrow_{\mathbf{r}} (B_1 \bullet B_2') & \text{[c]} \end{array}$$

The $\rightarrow_{\mathbf{r}}$ relation is the **compatible closure** of the $\mathbf{r}$ relation. Like $\mathbf{r}$, $\rightarrow_{\mathbf{r}}$ is a single-step reduction relation, but $\rightarrow_{\mathbf{r}}$ allows the reduction of any subexpression within the whole expression. The subexpression reduced with $\mathbf{r}$ is called the **redex**, and the text surrounding a redex is its **context**.

In particular, the $\rightarrow_{\mathbf{r}}$ relation includes $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{f})$. We can demonstrate this inclusion with the following proof tree:

$$\dfrac{\dfrac{(\mathtt{f} \bullet \mathtt{t}) \,\mathbf{r}\, \mathtt{t}}{(\mathtt{f} \bullet \mathtt{t}) \rightarrow_{\mathbf{r}} \mathtt{t}}\ \text{[a]}}{((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{f})}\ \text{[b]}$$

Continuing with $\rightarrow_{\mathbf{r}}$, we can reduce $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ to $\mathtt{t}$:

$$\begin{array}{rll} ((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) & \rightarrow_{\mathbf{r}} & (\mathtt{t} \bullet \mathtt{f}) \\ & \rightarrow_{\mathbf{r}} & \mathtt{t} \end{array}$$

Finally, if we define $\rightarrow\!\!\!\!\rightarrow_{\mathbf{r}}$ to be the reflexive–transitive closure of $\rightarrow_{\mathbf{r}}$, then we get $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow\!\!\!\!\rightarrow_{\mathbf{r}} \mathtt{t}$. Thus, $\rightarrow\!\!\!\!\rightarrow_{\mathbf{r}}$ is the natural **reduction relation** generated by $\mathbf{r}$.

In general, the mere reflexive closure $\asymp_{\mathbf{r}}$, equivalence closure $\approx_{\mathbf{r}}$, or reflexive-transitive closure $\rightsquigarrow\!\!\!\!\!\rightarrow_{\mathbf{r}}$ of a relation $\mathbf{r}$ will be uninteresting. Instead, we'll most often be interested in the compatible closure $\rightarrow_{\mathbf{r}}$ and its reflexive–transitive closure $\rightarrow\!\!\!\!\rightarrow_{\mathbf{r}}$, because they correspond to typical notions of evaluation. In addition, the equivalence closure $=_{\mathbf{r}}$ of $\rightarrow_{\mathbf{r}}$ is interesting because it relates expressions that produce the same result.

▷ **Exercise 1.3.** Explain why $(\mathtt{f} \bullet ((\mathtt{t} \bullet \mathtt{f}) \bullet \mathtt{f})) \not\rightsquigarrow\!\!\!\!\!\rightarrow_{\mathbf{r}} \mathtt{t}$.

▷ **Exercise 1.4.** Show that $(\mathtt{f} \bullet ((\mathtt{t} \bullet \mathtt{f}) \bullet \mathtt{f})) \rightarrow\!\!\!\!\rightarrow_{\mathbf{r}} \mathtt{t}$ by demonstrating a reduction with $\rightarrow_{\mathbf{r}}$.

1.6 Evaluation Function

The $\rightarrow\!\!\!\!\rightarrow_{\mathbf{r}}$ brings us close to a useful notion of evaluation, but we are not there yet. While $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow\!\!\!\!\rightarrow_{\mathbf{r}} \mathtt{t}$, it's also the case that $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow\!\!\!\!\rightarrow_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{f})$ and $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow\!\!\!\!\rightarrow_{\mathbf{r}} ((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$.

For evaluation, we're interested in whether B evaluates to $\mathtt{f}$ or to $\mathtt{t}$; any other mapping by $\rightarrow\!\!\!\!\rightarrow_{\mathbf{r}}$ or $=_{\mathbf{r}}$ is irrelevant. To capture this notion of evaluation, we define the $\mathit{eval}_{\mathbf{r}}$ relation as follows:

$$\mathit{eval}_{\mathbf{r}}(B) = \begin{cases} \mathtt{f} & \text{if } B =_{\mathbf{r}} \mathtt{f} \\ \mathtt{t} & \text{if } B =_{\mathbf{r}} \mathtt{t} \end{cases}$$

Here, we're using yet another notation to define a relation. This particular notation is suggestive of a **function**, i.e., a relation that maps each element to at most one element. We use the function notation because $\mathit{eval}_{\mathbf{r}}$ must be a function if it's going to make sense as an evaluator.

▷ **Exercise 1.5.** Among the relations $\mathbf{r}$, $\asymp_{\mathbf{r}}$, $\approx_{\mathbf{r}}$, $\rightsquigarrow\!\!\!\!\!\rightarrow_{\mathbf{r}}$, $\rightarrow_{\mathbf{r}}$, $\rightarrow\!\!\!\!\rightarrow_{\mathbf{r}}$, $=_{\mathbf{r}}$, and $\mathit{eval}_{\mathbf{r}}$, which are functions? For each non-function relation, find an expression and two expressions that it relates to.

1.7 Notation Summary

name	definition	intuition
$\_$	the base relation on members of an expression grammar	a single "reduction" step with no context
$\rightarrow_{\_}$	the compatible closure of $\_$ with respect to the expression grammar	a single step within a context
$\rightarrow\!\!\!\!\rightarrow_{\_}$	the reflexive–transitive closure of $\rightarrow_{\_}$	multiple evaluation steps (zero or more)
$=_{\_}$	the symmetric–transitive closure of $\rightarrow\!\!\!\!\rightarrow_{\_}$	equates expressions that produce the same result
$\mathit{eval}_{\_}$	$=_{\_}$ restricted to a range of results	complete evaluation

Chapter 2: Consistency of Evaluation

Now that we have structural induction, we're ready to return to the issue of $eval_{\mathbf{r}}$'s **consistency** as an evaluator. In other words, we'd like to prove that $eval_{\mathbf{r}}$ is a function. More formally, given a notion of results R:

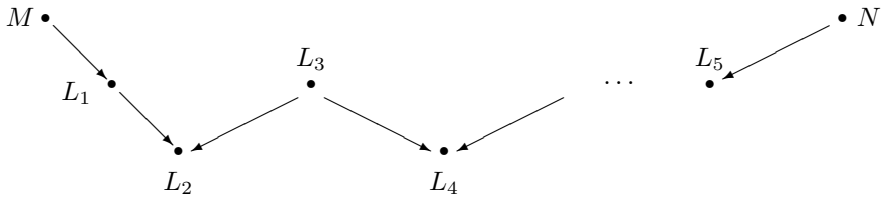
$$\begin{array}{rcl} R & = & \mathtt{t} \\ & | & \mathtt{f} \end{array}$$

we'd like to prove the following theorem:

> **Theorem 2.1**: If $eval_{\mathbf{r}}(B_0) = R_1$ and $eval_{\mathbf{r}}(B_0) = R_2$, then $R_1 = R_2$.

To prove the theorem, we can assume $eval_{\mathbf{r}}(B_0) = R_1$ and $eval_{\mathbf{r}}(B_0) = R_2$ for some B_0, R_1, and R_2, and we'd like to conclude that $R_1 = R_2$. By the definition of $eval_{\mathbf{r}}$, our assumption implies that $B_0 =_{\mathbf{r}} R_1$ and $B_0 =_{\mathbf{r}} R_2$ (using $=_{\mathbf{r}}$, not $=$). Hence, by the definition of $=_{\mathbf{r}}$ as an equivalence relation, $R_1 =_{\mathbf{r}} R_2$. To reach the conclusion that $R_1 = R_2$, we must study the nature of calculations, which is the general shape of proofs $M =_{\mathbf{r}} N$ when $M, N \in B$.

Since $=_{\mathbf{r}}$ is an extension of the one-step reduction $\rightarrow_{\mathbf{r}}$, a calculation to prove $M =_{\mathbf{r}} N$ is generally a series of one-step reductions based on $\mathbf{r}$ in both directions:



where each $L_n \in B$. Possibly, these steps can be rearranged such that all reduction steps go from M to some L and from N to the same L. In other words, if $M =_{\mathbf{r}} N$ perhaps there is an expression L such that $M \twoheadrightarrow_{\mathbf{r}} L$ and $N \twoheadrightarrow_{\mathbf{r}} L$.

If we can prove that such an L always exists, the proof of consistency is finished. Recall that we have

$$R_1 =_{\mathbf{r}} R_2$$

By the (as yet unproved) claim, there must be an expression L such that

$$R_1 \twoheadrightarrow_{\mathbf{r}} L \qquad \text{and} \qquad R_2 \twoheadrightarrow_{\mathbf{r}} L$$

But elements of R, which are just `t` and `f`, are clearly not reducible to anything except themselves. So $L = R_1$ and $L = R_2$, which means that $R_1 = R_2$.

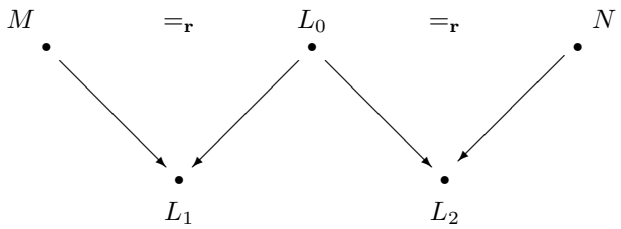
By the preceding argument we have reduced the proof of $eval_{\mathbf{r}}$'s consistency to a claim about the shape of arguments that prove $M =_{\mathbf{r}} N$. This crucial insight about the connection between a consistency proof for a formal equational system and the rearrangement of a series of reduction steps is due to Church and Rosser, who used this idea to analyze the consistency of a language called the λ-calculus (which we'll study soon). In general, an equality relation on terms generated from some basic notion of reduction is **Church-Rosser** if it satisfies the rearrangement property.

> **Theorem 2.2 [Church-Rosser for $=_{\mathbf{r}}$]**: If $M =_{\mathbf{r}} N$, then there exists an expression L such that $M \twoheadrightarrow_{\mathbf{r}} L$ and $N \twoheadrightarrow_{\mathbf{r}} L$.

Since we're given a particular $M =_{\mathbf{r}} N$, and since the definition of $=_{\mathbf{r}}$ is recursive, we can prove the theorem by induction of the structure of the derivation of $M =_{\mathbf{r}} N$.

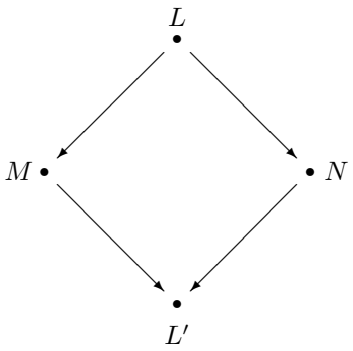
Proof for Theorem 2.2: By induction on the structure of the proof of $M =_{\mathbf{r}} N$.

- Base case:
  - **Case** $M \twoheadrightarrow_{\mathbf{r}} N$
    Let $L = N$, and the claim holds.
- Inductive cases:
  - **Case** $M =_{\mathbf{r}} N$ **because** $N =_{\mathbf{r}} M$
    By induction, an L exists for $N =_{\mathbf{r}} M$, and that is the L we want.
  - **Case** $M =_{\mathbf{r}} N$ **because** $M =_{\mathbf{r}} L_0$ **and** $L_0 =_{\mathbf{r}} N$
    By induction, there exists an L_1 such that $M \twoheadrightarrow_{\mathbf{r}} L_1$ and $L_0 \twoheadrightarrow_{\mathbf{r}} L_1$. Also by induction, there exists an L_2 such that $N \twoheadrightarrow_{\mathbf{r}} L_2$ and $L_0 \twoheadrightarrow_{\mathbf{r}} L_2$. In pictures we have:



    Now suppose that, whenever an L_0 reduces to both L_1 and L_2, there exists an L_3 such that $L_1 \twoheadrightarrow_{\mathbf{r}} L_3$ and $L_2 \twoheadrightarrow_{\mathbf{r}} L_3$. Then the claim we want to prove holds, because $M \twoheadrightarrow_{\mathbf{r}} L_3$ and $N \twoheadrightarrow_{\mathbf{r}} L_3$.

Again, we have finished the proof modulo the proof of yet another claim about the reduction system. The new property is called **diamond property** because a picture of the theorem demands that reductions can be arranged in the shape of a diamond:



Theorem 2.3 [Diamond Property for $\twoheadrightarrow_{\mathbf{r}}$]: If $L \twoheadrightarrow_{\mathbf{r}} M$ and $L \twoheadrightarrow_{\mathbf{r}} N$, then there exists an expression L' such that $M \twoheadrightarrow_{\mathbf{r}} L'$ and $N \twoheadrightarrow_{\mathbf{r}} L'$.

To prove this theorem, it will be helpful to first prove a diamond-like property for $\rightarrow_{\mathbf{r}}$:

Lemma 2.4 [Diamond-like Property for $\rightarrow_{\mathbf{r}}$]: If $L \rightarrow_{\mathbf{r}} M$ and $L \rightarrow_{\mathbf{r}} N$, then either

- $M = N$,
- $M \rightarrow_{\mathbf{r}} N$,

- $N \to_{\mathbf{r}} M$, or
- there exists an L' such that $M \to_{\mathbf{r}} L'$ and $N \to_{\mathbf{r}} L'$.

To prove this lemma, we consider the given $L \to_{\mathbf{r}} M$ and note that $\to_{\mathbf{r}}$ is recursively defined:

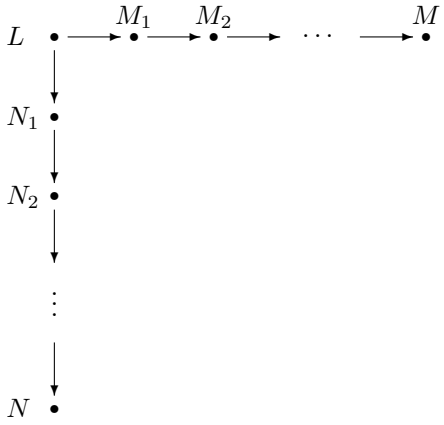
Proof for Lemma 2.4: By induction on the structure of the proof of $L \to_{\mathbf{r}} M$.

- Base case:
  - **Case** $L\,\mathbf{r}\,M$
    By the definition of $\mathbf{r}$, there are two subcases.
    - **Case** $L = (\mathtt{f} \bullet B_0)$ **and** $M = B_0$
      The expression L might be reduced in two ways with $\to_{\mathbf{r}}$ for N: to B_0, or to $(\mathtt{f} \bullet B_0')$ with $B_0 \to_{\mathbf{r}} B_0'$.
      If $N = B_0$, then $M = N$ and the clain holds.
      Otherwise, $N = (\mathtt{f} \bullet B_0')$. Then, since $M = B_0$, $M \to_{\mathbf{r}} B_0'$, and by the definition of $\mathbf{r}$, $N \to_{\mathbf{r}} B_0'$. Thus, the claim holds with $L' = B_0'$.
    - **Case** $L = (\mathtt{t} \bullet B_0)$ **and** $M = \mathtt{t}$
      Similar to the previous case; either $N = M$ or $N = (\mathtt{t} \bullet B_0')$ so that $N \to_{\mathbf{r}} M$.
- Inductive cases; without loss of generality, assume $L \not\mathbf{r} N$ (otherwise swap N and M).
  - **Case** $L = (B_1 \bullet B_2)$**,** $B_1 \to_{\mathbf{r}} B_1'$**, and** $M = (B_1' \bullet B_2)$
    We have two sub-cases:
    - $N = (B_1'' \bullet B_2)$, where $B_1 \to_{\mathbf{r}} B_1''$. Since $B_1 \to_{\mathbf{r}} B_1'$, and $B_1 \to_{\mathbf{r}} B_1''$, we can apply induction. If $B_1' = B_1''$, then $M = N$, and the claim holds. If $B_1'' \to_{\mathbf{r}} B_1'$, then $M \to_{\mathbf{r}} N$ and the claim holds; similarly, if $B_1' \to_{\mathbf{r}} B_1''$, then $N \to_{\mathbf{r}} M$ and the claim holds. Finally, if $B_1' \to_{\mathbf{r}} B_1'''$ and $B_1'' \to_{\mathbf{r}} B_1'''$, then $M \to_{\mathbf{r}} (B_1''' \bullet B_2)$ and $N \to_{\mathbf{r}} (B_1''' \bullet B_2)$, and the claim holds with $L' = (B_1''' \bullet B_2)$.
    - $N = (B_1 \bullet B_2')$ with $B_2 \to_{\mathbf{r}} B_2'$. Since $B_1 \to_{\mathbf{r}} B_1'$, $N \to_{\mathbf{r}} (B_1' \bullet B_2')$. Similarly, $M \to_{\mathbf{r}} (B_1' \bullet B_2')$. Thus, the claim holds with $L' = (B_1' \bullet B_2')$.
  - **Case** $L = (B_1 \bullet B_2)$**,** $B_2 \to_{\mathbf{r}} B_2'$**, and** $M = (B_1 \bullet B_2')$
    Analogous to the previous case.

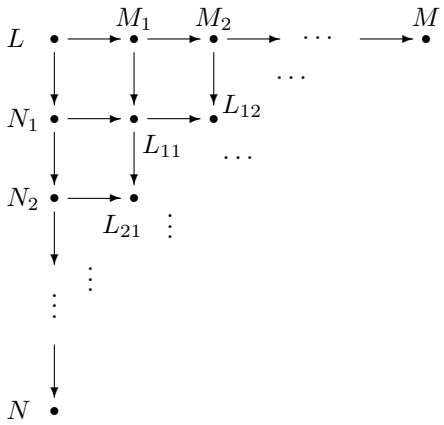
Now that we know that the one-step reduction satisfies a diamond-like property, we can show that its transitive–reflexive closure satifies the diamond property. Assume that $L \twoheadrightarrow_{\mathbf{r}} M$ and $L \twoheadrightarrow_{\mathbf{r}} N$. By the inductive definition of the reduction relation $\twoheadrightarrow_{\mathbf{r}}$,

$$L \to_{\mathbf{r}}^{m} M \qquad \text{and} \qquad L \to_{\mathbf{r}}^{n} N$$

for some $m, n \in \mathbb{N}$, where $\to_{\mathbf{r}}^{m}$ means m steps with $\to_{\mathbf{r}}$. Pictorially, we have



Using the diamond-like property for the one-step reduction, we can now fill in expressions L_{11}, L_{21}, L_{12}, etc. until the entire square is filled out:



Formally, this idea can also be cast as an induction.

The preceding arguments also show that $M =_{\mathbf{r}} R$ if and only if $M \twoheadrightarrow_{\mathbf{r}} R$. Consequently, we could have defined the evaluation via reduction without loss of generality. Put differently, symmetric reasoning steps do not help in the evaluation of B expressions. In in the next chapter, however, we will introduce a programming language for which such apparent backward steps can truly shorten the calculation of the result of a program.

After determining that a program has a unique value, the question arises whether a program *always* has a value.

Theorem 2.5: For any B_0, $eval_{\mathbf{r}}(B_0) = R_0$ for some R_0.

It follows that an algorithm exists for deciding the equality of B expressions: evaluate both expressions and compare the results. Realistic programming languages include arbitrary non-terminating expressions and thus preclude a programmer from deciding the equivalence of expressions in this way.

▷ **Exercise 2.1.** Prove Theorem 2.3 (formally, instead of using a diagram).

▷ **Exercise 2.2.** Prove Theorem 2.5.

Chapter 3: The λ-Calculus

The B language is too restrictive to be useful for any real programming task. Since it has no abstraction capability—i.e., no ability to define functions—it is not remotely as powerful as a realistic programming language.

In this chapter, we study a language called the **λ-calculus**, which was invented by Church. Although the syntax and reduction relations for the λ-calculus are minimal, it is closely related to useful languages like Scheme and ML. In those languages, functions not only manipulate booleans, integers, and pairs, but also other functions. In other words, functions are values. For example, the `map` function consumes a function and a list of elements, and applies the function to each element of the list. Similarly, a `derivative` function might take a function (over real numbers) and return a new function that implements its derivative.

In the λ-calculus, the *only* kind of value is a function, but we will see how other kinds of values (including booleans, integers, and pairs) can be defined in terms of functions.

3.1 Functions in the λ-Calculus

The syntax of the λ-calculus provides a simple, regular method for writing down functions for application, and also as the inputs and outputs of other functions. The specification of such functions in the λ-calculus concentrates on the rule for going from an argument to a result, and ignores the issues of naming the function and its domain and range. For example, while a mathematician specifies the identity function on some set A as

$$\forall x \in A, f(x) = x$$

or

$$f : \left\{ \begin{array}{ccc} A & \longrightarrow & A \\ x & \mapsto & x \end{array} \right.$$

in the λ-calculus syntax, we write

$$(\lambda x.x)$$

An informal reading of the expression $(\lambda x.x)$ says: "if the argument is called x, then the output of the function is x," In other words, the function outputs the datum that it inputs.

To write down the application of a function f to an argument a, the λ-calculus uses ordinary mathematical syntax, modulo the placement of parentheses:

$$(f\ a)$$

For example, the expression representing the application of the identity function to a is:

$$((\lambda x.x)\ a)$$

Another possible argument for the identity function is itself:

$$((\lambda x.x)\ (\lambda x.x))$$

Here is an expression representing a function that takes an argument, ignores it, and returns the identity function:

$$(\lambda y.(\lambda x.x))$$

Here is an expression representing a function that takes an argument and returns a function; the returned function ignores its own argument, and returns the argument of the original function:

$$(\lambda y.(\lambda x.y))$$

The λ-calculus supports only single-argument functions, but this last example shows how a function can effectively take two arguments, x and y, by consuming the first argument, then returning another function to get the second argument. This technique is called **currying**.

In conventional mathematical notation, $f(a)$ can be "simplified" by taking the expression for the definition of f, and substituting a everywhere for f's formal argument. For example, given $f(x) = x$, $f(a)$ can be simplified to a. Simplification of λ-calculus terms is similar: $((\lambda x.x)\ a)$ can be simplified to a, which is the result of taking the body (the part after the dot) of $(\lambda x.x)$, and replacing the formal argument x (the part before the dot) with the actual argument a. Here are several examples:

$$\begin{array}{lcll} ((\lambda x.x)\ a) & \rightarrow & a & \text{by replacing } x \text{ with } a \text{ in } x \\ ((\lambda x.x)\ (\lambda y.y)) & \rightarrow & (\lambda y.y) & \text{by replacing } x \text{ with } (\lambda y.y) \text{ in } x \\ ((\lambda y.(\lambda x.y))\ a) & \rightarrow & (\lambda x.a) & \text{by replacing } y \text{ with } a \text{ in } (\lambda x.y) \end{array}$$

3.2 λ-Calculus Grammar and Reductions

The general grammar for expressions in the λ-calculus is defined by M (with aliases N and L):

$$\begin{array}{lcl} M, N, L & = & X \\ & | & (\lambda X.M) \\ & | & (M\ M) \\ X & = & \text{a variable: } x, y, \ldots \end{array}$$

The following are example members of M:

$$\begin{array}{ccc} x & (x\ y) & ((x\ y)\ (z\ w)) \\ (\lambda x.x) & & (\lambda y.(\lambda z.y)) \\ (f(\lambda y.(y\ y))) & & ((\lambda y.(y\ y))\ (\lambda y.(y\ y))) \end{array}$$

The first example, x, has no particular intuitive meaning, since x is left undefined. Similarly, $(x\ y)$ means "x applied to y", but we cannot say more, since x and y are undefined. In contrast, the example $(\lambda x.x)$ corresponds to the identity function. The difference between the last example and the first two is that x appears **free** in the first two expressions, but it only appears **bound** in the last one.

The relation $\mathcal{FV}$, maps an expression to a set of **free variables** in the expression. Intuitively, x is a free variable in an expression if it appears outside of any $(\lambda x.\_)$. More formally, we define the relation $\mathcal{FV}$ as follows:

$$\begin{array}{lcl} \mathcal{FV}(\ X\ ) & = & \{X\} \\ \mathcal{FV}(\ (\lambda X.M)\ ) & = & \mathcal{FV}(M) \setminus \{X\} \\ \mathcal{FV}(\ (M_1\ M_2)\ ) & = & \mathcal{FV}(M_1) \cup \mathcal{FV}(M_2) \end{array}$$

Examples for $\mathcal{FV}$:

$$\begin{array}{lcl} \mathcal{FV}(\ x\ ) & = & \{x\} \\ \mathcal{FV}(\ (x\ (y\ x))\ ) & = & \{x, y\} \\ \mathcal{FV}(\ (\lambda x.(x\ y))\ ) & = & \{y\} \\ \mathcal{FV}(\ (z\ (\lambda z.z))\ ) & = & \{z\} \end{array}$$

Before defining a reduction relation on λ-calculus expressions, we need one more auxilliary relation to deal with variable subsitition. The relation $\_[\_ \leftarrow \_]$ maps a source expression, a variable, and an argument expression to a target expression. The target expression is the same

as the source expression, except that free instances of the variable are replaced by the argument expression:

$$
\begin{array}{lcl}
X_1[X_1 \leftarrow M] & = & M \\
X_2[X_1 \leftarrow M] & = & X_2 \\
 & & \text{where } X_1 \neq X_2 \\
(\lambda X_1.M_1)[X_1 \leftarrow M_2] & = & (\lambda X_1.M_1) \\
(\lambda X_1.M_1)[X_2 \leftarrow M_2] & = & (\lambda X_3.M_1[X_1 \leftarrow X_3][X_2 \leftarrow M_2]) \\
 & & \text{where } X_1 \neq X_2,\ X_3 \notin \mathcal{FV}(M_2) \\
 & & \quad \text{and } X_3 \notin \mathcal{FV}(M_1) \backslash \{X_1\} \\
(M_1\ M_2)[X \leftarrow M_3] & = & (M_1[X \leftarrow M_3]\ M_2[X \leftarrow M_3])
\end{array}
$$

Examples for $\_[\_ \leftarrow \_]$:

$$
\begin{array}{lcl}
x[x \leftarrow (\lambda y.y)] & = & (\lambda y.y) \\
z[x \leftarrow (\lambda y.y)] & = & z \\
(\lambda x.x)[x \leftarrow (\lambda y.y)] & = & (\lambda x.x) \\
(\lambda y.(x\ y))[x \leftarrow (\lambda y.y)] & = & (\lambda z.((\lambda y.y)\ z)) \ \text{ or } \ (\lambda y.((\lambda y.y)\ y)) \\
(\lambda y.(x\ y))[x \leftarrow (\lambda x.y)] & = & (\lambda z.((\lambda x.y)\ z))
\end{array}
$$

Finally, to define the general reduction relation $\mathbf{n}$ for the λ-calculus, we first define three simple reduction relations, α, β, and η:

$$
\begin{array}{llll}
(\lambda X_1.M) & \alpha & (\lambda X_2.M[X_1 \leftarrow X_2]) & \text{where } X_2 \notin \mathcal{FV}(M) \\
((\lambda X.M_1)\ M_2) & \beta & M_1[X \leftarrow M_2] & \\
(\lambda X.(M\ X)) & \eta & M & \text{where } X \notin \mathcal{FV}(M)
\end{array}
$$

- The α relation renames a formal argument. It encodes the fact that functions like $(\lambda x.x)$ and $(\lambda y.y)$ are the same function, simply expressed with different names for the argument.
- The β relation is the main reduction relation, encoding function application.
- The η relation encodes the fact that, if a function f takes its argument and immediately applies the argument to g, then using f is equivalent to using g.

The general reduction relation $\mathbf{n}$ is the union of α, β, and η:

$$\mathbf{n} = \alpha \cup \beta \cup \eta$$

As usual, we define $\rightarrow_{\mathbf{n}}$ as the compatible closure of $\mathbf{n}$, $\twoheadrightarrow_{\mathbf{n}}$ as the reflexive–transitive closure of $\rightarrow_{\mathbf{n}}$, and $=_{\mathbf{n}}$ as the symmetric closure of $\twoheadrightarrow_{\mathbf{n}}$. We also define $\rightarrow^{\alpha}_{\mathbf{n}}$, $\rightarrow^{\beta}_{\mathbf{n}}$, and $\rightarrow^{\eta}_{\mathbf{n}}$ as the compatible closures of α, β, and η, respectively. (The compatible closure of α would normally be written $\rightarrow_{\alpha}$, but we use $\rightarrow^{\alpha}_{\mathbf{n}}$ to emphasize that $\rightarrow_{\mathbf{n}} = \rightarrow^{\alpha}_{\mathbf{n}} \cup \rightarrow^{\beta}_{\mathbf{n}} \cup \rightarrow^{\eta}_{\mathbf{n}}$.)

Here is one of many possible reductions for $((\lambda x.((\lambda z.z)\ x))\ (\lambda x.x))$, where the underlined portion of the expression is the redex (the part that is reduced by $\mathbf{n}$) in each step:

$$
\begin{array}{lll}
((\lambda x.((\lambda z.z)\ x))\ \underline{(\lambda x.x)}) & \rightarrow^{\alpha}_{\mathbf{n}} & (\underline{(\lambda x.((\lambda z.z)\ x))}\ (\lambda y.y)) \\
 & \rightarrow^{\eta}_{\mathbf{n}} & \underline{((\lambda z.z)\ (\lambda y.y))} \\
 & \rightarrow^{\beta}_{\mathbf{n}} & (\lambda y.y)
\end{array}
$$

Here is another reduction of the same expression:

$$
\begin{array}{lll}
((\lambda x.\underline{((\lambda z.z)\ x)})\ (\lambda x.x)) & \rightarrow^{\beta}_{\mathbf{n}} & \underline{((\lambda x.x)\ (\lambda x.x))} \\
 & \rightarrow^{\beta}_{\mathbf{n}} & (\lambda x.x)
\end{array}
$$

The parentheses in an expression are often redundant, and they can make large expressions difficult to read. Thus, we adopt a couple of conventions for dropping parentheses, plus one for dropping λs:

- Application associates to the left: $M_1\ M_2\ M_3$ means $((M_1\ M_2)\ M_3)$
- Application is stronger than abstraction: $\lambda X.M_1 M_2$ means $(\lambda X.(M_1\ M_2))$
- Consecutive lambdas can be collapsed: $\lambda XYZ.M$ means $(\lambda X.(\lambda Y.(\lambda Z.M)))$

With these conventions, $((\lambda x.((\lambda z.z)\ x))\ (\lambda x.x))$ can be abbreviated $(\lambda x.(\lambda z.z)\ x)\ \lambda x.x$, and the first reduction above can be abbreviated as follows:

$$\begin{array}{lll} (\lambda x.(\lambda z.z)\ x)\ \underline{\lambda x.x} & \to^{\alpha}_{\mathbf{n}} & \underline{(\lambda x.(\lambda z.z)\ x)}\ \lambda y.y \\ & \to^{\eta}_{\mathbf{n}} & \underline{(\lambda z.z)\ \lambda y.y} \\ & \to^{\beta}_{\mathbf{n}} & \lambda y.y \end{array}$$

▷ **Exercise 3.1.** Reduce the following expressions with $\to_{\mathbf{n}}$ until no more $\to^{\beta}_{\mathbf{n}}$ reductions are possible. Show all steps.

- $(\lambda x.x)$
- $(\lambda x.(\lambda y.y\ x))\ (\lambda y.y)\ (\lambda x.x\ x)$
- $(\lambda x.(\lambda y.y\ x))\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$

▷ **Exercise 3.2.** Prove the following equivalences by showing reductions.

- $(\lambda x.x) =_{\mathbf{n}} (\lambda y.y)$
- $(\lambda x.(\lambda y.(\lambda z.z\ z)\ y)\ x)(\lambda x.x\ x) =_{\mathbf{n}} (\lambda a.a\ ((\lambda g.g)\ a))\ (\lambda b.b\ b)$
- $\lambda y.(\lambda x.\lambda y.x)\ (y\ y) =_{\mathbf{n}} \lambda a.\lambda b.(a\ a)$
- $(\lambda f.\lambda g.\lambda x.f\ x\ (g\ x))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x) =_{\mathbf{n}} \lambda x.x$

3.3 Encoding Booleans

For the B languages, we arbitrarily chose the symbols `f` and `t` to correspond with "false" and "true", respectively. In the λ-calculus, we make a different choice—which, though arbitrary in principle, turns out to be convenient:

$$\begin{array}{lll} \mathsf{true} & \doteq & \lambda x.\lambda y.x \\ \mathsf{false} & \doteq & \lambda x.\lambda y.y \\ \mathsf{if} & \doteq & \lambda v.\lambda t.\lambda f.v\ t\ f \end{array}$$

The $\doteq$ notation indicates that we are defining a shorthand, or "macro", for an expression. The macros for true, false, and if will be useful if they behave in a useful way. For example, we would expect that

$$\mathsf{if}\ \mathsf{true}\ M\ N =_{\mathbf{n}} M$$

for any M and N. We can show that this equation holds by expanding the macros:

$$\begin{array}{lcl} \mathsf{if\ true}\ M\ N & = & (\lambda v.\lambda t.\lambda f.v\ t\ f)\ (\lambda x.\lambda y.x)\ M\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda t.\lambda f.(\lambda x.\lambda y.x)\ t\ f)\ M\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda f.(\lambda x.\lambda y.x)\ M\ f)\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda x.\lambda y.x)\ M\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda y.M)\ N \\ & \to^{\beta}_{\mathbf{n}} & M \end{array}$$

Similarly, $\mathsf{if\ false}\ M\ N =_{\mathbf{n}} N$:

$$\begin{array}{lcl} \mathsf{if\ false}\ M\ N & = & (\lambda v.\lambda t.\lambda f.v\ t\ f)\ (\lambda x.\lambda y.y)\ M\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda t.\lambda f.(\lambda x.\lambda y.y)\ t\ f)\ M\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda f.(\lambda x.\lambda y.y)\ M\ f)\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda x.\lambda y.y)\ M\ N \\ & \to^{\beta}_{\mathbf{n}} & (\lambda y.y)\ N \\ & \to^{\beta}_{\mathbf{n}} & N \end{array}$$

Actually, it turns out that $(\mathsf{if\ true}) =_{\mathbf{n}} \mathsf{true}$ and $(\mathsf{if\ false}) =_{\mathbf{n}} \mathsf{false}$. In other words, our representation of true acts like a conditional that branches on its first argument, and false acts like a conditional that branches on its second argument; the if macro is simply for readability.

▷ **Exercise 3.3.** Show that $(\mathsf{if\ true}) =_{\mathbf{n}} \mathsf{true}$ and $(\mathsf{if\ false}) =_{\mathbf{n}} \mathsf{false}$.

▷ **Exercise 3.4.** Define macros for binary and and or prefix operators that evaluate in the natural way with true and false (so that $\mathsf{and\ true\ false} =_{\mathbf{n}} \mathsf{false}$, etc.).

3.4 Encoding Pairs

To encode pairs, we need three operations: one that combines two values, one that extracts the first of the values, and one that extracts the second of the values. In other words, we need functions mkpair, fst, and snd that obey the following equations:

$$\begin{array}{lcl} \mathsf{fst}\ (\mathsf{mkpair}\ M\ N) & =_{\mathbf{n}} & M \\ \mathsf{snd}\ (\mathsf{mkpair}\ M\ N) & =_{\mathbf{n}} & N \end{array}$$

We will also use the notation $\langle M, N\rangle$ as shorthand for the pair whose first element is M and whose second element is N. One way to find definitions for mkpair, etc. is to consider what a $\langle M, N\rangle$ value might look like.

Since our only values are functions, $\langle M, N\rangle$ must be a function. The function has to contain inside it the expressions M and N, and it has to have some way of returning one or the other to a user of the pair, depending on whether the user wants the first or second element. This suggests that a user of the pair should call the pair as a function, supplying true to get the first element, or false to get the second one:

$$\langle M, N\rangle \doteq \lambda s.\mathsf{if}\ s\ M\ N$$

As mentioned in the previous section, the if function is really not necessary, and the above can be simplified by dropping the if.

With this encoding, the fst function takes a pair, then applies it as a function to the argument true:

$$\mathsf{fst} \doteq \lambda p.p\ \mathsf{true}$$

Similarly, snd applies its pair argument to false. Finally, to define mkpair, we abstract the abbreviation of $\langle M, N\rangle$ over arbitrary M and N.

$$\begin{array}{lcl} \langle M, N\rangle & \doteq & \lambda s.s\ M\ N \\ \mathsf{mkpair} & \doteq & \lambda x.\lambda y.\lambda s.s\ x\ y \\ \mathsf{fst} & \doteq & \lambda p.p\ \mathsf{true} \\ \mathsf{snd} & \doteq & \lambda p.p\ \mathsf{false} \end{array}$$

▷ **Exercise 3.5.** Show that mkpair, fst, and snd obey the equations at the beginning of this section.

3.5 Encoding Numbers

There are many ways to encode numbers in the λ-calculus, but the most popular encoding is due to Church, and the encoded numbers are therefore called **Church numerals**. The idea is that a natural number n is encoded by a function of two arguments, f and x, where the function applies f to x n times. Thus, the function for 0 takes an f and x and returns x (which corresponds to applying f zero times). The function 1 applies f one time, and so on.

$$\begin{array}{rcl} 0 & \doteq & \lambda f.\lambda x.x \\ 1 & \doteq & \lambda f.\lambda x.f\ x \\ 2 & \doteq & \lambda f.\lambda x.f\ (f\ x) \\ 3 & \doteq & \lambda f.\lambda x.f\ (f\ (f\ x)) \\ & \ldots & \end{array}$$

The function add1 should take the representation of a number n and produce the representation of a number $n + 1$. In other words, it takes a 2-argument function and returns another 2-argument function; the new function applies its first argument to the second argument $n + 1$ times. To get the first n applications, the new function can use the old one.

$$\mathsf{add1} \doteq \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

Like our encoding of true and false, this encoding of numbers turns out to be convenient. To add two numbers n and m, all we have to do is apply add1 to n m times—and m happens to be a function that will take add1 and apply it m times!

$$\mathsf{add} \doteq \lambda n.\lambda m.m\ \mathsf{add1}\ n$$

The idea of using the number as a function is also useful for defining iszero, a function that takes a number and returns true if the number is 0, false otherwise. We can implement iszero by using a function that ignores its argument and always returns false; if this function is applied 0 times to true, the result will be true, otherwise the result will be false.

$$\mathsf{iszero} \doteq \lambda n.n\ (\lambda x.\mathsf{false})\ \mathsf{true}$$

To generalize iszero to number equality, we need subtraction. In the same way that addition can be built on add1, subtraction can be built on sub1. But, although the add1, add, and iszero functions are fairly simple in Church's number encoding, sub1 is more complex. The number function that sub1 receives as its argument applies a function n times, but the function returned by sub1 should apply the function one *less* time. Of course, the inverse of an arbitrary function is not available for reversing one application.

The function implementing sub1 has two parts:

- Pair the given argument x with the value true. The true indicates that an application of f should be skipped.

- Wrap the given function f to take pairs, and to apply f only if the pair contains false. Always return a pair containing false, so that f will be applied in the future.

The function wrap wraps a given f:

$$\mathsf{wrap} \doteq \lambda f.\lambda p.\langle \mathsf{false}, \mathsf{if}\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p)\ (f\ (\mathsf{snd}\ p))\rangle$$

The function sub1 takes an n and returns a new function. The new function takes f and x, wraps the f with wrap, pairs x with true, uses n on (wrap f) and $\langle \mathsf{true}, x\rangle$, and extracts the second part of the result—which is f applied to x $n-1$ times.

$$\mathsf{sub1} \doteq \lambda n.\lambda f.\lambda x.\mathsf{snd}\ (n\ (\mathsf{wrap}\ f)\ \langle \mathsf{true}, x\rangle)$$

A note on encodings: The encoding for 0 is exactly the same as the encoding for false. Thus, no program can distinguish 0 form false, and programmers must be careful that only true and false appear in boolean contexts, etc. This is analogous to the way that C uses the same pattern of bits to implement 0, false, and the null pointer.

▷ **Exercise 3.6.** Show that $\mathsf{add1}\ 1 =_{\mathbf{n}} 2$.

▷ **Exercise 3.7.** Show that $\mathsf{iszero}\ 1 =_{\mathbf{n}} \mathsf{false}$.

▷ **Exercise 3.8.** Show that $\mathsf{sub1}\ 1 =_{\mathbf{n}} 0$.

▷ **Exercise 3.9.** Define mult using the technique that allowed us to define add. In other words, implement (mult n m) as n additions of m to 0 by exploiting the fact that n itself applies a function n times. Hint: what kind of value is (add m)?

▷ **Exercise 3.10.** The λ-calculus provides no mechanism for signalling an error. What happens when sub1 is applied to 0? What happens when iszero is applied to true?

3.6 Recursion

An exercise in the previous section asks you to implement mult in the same way that we implemented add. Such an implementation exploits information about the way that numbers are encoded by functions.

Given the functions iszero, add, and sub1 from the previous section, we can also implement mult without any knowledge of the way that numbers are encoded (which is the way that programmers normally implement functions). We must define a recursive program that checks whether the first argument is 0, and if not, adds the second argument to a recursive call that decrements the first argument.

$$\mathsf{mult} \stackrel{?}{\doteq} \lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ (\mathsf{mult}\ (\mathsf{sub1}\ n)\ m))$$

The problem with the above definition of the macro mult is that it refers to itself (i.e., it's recursive), so there is no way to expand mult to a pure λ-calculus expression. Consequently, the abbreviation is illegal.

3.6.1 Recursion via Self-Application

How can the multiplier function get a handle to itself? The definition of the multipler function is not available as the mult macro is defined, but the definition is available later. In particular, when we call the multiplier function, we necesarily have a handle to the multiplier function.

Thus, instead of referring to itself directly, the multiplier function could have us supply a multiply function t (itself) as well as arguments to multiply. More precisely, using this strategy, the macro we define is no longer a multiplier function, but a multiplier *maker* instead: it takes some function t then produces a multiplier function that consumes two more arguments to multiply:

$$\mathsf{mkmult}_0 \doteq \lambda t.\lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ (t\ (\mathsf{sub1}\ n)\ m))$$

This mkmult_0 macro is well defined, and $(\mathsf{mkmult}_0\ t)$ produces a multiplication function... assuming that t is a multiplication function! Obviously, we still do not have a multiplication function. We tried to parameterize the original mult definition by itself, but in doing so we lost the mult definition.

Although we can't supply a multiplication function as t, we could instead supply a multiplication maker as t. Would that be useful, or do we end up in an infinite regress? As it turns out, supplying a maker to a maker can work.

Assume that applying a maker to a maker produces a multiplier. Therefore, the initial argument t to a maker will be a maker. In the body of the maker, wherever a multiplier is needed, we use $(t\ t)$—because t will be a maker, and applying a maker to itself produces a multiplier. Here is a maker, mkmult_1 that expects a maker as its argument:

$$\mathsf{mkmult}_1 \doteq \lambda t.\lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ ((t\ t)\ (\mathsf{sub1}\ n)\ m))$$

If $\mathsf{mkmulti}_1$ works, then we can get a mult function by applying $\mathsf{mkmulti}_1$ to itself:

$$\mathsf{mult} \doteq (\mathsf{mkmult}_1\ \mathsf{mkmult}_1)$$

Let's try this suspicious function on 0 and m (for some arbitrary m) to make sure that we get 0 back. We'll expand abbreviations only as necessary, and we'll assume that abbreivations like iszero and 0 behave in the expected way:

$$\begin{array}{ll}
\mathsf{mult}\ 0\ m & \\
\quad = & (\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ 0\ m \\
\rightarrow_{\mathbf{n}} & (\lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ n)\ m)))\ 0\ m \\
\rightarrow_{\mathbf{n}} & (\lambda m.\mathsf{if}\ (\mathsf{iszero}\ 0)\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ 0)\ m)))\ m \\
\rightarrow_{\mathbf{n}} & \mathsf{if}\ (\mathsf{iszero}\ 0)\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ 0)\ m)) \\
\rightarrow\!\!\rightarrow_{\mathbf{n}} & \mathsf{if}\ \mathsf{true}\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ 0)\ m)) \\
\rightarrow\!\!\rightarrow_{\mathbf{n}} & 0
\end{array}$$

So far, so good. What if we multiply n and m, for some $n \neq 0$?

$$\begin{array}{ll}
\mathsf{mult}\ n\ m & \\
\quad = & (\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ n\ m \\
\rightarrow_{\mathbf{n}} & (\lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ n)\ m)))\ n\ m \\
\rightarrow_{\mathbf{n}} & (\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ n)\ m)))\ m \\
\rightarrow_{\mathbf{n}} & \mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ n)\ m)) \\
\rightarrow\!\!\rightarrow_{\mathbf{n}} & \mathsf{if}\ \mathsf{false}\ 0\ (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ n)\ m)) \\
\rightarrow\!\!\rightarrow_{\mathbf{n}} & (\mathsf{add}\ m\ ((\mathsf{mkmult}_1\ \mathsf{mkmult}_1)\ (\mathsf{sub1}\ n)\ m))
\end{array}$$

Since $\mathsf{mult} = (\mathsf{mkmult}_1\ \mathsf{mkmult}_1)$, the last step above can also be written as $(\mathsf{add}\ m\ (\mathsf{mult}\ (\mathsf{sub1}\ n)\ m))$. Thus,

$$\begin{array}{lll} \mathsf{mult}\ 0\ m & \rightarrow\!\!\!\!\rightarrow_{\mathbf{n}} & 0 \\ \mathsf{mult}\ n\ m & \rightarrow\!\!\!\!\rightarrow_{\mathbf{n}} & (\mathsf{add}\ m\ (\mathsf{mult}\ (\mathsf{sub1}\ n)\ m)) \quad \text{if } n \neq 0 \end{array}$$

This is exactly the relationship we want among mult, add, sub1, and 0.

▷ **Exercise 3.11.** Define a macro mksum such that (mksum mksum) acts like a "sum" function by consuming a number n and adding all the numbers from 0 to n.

3.6.2 Lifting Out Self-Application

The technique of the previous section will let us define any recursive function that we want. It's somewhat clumsy, though, because we have to define a maker function that applies an initial argument to itself for every recursive call. For convenience, we would like to pull the self-application pattern out into its own abstraction.

More concretely, we want a function, call it mk, that takes any maker like mkmult_0 and produces a made function. For example, $(\mathsf{mk}\ \mathsf{mkmult}_0)$ should be a multiplier.

$$\mathsf{mk} \stackrel{?}{\doteq} \lambda t.t\ (\mathsf{mk}\ t)$$

The mk definition above is ill-formed, but we can start with this bad definition to get the idea. The mk function is supposed to take a maker, t, and produce a made function. It does so by calling the function-expecting maker with $(\mathsf{mk}\ t)$ — which is supposed to create a made function.

We can fix the circular mk definition using the technique of the previous section:

$$\mathsf{mkmk} \doteq \lambda k.\lambda t.t\ ((k\ k)\ t)$$

$$\mathsf{mk} \doteq (\mathsf{mkmk}\ \mathsf{mkmk})$$

We can check that $(\mathsf{mk}\ \mathsf{mkmult}_0)$ behaves like mult:

$$\begin{array}{ll} & (\mathsf{mk}\ \mathsf{mkmult}_0)\ 0\ m \\ = & ((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ 0\ m \\ = & (((\lambda k.\lambda t.t\ ((k\ k)\ t))\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ 0\ m \\ \rightarrow_{\mathbf{n}} & ((\lambda t.t\ ((\mathsf{mkmk}\ \mathsf{mkmk})\ t)\ \mathsf{mkmult}_0)\ 0\ m \\ \rightarrow_{\mathbf{n}} & (\mathsf{mkmult}_0\ ((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0))\ 0\ m \\ \rightarrow_{\mathbf{n}} & (\lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ n)\ m)))\ 0\ m \\ \rightarrow_{\mathbf{n}} & (\lambda m.\mathsf{if}\ (\mathsf{iszero}\ 0)\ 0\ (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ 0)\ m)))\ m \\ \rightarrow_{\mathbf{n}} & \mathsf{if}\ (\mathsf{iszero}\ 0)\ 0\ (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ 0)\ m)) \\ \rightarrow\!\!\!\!\rightarrow_{\mathbf{n}} & 0 \end{array}$$

$$\begin{array}{ll} & (\mathsf{mk}\ \mathsf{mkmult}_0)\ n\ m \\ = & ((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ n\ m \\ = & (((\lambda k.\lambda t.t\ ((k\ k)\ t))\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ n\ m \\ \rightarrow_{\mathbf{n}} & ((\lambda t.t\ ((\mathsf{mkmk}\ \mathsf{mkmk})\ t)\ \mathsf{mkmult}_0)\ n\ m \\ \rightarrow_{\mathbf{n}} & (\mathsf{mkmult}_0\ ((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0))\ n\ m \\ \rightarrow_{\mathbf{n}} & (\lambda n.\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ n)\ m)))\ 0\ m \\ \rightarrow_{\mathbf{n}} & (\lambda m.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ n)\ m)))\ m \\ \rightarrow_{\mathbf{n}} & \mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ n)\ m)) \\ \rightarrow\!\!\!\!\rightarrow_{\mathbf{n}} & (\mathsf{add}\ m\ (((\mathsf{mkmk}\ \mathsf{mkmk})\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ n)\ m)) \\ = & (\mathsf{add}\ m\ ((\mathsf{mk}\ \mathsf{mkmult}_0)\ (\mathsf{sub1}\ n)\ m)) \end{array}$$

3.6.3 Fixed Points and the Y Combinator

The mk function should seem somewhat mysterious at this point. Somehow, it makes mkmult_0 useful, even though mkmult_0 can only make a multiplier when it is given the multiplier that it is supposed to make!

In general, mkmult_0 might be given any function, and the resulting "multiplier" that it returns would behave in many different ways for many different input functions, and the output function would typically be quite different from the input function. But mk somehow manages to pick an input function for mkmult_0 so that the output function is the same as the input function. In other words, mk finds the **fixed point** of mkmult_0.

As it turns out, mk can find the fixed point of any function. In other words, if applying mk to M produces N, then applying M to N simply produces N again.

> **Theorem 3.1**: $M\ (\mathsf{mk}\ M) =_{\mathbf{n}} (\mathsf{mk}\ M)$ for any M
>
> **Proof for Theorem 3.1**: Since $=_{\mathbf{n}}$ is the symmetric closure of $\twoheadrightarrow_{\mathbf{n}}$, we can prove the claim by showing that $\mathsf{mk}\ M \twoheadrightarrow_{\mathbf{n}} M\ (\mathsf{mk}\ M)$:
>
> $$\begin{array}{rcl} \mathsf{mk}\ M & = & (\mathsf{mkmk}\ \mathsf{mkmk})\ M \\ & = & ((\lambda k.\lambda t.t\ ((k\ k)\ t))\ \mathsf{mkmk})\ M \\ & \rightarrow_{\mathbf{n}} & (\lambda t.t\ ((\mathsf{mkmk}\ \mathsf{mkmk})\ t)\ M \\ & \rightarrow_{\mathbf{n}} & M\ ((\mathsf{mkmk}\ \mathsf{mkmk})\ M) \\ & = & M\ (\mathsf{mk}\ M) \end{array}$$

A function that behaves like mk is called a **fixed-point operator**. The mk function is only one of many fixed-point operators in the λ-calculus. The more famous one is called Y:[1]

$$\mathsf{Y} \doteq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

In general, Y lets us define recursive functions more easily than the manual technique of Section 3.6.1. For example, we can define sum as

$$\mathsf{sum} \doteq \mathsf{Y}\ (\lambda s.\lambda n.\mathsf{if}\ (\mathsf{iszero}\ n)\ 0\ (\mathsf{add}\ n\ (s\ (\mathsf{sub1}\ n))))$$

Since we do not have to repeat a large maker expression, we can skip the intermediate maker abbreviations mksum, and instead apply Y directly to a maker function.

In addition, a programmer who sees the above definition of sum will immediately note the Y, see that s is the argument of Y's argument, and then read $\lambda n \ldots$ as the definition of a recursive function named s.

▷ **Exercise 3.12.** Prove that $M\ (\mathsf{Y}\ M) =_{\mathbf{n}} (\mathsf{Y}\ M)$ for any M.

▷ **Exercise 3.13.** Define an encoding for Lisp cons cells, consisting of the following macros:

- null, a constant
- cons, a function that takes two arguments and returns a cons cell
- isnull, a function that returns true if its argument is null, false if it is a cons cell
- car, a function that takes a cons cell and returns its first element
- cdr, a function that takes a cons cell and returns its second element

[1] The term Y **combinator** actually refers to the whole family of fixed-point operators. More on this later.

In particular, your encoding must satisfy the following equations:

$$\begin{array}{lll} (\mathsf{isnull}\ \mathsf{null}) & =_{\mathbf{n}} & \mathsf{true} \\ (\mathsf{isnull}\ (\mathsf{cons}\ M\ N)) & =_{\mathbf{n}} & \mathsf{false} \\ (\mathsf{car}\ (\mathsf{cons}\ M\ N)) & =_{\mathbf{n}} & M \\ (\mathsf{cdr}\ (\mathsf{cons}\ M\ N)) & =_{\mathbf{n}} & N \end{array}$$

Your encoding need not assign any particular meaning to expressions such as (car null) or (car cons).

▷ **Exercise 3.14.** Using your encoding from the previous exercise, define length, which takes a list of booleans and returns the number of cons cells in the list. A list of booleans is either null, or (cons b l) where b is true or false and l is a list of booleans.

3.7 Reduction Strategy and Normal Form

When is an expression fully reduced? Almost any expression can be reduced by $\rightarrow^{\alpha}_{\mathbf{n}}$, which merely renames variables, so it shouldn't count in the definition. Instead, a fully reduced expression is one that cannot be reduced via $\rightarrow^{\beta}_{\mathbf{n}}$ or $\rightarrow^{\eta}_{\mathbf{n}}$.

> An expression is a **normal form** if it cannot be reduced by $\rightarrow^{\beta}_{\mathbf{n}}$ or $\rightarrow^{\eta}_{\mathbf{n}}$.
>
> M **has normal form** N if $M =_{\mathbf{n}} N$ and N is a normal form.

A normal form acts like the result of a λ-calculus expression. If an expression has a normal form, then it has exactly *one* normal form (possibly via many different reductions). More precisely, there is one normal form modulo $\rightarrow^{\alpha}_{\mathbf{n}}$ renamings.

> **Theorem 3.2 [Normal Forms]**: If $L =_{\mathbf{n}} M$, $L =_{\mathbf{n}} N$, and both M and N are normal forms, then $M =_{\alpha} N$.

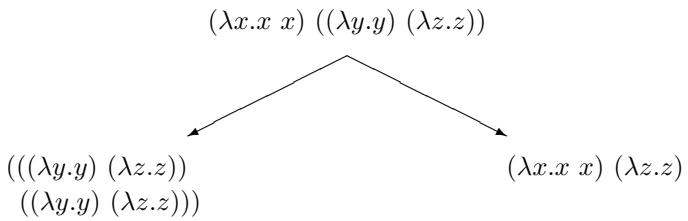
As usual, $=_{\alpha}$ is the equivalence generated by the compatible closure of α. Theorem 3.2 is easy to prove given the Church-Rosser property for the λ-calculus:

> **Theorem 3.3 [Church-Rosser for $=_{\mathbf{n}}$]**: If $M =_{\mathbf{n}} N$, then there exists an L' such that $M \twoheadrightarrow_{\mathbf{n}} L'$ and $N \twoheadrightarrow_{\mathbf{n}} L'$.

As for $=_{\mathbf{r}}$, then proof of this theorem hinges on the diamond property for $\twoheadrightarrow_{\mathbf{n}}$.

> **Theorem 3.4 [Diamond Property for $\twoheadrightarrow_{\mathbf{n}}$]**: If $L \twoheadrightarrow_{\mathbf{r}} M$ and $L \twoheadrightarrow_{\mathbf{r}} N$, then there exists an expression L' such that $M \twoheadrightarrow_{\mathbf{r}} L'$ and $N \twoheadrightarrow_{\mathbf{r}} L'$.

The one-step relation $\rightarrow_{\mathbf{n}}$ does not obey the diamond property, or even a slightly contorted one, such as the diamond-like property of $\rightarrow_{\mathbf{r}}$. The reason is that $\rightarrow^{\beta}_{\mathbf{n}}$ can duplicate reducible expressions. For example:



There's no single step that will bring both of the bottom expressions back together. As we'll see in the next chapter, the way around this problem is to define a notion of parallel reductions on independent subexpression, so that both $((\lambda y.y)\ (\lambda z.z))$ sub-expressions on the left can be reduced at once. For now, though, we will not try to prove Theorem 3.4. Instead, we will demonstrate the diamond property for a related language in the next chapter.

Unlike expressions in B, where every expression has a result `f` or `t`, not every λ-calculus expression has a normal form. For example, Ω has no normal form:

$$\Omega \doteq ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

Even if an expression has a normal form, then there may be an inifite reduction sequence for the expression that never reaches a normal form. For example:

$(\lambda y.\lambda z.z)((\lambda x.x\ x)\ (\lambda w.w\ w))$
$\rightarrow_{\mathbf{n}}\quad \lambda z.z$ — normal form

$(\lambda y.\lambda z.z)((\lambda x.x\ x)\ (\lambda w.w\ w))$
$\rightarrow_{\mathbf{n}}\quad (\lambda y.\lambda z.z)((\lambda w.w\ w)\ (\lambda w.w\ w))$
$\rightarrow_{\mathbf{n}}\quad (\lambda y.\lambda z.z)((\lambda w.w\ w)\ (\lambda w.w\ w))$
$\rightarrow_{\mathbf{n}}\quad \ldots$ — same expression forever

Thus, Theorem 3.2 guarantees that at most one normal form exists, but we do not yet have a way to find it. Intuitively, the problem with the non-terminating reduction above is that we are evaluating the argument of a function that will not be used in the function body. This suggests a strategy where we apply the leftmost β or η reduction in an expression to reach a normal form:

$M \rightarrow_{\bar{\mathbf{n}}} N$	if	$M\,\beta\,N$
$M \rightarrow_{\bar{\mathbf{n}}} N$	if	$M\,\eta\,N$
$(\lambda X.M) \rightarrow_{\bar{\mathbf{n}}} (\lambda X.N)$	if	$M \rightarrow_{\bar{\mathbf{n}}} N$
$(M\ N) \rightarrow_{\bar{\mathbf{n}}} (M'\ N)$	if	$M \rightarrow_{\bar{\mathbf{n}}} M'$
	and	$\forall L,\ (M\ N)\ \not\beta\ L$ [and $(M\ N)\ \not\eta\ L$]
$(M\ N) \rightarrow_{\bar{\mathbf{n}}} (M\ N')$	if	$N \rightarrow_{\bar{\mathbf{n}}} N'$
	and	M is a normal form
	and	$\forall L,\ (M\ N)\ \not\beta\ L$ [and $(M\ N)\ \not\eta\ L$]

The $\rightarrow_{\bar{\mathbf{n}}}$ relation is guaranteed to find a normal form if one exists.

Theorem 3.5 [Normal-Order Reduction]: M has normal form N if and only if $M \rightarrow\!\!\!\!\rightarrow_{\bar{\mathbf{n}}} N$.

Although a normal-order reduction always finds a normal form if it exists, practically no programming language uses this form of reduction. The reason is that this strategy, while powerful, is often slow. For example, in the previous diagram showing the non-diamondness of $\rightarrow_{\mathbf{n}}$, the reduction on the left corresponds to normal-order, but the reduction on the right produces the identity function in fewer steps. Another issue is that $\rightarrow_{\bar{\mathbf{n}}}$ evaluates inside of functions before they are applied.

In any case, since we have a notion of unique normal form, a natural question to ask is whether we can define an $eval_{\mathbf{n}}$ in the same way as $eval_{\mathbf{r}}$:

$$eval_{\mathbf{n}}(M) \stackrel{?}{=} N \text{ if } M =_{\mathbf{n}} N \text{ and } N \text{ is a normal form}$$

The above definition is imprecise with respect to α-renaming, but there is a deeper problem. As we have seen, there are functions like `mk` and `Y` that behave in the same way, but cannot be reduced to each other. In the next chapter, we will look at a way to resolve this problem.

▷ **Exercise 3.15.** Prove that $((\lambda x.x\ x)\ (\lambda x.x\ x))$ has no normal form.

3.8 History

Church invented the lambda calculus slightly before Turing invented Turing machines.

Barendregt [2] provides a comprehensive study of Church's λ-calculus as a logical system, and many conventions for the treatment of terms originate with Barendregt. His book provides numerous techniques applicable to the calculus, though it does not cover λ-calculus as a calculus for a programming language.

Part II

Models of Realistic Languages

Chapter 4: ISWIM

Church developed the λ-calculus as a way of studying all of mathematics, as opposed to mechanical computation specifically. In the 1960's Landin showed that Church's calculus was, in fact, not quite an appropriate model of most programming languages. For example, the λ-calculus expression

$$(\lambda x.1)\ (\texttt{sub1}\ \lambda y.y)$$

reduces to (the encoding of) 1, even though most languages would instead complain about (sub1 $\lambda y.y$). The problem is not just in the encoding of sub1 and $\lambda y.y$, but that a β reduction on the entire expression can ignore the argument (sub1 $\lambda y.y$) completely.

Whether such call-by-name behavior is desirable or not is open to debate. In any case, many languages do not support call-by-name. Instead, they support call-by-value, where the arguments to a function must be fully evaluated before the function is applied.

In this chapter, we introduce Landin's **ISWIM**, which more closely models the core of call-by-value languages such as Scheme and ML. The basic syntax of ISWIM is the same as for the λ-calculus, and the notions of free variables and substition are the same. Unlike the λ-calculus, and more like real programming languages, ISWIM comes with a set of basic constants and primitive operations. But the more fundamental difference is in ISWIM's call-by-value reduction rules.

4.1 ISWIM Expressions

The grammar for ISWIM extends the λ-calculus grammar:

M, N, L, K	$=$	X
	$\mid$	$(\lambda X.M)$
	$\mid$	$(M\ M)$
	$\mid$	b
	$\mid$	$(o^n\ M\ \ldots\ M)$
X	$=$	a variable: x, y, ...
b	$=$	a basic constant
o^n	$=$	an n-ary primitive operation

where an expression $(o^n\ M_1\ \ldots\ M_m)$ is valid only if $n = m$. We could define the sets b and o^n in a variety of ways; high-level properties of the language will remain unchanged. For concreteness, we will use the following definitions for b and o^n:

b	$=$	$\{\ulcorner n \urcorner \mid n \in \mathbb{Z}\}$
o^1	$=$	$\{\texttt{add1}, \texttt{sub1}, \texttt{iszero}\}$
o^2	$=$	$\{+, -, *, \uparrow\}$

The syntactic object $\ulcorner 1 \urcorner$ represents the integer 1. The syntactic object + represents an addition operator, $\uparrow$ is exponentiation, etc. In addition, we define a helpful if0 macro:

$$(\textsf{if0}\ K\ M\ N) \doteq (((\texttt{iszero}\ K)\ (\lambda X.M)\ (\lambda X.N))\ \ulcorner 0 \urcorner)$$
$$\text{where } X \notin \mathcal{FV}(M) \cup \mathcal{FV}(N)$$

The $\mathcal{FV}$ and $\_[\_ \leftarrow \_]$ relations extend to the new grammar in the obvious way:

$$
\begin{array}{lcl}
\mathcal{FV}(\ b\ ) & = & \{\,\} \\
\mathcal{FV}(\ X\ ) & = & \{X\} \\
\mathcal{FV}(\ (\lambda X.M)\ ) & = & \mathcal{FV}(M) \setminus \{X\} \\
\mathcal{FV}(\ (M_1\ M_2)\ ) & = & \mathcal{FV}(M_1)\ \cup\ \mathcal{FV}(M_2) \\
\mathcal{FV}(\ (o^n M_1\ \ldots M_n)\ ) & = & \mathcal{FV}(M_1)\ \cup\ \ldots\ \mathcal{FV}(M_n) \\
 & & \\
b[X \leftarrow M] & = & b \\
X_1[X_1 \leftarrow M] & = & M \\
X_2[X_1 \leftarrow M] & = & X_2 \\
 & & \text{where } X_1 \neq X_2 \\
(\lambda X_1.M_1)[X_1 \leftarrow M_2] & = & (\lambda X_1.M_1) \\
(\lambda X_1.M_1)[X_2 \leftarrow M_2] & = & (\lambda X_3.M_1[X_1 \leftarrow X_3][X_2 \leftarrow M_2]) \\
 & & \text{where } X_1 \neq X_2,\ X_3 \notin \mathcal{FV}(M_2) \\
 & & \quad \text{and } X_3 \notin \mathcal{FV}(M_1)\backslash\{X_1\} \\
(M_1\ M_2)[X \leftarrow M_3] & = & (M_1[X \leftarrow M_3]\ M_2[X \leftarrow M_3]) \\
(o^n\ M_1\ \ldots\ M_n)[X \leftarrow M] & = & (o^n\ M_1[X \leftarrow M]\ \ldots\ M_n[X \leftarrow M])
\end{array}
$$

4.2 ISWIM Reductions

Since functions in ISWIM accept only fully-evaluated arguments, we must first define the set of **values**. The set of values V (aliases U and W) is a subset of the set of expressions.

$$
\begin{array}{rcl}
V, U, W & = & b \\
 & | & X \\
 & | & (\lambda X.M)
\end{array}
$$

In particular, an application expression is never a value, while an abstraction (i.e., a function) is always a value, regardless of the shape of its body.

The core reduction relation for ISWIM is β_{v}, which is like β, except that the argument must be in V instead of M:

$$((\lambda X.M)\ V) \quad \beta_{\mathsf{v}} \quad M[X \leftarrow V]$$

Restricting the argument to be a member of V forces an order on evaluation, to some degree. For example, $((\lambda x.1)\ (\texttt{sub1}\ \lambda y.y))$ cannot be reduced to 1 with β_{v} because $(\texttt{sub1}\ \lambda y.y)$ is not a member of V. Similarly, $((\lambda x.1)\ (\texttt{sub1}\ 1))$ must be reduced to $((\lambda x.1)\ 0)$ then 1, in that order.

ISWIM does not have a relation analogous to η, which reflects the fact that a programming language implementation generally does not extract functions applied in the body of another function. Furthermore, from now on, we will generally use α-equivalence ($=_\alpha$) when comparing expressions, instead of viewing α as a reduction.

In addition to function application, the ISWIM reduction rules need to account for primitive operations. In the same way that b and o^n are abstract in principle (even if we define concrete sets for examples), the reductions associated with the primitive operations are represented by an abstract δ. The δ relation maps each o^n plus n basic constants to a value. For concreteness,

we choose the following δ:

$$\begin{array}{lll}
(\texttt{add1}\ \ulcorner m\urcorner) & \mathbf{b}^1 & \ulcorner m+1\urcorner \\
(\texttt{sub1}\ \ulcorner m\urcorner) & \mathbf{b}^1 & \ulcorner m-1\urcorner \\
(\texttt{iszero}\ \ulcorner 0\urcorner) & \mathbf{b}^1 & \lambda xy.x \\
(\texttt{iszero}\ \ulcorner n\urcorner) & \mathbf{b}^1 & \lambda xy.y \qquad n \neq 0 \\
 & & \\
(+\ \ulcorner m\urcorner\ \ulcorner n\urcorner) & \mathbf{b}^2 & \ulcorner m+n\urcorner \\
(-\ \ulcorner m\urcorner\ \ulcorner n\urcorner) & \mathbf{b}^2 & \ulcorner m-n\urcorner \\
(*\ \ulcorner m\urcorner\ \ulcorner n\urcorner) & \mathbf{b}^2 & \ulcorner m\cdot n\urcorner \\
(\uparrow\ \ulcorner m\urcorner\ \ulcorner n\urcorner) & \mathbf{b}^2 & \ulcorner m^n\urcorner \\
 & & \\
\delta & = & \mathbf{b}^1 \cup \mathbf{b}^2
\end{array}$$

By combining β and δ, we arrive at the complete reduction relation $\mathbf{v}$:

$$\mathbf{v} = \beta_{\mathsf{v}} \cup \delta$$

As usual, $\rightarrow_{\mathbf{v}}$ is the compatible closure of $\mathbf{v}$, $\twoheadrightarrow_{\mathbf{v}}$ is the reflexive–transitive closure of $\rightarrow_{\mathbf{v}}$, and $=_{\mathbf{v}}$ is the symmetric closure of $\twoheadrightarrow_{\mathbf{v}}$.

▷ **Exercise 4.1.** Show a reduction of

$$(\lambda w.(-\ (w\ \ulcorner 1\urcorner)\ \ulcorner 5\urcorner))\ ((\lambda x.x\ \ulcorner 10\urcorner)\ \lambda yz.(+\ z\ y))$$

to a value with $\rightarrow_{\mathbf{v}}$.

4.3 The $\mathsf{Y_V}$ Combinator

For the pure λ-calculus, we defined a function Y that finds a fixed point of any expression, and can therefore be used to define recursive functions. Although it's stilll true in ISWIM that $(f\ (\mathsf{Y}\ f))$ is the same as $(\mathsf{Y}\ f)$ for any f, this fact does not turn out to be useful:

$$\begin{array}{rcl}
\mathsf{Y}\ f & = & (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ f \\
 & \rightarrow_{\mathbf{v}} & (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\
 & \rightarrow_{\mathbf{v}} & f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))
\end{array}$$

The problem is that the argument to f in the outermost application above is not a value—and cannot be reduced to a value—so $(\mathsf{Y}\ f)$ never produces the promised fixed-point function.

We can avoid the infinite reduction by changing each application M within Y to $(\lambda X.M\ X)$, since the application is supposed to return a function anyway. This inverse-η transformation puts a value in place of an infinitely reducing application. The final result is the $\mathsf{Y_V}$ combinator:

$$\begin{array}{l}
\mathsf{Y_V} = \ (\lambda f. \\
\qquad\qquad (\lambda x. \\
\qquad\qquad\quad (\ (\lambda g.\,(f\ (\lambda x.\,((g\ g)\ x)))) \\
\qquad\qquad\qquad (\lambda g.\,(f\ (\lambda x.\,((g\ g)\ x))))\ x)))
\end{array}$$

The $\mathsf{Y_V}$ combinator works when applied to a function that takes a function and returns another one.

Theorem 4.1 [Fixed-Point Theorem for $\mathsf{Y_V}$]: If $K = \lambda gx.L$, then $(K\ (\mathsf{Y_V}\ K)) =_{\mathbf{v}} (\mathsf{Y_V}\ K)$.

Proof for Theorem 4.1: The proof is a straightforward calculation where all of the basic proof steps are β_{v} steps:

$$\begin{array}{ll}
& (\mathsf{Y}_{\mathsf{V}}\ K) \\
= & ((\lambda f.\lambda x.((\lambda g.(f\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(f\ (\lambda x.((g\ g)\ x))))\ x))\ K) \\
\rightarrow_{\mathsf{v}} & \lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x) \\
\rightarrow_{\mathsf{v}} & \lambda x.((K\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\
\rightarrow_{\mathsf{v}} & \lambda x.(((\lambda g x.L)\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\
\rightarrow_{\mathsf{v}} & \lambda x.((\lambda x.L[g \leftarrow (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x))])\ x) \\
\rightarrow_{\mathsf{v}} & \lambda x.L[g \leftarrow (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))] \\
{}_{\mathsf{v}}\leftarrow & ((\lambda g x.L)\ (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))) \\
= & (K\ (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))) \\
{}_{\mathsf{v}}\leftarrow & (K\ (\mathsf{Y}_{\mathsf{V}}\ K))
\end{array}$$

The proof of Theorem 4.1 looks overly complicated because arguments to procedures must be values for β_{v} to apply. Thus instead of calculating

$$\begin{array}{ll}
& \lambda x.((K\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\
= & \lambda x.((K\ (\mathsf{Y}_{\mathsf{V}}\ K))\ x) \\
= & \lambda x.((\lambda g x.L)\ (\mathsf{Y}_{\mathsf{V}}\ K))\ x)
\end{array}$$

we need to carry around the *value* computed by $(\mathsf{Y}_{\mathsf{V}}\ K)$. To avoid this complication in the future, we show that an argument that is provably equal to a value (but is not necessarily a value yet) can already be used as if it were a value for the purposes of $=_{\mathbf{v}}$.

Theorem 4.2: If $M =_{\mathbf{v}} V$, then for all $(\lambda X.N)$, $((\lambda X.N)\ M) =_{\mathbf{v}} N[X \leftarrow M]$.

Proof for Theorem 4.2: The initial portion of the proof is a simple calculation:

$$((\lambda X.N)\ M) =_{\mathbf{v}} ((\lambda X.N)\ V) =_{\mathbf{v}} N[X \leftarrow V] =_{\mathbf{v}} N[X \leftarrow M]$$

The last step in this calculation, however, needs a separate proof by induction on N, showing that $N[X \leftarrow M] =_{\mathbf{v}} N[X \leftarrow L]$ if $M =_{\mathbf{v}} L$:

- Base cases:
  - **Case** $N = b$
    $b[X \leftarrow M] = b =_{\mathbf{v}} b = b[X \leftarrow L]$.
  - **Case** $N = Y$
    If $Y = X$, then $X[X \leftarrow M] = M =_{\mathbf{v}} L = X[X \leftarrow L]$. Otherwise, $Y[X \leftarrow M] = Y =_{\mathbf{v}} Y = Y[X \leftarrow L]$.
- Inductive cases:
  - **Case** $N = (\lambda Y.N')$
    If $Y = X$, then $N[M \leftarrow X] = N =_{\mathbf{v}} N = N[L \leftarrow X]$. Otherwise, by induction, $N'[X \leftarrow M] =_{\mathbf{v}} N'[X \leftarrow L]$. Then,
    $$\begin{array}{ll}
    & (\lambda Y.N')[X \leftarrow M] \\
    = & (\lambda Y.N'[X \leftarrow M]) \\
    =_{\mathbf{v}} & (\lambda Y.N'[X \leftarrow L]) \\
    = & (\lambda Y.N')[X \leftarrow L]
    \end{array}$$
  - **Case** $N = (N_1\ N_2)$
    By induction, $N_i[X \leftarrow M] =_{\mathbf{v}} N_i[X \leftarrow L]$ for $i \in [1, 2]$. Then,
    $$\begin{array}{ll}
    & (N_1\ N_2)[X \leftarrow M] \\
    = & (N_1[X \leftarrow M]\ N_2[X \leftarrow M]) \\
    =_{\mathbf{v}} & (N_1[X \leftarrow L]\ N_2[X \leftarrow L]) \\
    = & (N_1 N_2)[X \leftarrow L]
    \end{array}$$

– **Case** $N = (o^n\ N_1 \ldots\ N_n)$
Analogous to the previous case.

4.4 Evaluation

For defining an evaluator, functions as values pose just as much of a problem for ISWIM as for the λ-calculus. For example, the expression

$$(\lambda x.x)\ (\lambda y.(\lambda x.x)\ \ulcorner 0 \urcorner)$$

is clearly equivalent to both

$$\lambda y.(\lambda x.x)\ \ulcorner 0 \urcorner$$

and

$$\lambda y.\ulcorner 0 \urcorner$$

Which one should we pick as *the* result of an evaluator? While all of these results hopefully represent one and the same function, it is also clear that there are infinitely many ways to represent a function. We therefore adopt a solution that essentially all practical programming systems implement: when a program reduces to a function value, the $eval_{\mathbf{v}}$ function merely returns a token indicating that the value is a function.

We define A to be the set of results for evaluating ISWIM functions:

$$A = b \cup \{\texttt{function}\}$$

The partial function $eval_{\mathbf{v}}$ is defined as follows:

$$eval_{\mathbf{v}}(M) = \begin{cases} b & \text{if } M =_{\mathbf{v}} b \\ \texttt{function} & \text{if } M =_{\mathbf{v}} \lambda X.N \end{cases}$$

If $eval_{\mathbf{v}}(M)$ does not exist, we say that M **diverges**. For example, Ω diverges.

▷ **Exercise 4.2.** Suppose that we try to strengthen the evaluation function as follows:

$$eval_1(M) = \begin{cases} b & \text{if } M =_{\mathbf{v}} b \\ \texttt{function1} & \text{if } M =_{\mathbf{v}} \lambda X.N \qquad N \neq \lambda Y.N' \text{ for any } Y, N' \\ \texttt{function+} & \text{if } M =_{\mathbf{v}} \lambda X.\lambda Y.N \end{cases}$$

Is $eval_1$ a function? If so, prove it. If not, provide a counter-example.

4.5 Consistency

The definition of ISWIM's evaluator relies on an equational calculus for ISWIM. Although the calculus is based on intuitive arguments and is almost as easy to use as a system of arithmetic, it is far from obvious whether the evaluator is a function that always returns a unique answer for a program. But for a programmer who needs a deterministic, reliable programming language, this fact is crucial and needs to be established formally. In short, we need to modify the consistency theorem of $eval_{\mathbf{r}}$ for ISWIM's $eval_{\mathbf{v}}$. The proof of this theorem turns out to be far more complex than the one for $eval_{\mathbf{r}}$, though it follows the same outline.

We start with the basic theorem asserting the Church-Rosser property for ISWIM's calculus.

Theorem 4.3 [Consistency of $eval_{\mathbf{v}}$]: The relation $eval_{\mathbf{v}}$ is a partial function.

Proof for Theorem 4.3: Assume that the Church-Rosser Property holds: if $M =_v N$ then there exists an expression L such that $M \twoheadrightarrow_{\mathbf{v}} L$ and $N \twoheadrightarrow_{\mathbf{v}} L$.

Let $eval_{\mathbf{v}}(M) = A_1$ and $eval_{\mathbf{v}}(M) = A_2$ for answers A_1 and A_2. We need to show that $A_1 = A_2$. Based on the definition of ISWIM answers, we distinguish two cases:

- **Case** $A_1 \in b$, $A_2 \in b$
  It follows from Church-Rosser that two basic constants are provably equal if and only if they are identical, since neither is reducible.
- **Case** $A_1 = \texttt{function}$, $A_2 \in b$
  By the definition of $eval_{\mathbf{v}}$, $M =_{\mathbf{v}} A_2$ and $M =_{\mathbf{v}} \lambda x.N$ for some N, so $A_2 =_{\mathbf{v}} \lambda x.N$. Again by Church-Rosser and the irreducibility of constants, $\lambda x.N \twoheadrightarrow_{\mathbf{v}} A_2$. But by definition of the reduction relation $\twoheadrightarrow_{\mathbf{v}}$, $\lambda x.N \twoheadrightarrow_{\mathbf{v}} K$ implies that $K = \lambda x.K'$. Thus it is impossible that $\lambda x.N$ reduces to A_2, which means that the assumptions of the case are contradictory.
- **Case** $A_1 \in b$, $A_2 = \texttt{function}$
  Analogous to the previous case.
- **Case** $A_1 = \texttt{function}$, $A_2 = \texttt{function}$
  Then $A_1 = A_2$.

These are all possible cases, and hence, it is always true that $A_1 = A_2$, so $eval_{\mathbf{v}}$ is a function.

By the preceding proof, we have reduced the consistency property to the Church-Rosser property for ISWIM.

Theorem 4.4 [Church-Rosser for ISWIM]: If $M =_{\mathbf{v}} N$, then there exists an expression L such that $M \twoheadrightarrow_{\mathbf{v}} L$ and $N \twoheadrightarrow_{\mathbf{v}} L$.

Proof for Theorem 4.4: The proof is essentially a replica of the proof for $=_{\mathbf{r}}$ (Theorem 2.2). It assumes a diamond property for the reduction relation $\twoheadrightarrow_{\mathbf{v}}$.

After disposing of the Consistency and Church-Rosser theorems, we have arrived at the core of the problem. Now we need to show the diamond property for ISWIM.

Theorem 4.5 [Diamond Property for $\twoheadrightarrow_{\mathbf{v}}$]: If $L \twoheadrightarrow_{\mathbf{v}} M$ and $L \twoheadrightarrow_{\mathbf{v}} N$, then there exists an expression K such that $M \twoheadrightarrow_{\mathbf{v}} K$ and $N \twoheadrightarrow_{\mathbf{v}} K$.

For $\twoheadrightarrow_{\mathbf{r}}$, a diamond-like property holds for the single-step relation $\rightarrow_{\mathbf{r}}$, from which the diamond property for the transitive closure easily follows. But, analogous to the example in the previous chapter, it is clear that the diamond property *cannot* hold for $\rightarrow_{\mathbf{v}}$. The β_{v} reduction can copy redexes in the argument of a function, which prevents the existence of the common contractum in certain diamonds. For an example, consider the expression

$$\underline{((\lambda x.(x\ x))\ (\lambda y.\underline{((\lambda x.x)\ (\lambda x.x))}))}$$

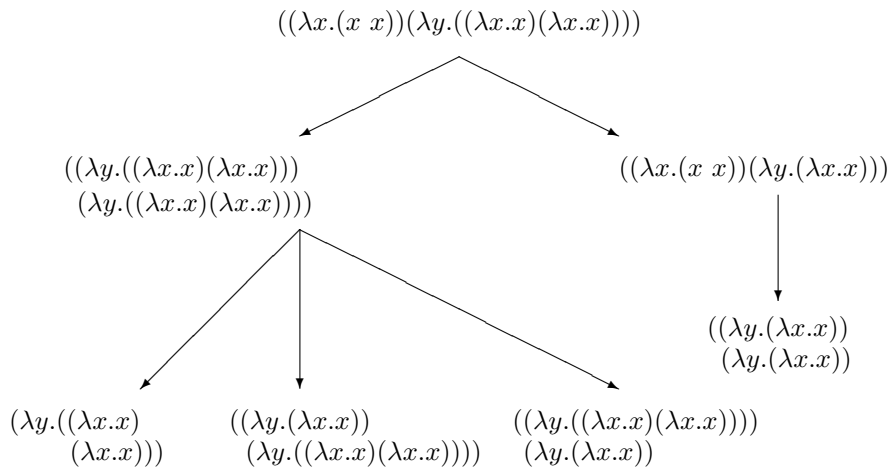
which contains the two overlapping and underlined β_v-redexes. By one or the other of one of them, we get the expressions

$$\underline{((\lambda x.(x\ x))\ (\lambda y.(\lambda x.x)))}$$

and

$$((\lambda y.\underline{((\lambda x.x)\ (\lambda x.x))})\ (\lambda y.\underline{((\lambda x.x)\ (\lambda x.x))}))$$

Both expressions contain redexes, but as the following diagram shows, the one-step reductions starting from these expressions do not lead to a common expression.



Hence, the one-step relation $\rightarrow_{\mathbf{v}}$ does not satisfy the diamond property.

Since the problem of the one-step relation based on $\mathbf{v}$ is caused by reductions that duplicate redexes, we consider an extension $\hookrightarrow_{\mathbf{v}}$ of the one-step reduction $\rightarrow_{\mathbf{v}}$ that contracts several non-overlapping redexes in parallel. If this extension satisfies the diamond property, and if its transitive–reflexive closure is the same as that of $\rightarrow_{\mathbf{v}}$, then we can prove the diamond property for the $\twoheadrightarrow_{\mathbf{v}}$ reduction.

The parallel extension $\hookrightarrow_{\mathbf{v}}$ is defined as follows:

M	$\hookrightarrow_{\mathbf{v}}$	N	if $M =_\alpha N$
$(o^n\ b_1\ \ldots b_n)$	$\hookrightarrow_{\mathbf{v}}$	$\delta(o^n, b_1, \ldots b_n)$	if $\delta(o^n, b_1, \ldots b_n)$ is defined
$((\lambda X.M)\ N)$	$\hookrightarrow_{\mathbf{v}}$	$M'[X \leftarrow V]$	if $M \hookrightarrow_{\mathbf{v}} M'$ and $N \hookrightarrow_{\mathbf{v}} V$
$(M\ N)$	$\hookrightarrow_{\mathbf{v}}$	$(M'\ N')$	if $M \hookrightarrow_{\mathbf{v}} M'$ and $N \hookrightarrow_{\mathbf{v}} N'$
$(\lambda X.M)$	$\hookrightarrow_{\mathbf{v}}$	$(\lambda x.M')$	if $M \hookrightarrow_{\mathbf{v}} M'$
$(o^n\ M_1\ \ldots\ M_n)$	$\hookrightarrow_{\mathbf{v}}$	$(o^n\ M'_1\ \ldots\ M'_n)$	if $M_i \hookrightarrow_{\mathbf{v}} M'_i, i \in [1, n]$

This relation satisfies the diamond property.

Theorem 4.6 [Diamond Property for $\hookrightarrow_{\mathbf{v}}$]: If $L \hookrightarrow_{\mathbf{v}} M$ and $L \hookrightarrow_{\mathbf{v}} N$, then there exists an expression K such that $M \hookrightarrow_{\mathbf{v}} K$ and $N \hookrightarrow_{\mathbf{v}} K$.

Proof for Theorem 4.6: By induction on the structure of the proof tree of $L \hookrightarrow_{\mathbf{v}} M$.

- Base cases:
  - **Case** $L =_\alpha M$
    Set $K = N$, and the claim holds.
  - **Case** $L = (o^n\ b_1 \ldots\ b_n), M = \delta(o^n, b_1, \ldots b_n)$
    Since δ is a function there is no other possible reduction for L; set $K = N = M$.
- Inductive cases:
  - **Case** $L = ((\lambda X.M_0)\ L_1)$**,** $M = M'_0[X \leftarrow V]$**,** $M_0 \hookrightarrow_{\mathbf{v}} M'_0$**,** $L_1 \hookrightarrow_{\mathbf{v}} V$
    There are two possiblities for the reduction path $L \hookrightarrow_{\mathbf{v}} N$. Either the two parts of the application merely reduce separately, or the application is also reduced via $\beta_{\mathbf{v}}$:

* **Case** $L = ((\lambda X.M_0)\ L_1)$, $N = ((\lambda X.M_0'')\ L_1'')$, $M_0 \hookrightarrow_{\mathbf{v}} M_0''$, $L_1 \hookrightarrow_{\mathbf{v}} L_1''$
  In this case, M_0, M_0', and M_0'' satisfy the antecedent of the inductive hypothesis, and so do L_1, V, and L_1''. Hence, there are expressions K_0 and K_1 that complete the upper half of these two diamons. If we also knew that substitution and parallel reduction commute, we could conclude that $K = K_0[X \leftarrow K_1]$, since $((\lambda X.M_0'')\ L_1'') \hookrightarrow_{\mathbf{v}} K_0[X \leftarrow K_1]$ and $M_0'[X \leftarrow V] \hookrightarrow_{\mathbf{v}} K'[X \leftarrow K_1]$.
* **Case** $L = ((\lambda X.M_0)\ L_1)$, $N = M_0''[X \leftarrow V']$, $M_0 \hookrightarrow_{\mathbf{v}} M_0''$, $L_1 \hookrightarrow_{\mathbf{v}} V'$
  As in the first subcase, M_0, M_0', and M_0'' and L_1, V, and V'' determine upper halves of diamonds. By the induction hypothesis, each yields two lower halves of diamonds and related terms K_0 and K_1. And again, if substitution and parallel reduction commute, setting $K = K_0[X \leftarrow V]$ concludes the subcase.

  To complete this case, we are obliged to prove the commutation property. We postpone this proof, since other cases also depend on it; see Lemma 4.7.

– **Case** $L = (L_1\ L_2)$, $M = (M_1\ M_2)$, $L_i \hookrightarrow_{\mathbf{v}} M_i$
  Analogous to the previous case.

– **Case** $L = (o^n\ L_1 \ldots L_n)$, $M = (o^n\ M_1 \ldots M_n)$, $L_i \hookrightarrow_{\mathbf{v}} M_i$
  There are two cases:
  * **Case** $L_1 = b_1, \ldots\ L_n = b_n$, $N = \delta(o^n, b_1, \ldots b_n)$
    Then $M = L$ and $K = N$.
  * **Case** $N = (o^n\ N_1 \ldots N_n), L_i \hookrightarrow_{\mathbf{v}} N_i$
    An n-fold use of the inductive hypothesis yields the conclusion.

– **Case** $L = (\lambda X.L_0)$, $M = (\lambda X.M_0)$, $L_0 \hookrightarrow_{\mathbf{v}} M_0$
  The only possibility for N is $N = (\lambda X.N_0)$ with $L_0 \hookrightarrow_{\mathbf{v}} N_0$. The induction hypothesis applies to L_0, M_0, and N_0 and yields an expression K_0; $K = (\lambda X.K_0)$.

To finish the preceding proof, we need to show that substitutions in expressions that are related by parallel reduction do not affect the reduction step.

Lemma 4.7: If $M \hookrightarrow_{\mathbf{v}} M'$ and $N \hookrightarrow_{\mathbf{v}} N'$ then $M[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M'[X \leftarrow N']$.

Proof for Lemma 4.7: By induction on the proof tree of $M \hookrightarrow_{\mathbf{v}} M'$:

- Base cases:
  – **Case** $M =_\alpha M'$
    In this special case, the claim essentially says that if $N \hookrightarrow_{\mathbf{v}} N'$, then $M[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M[X \leftarrow N']$. The proof of this specialized claim is an induction on the structure of M. We leave this proof as an exercise.
  – **Case** $M = (o^n\ b_1 \ldots\ b_n)$, $M' = \delta(o^n, b_1, \ldots, b_n)$
    M and M' are closed, so $M[X \leftarrow N] = M$ and $M'[X \leftarrow N] = M'$, and we know already that $M \hookrightarrow_{\mathbf{v}} M'$.
- Inductive cases:
  – **Case** $M = ((\lambda X_0.M_0)\ L_0)$, $M' = M_0'[X_0 \leftarrow V]$, $M_0 \hookrightarrow_{\mathbf{v}} M_0'$, $L_0 \hookrightarrow_{\mathbf{v}} V$
    By induction, $M_0[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M_0'[X \leftarrow N']$ and $L_0[X \leftarrow N] \hookrightarrow_{\mathbf{v}} V[X \leftarrow N']$. Thus:

$$\begin{array}{rcll} M[X \leftarrow N] & = & ((\lambda X_0.M_0[X \leftarrow N])\ L_0[X \leftarrow N]) & \\ & \hookrightarrow_{\mathbf{v}} & M_0'[X \leftarrow N'][X_0 \leftarrow V[X \leftarrow N']] & \\ & = & M_0'[X_0 \leftarrow V][X \leftarrow N'] & (\dagger) \\ & =_\alpha & M'[X \leftarrow N'] & \end{array}$$

Equation (†) is a basic property of the substitution function. We leave this step as an exercise.

– **Case** $M = (M_1\ M_2)$, $M' = (M'_1\ M'_2)$, $M_i \hookrightarrow_{\mathbf{v}} M'_i$
By induction, $M_i[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M'_i[X \leftarrow N']$ for both $i = 1$ and $i = 2$. Thus,

$$\begin{array}{rcl} M[X \leftarrow N] & = & (M_1[X \leftarrow N]\ M_2[X \leftarrow N]) \\ & \hookrightarrow_{\mathbf{v}} & (M'_1[X \leftarrow N]\ M'_2[X \leftarrow N]) \\ & \hookrightarrow_{\mathbf{v}} & (M'_1[X \leftarrow N']\ M'_2[X \leftarrow N']) \\ & = & (M'_1\ M'_2)[X \leftarrow N'] \\ & = & M'[X \leftarrow N'] \end{array}$$

So the claim holds.

– **Case** $M = (o^n\ M_1 \ldots\ M_n)$, $M' = (o^n\ M'_1 \ldots\ M'_n)$, $M_i \hookrightarrow_{\mathbf{v}} M'_i$
Analogous to the previous case.

– **Case** $M = (\lambda X.M_0)$, $M' = (\lambda X.M'_0)\ M_0 \hookrightarrow_{\mathbf{v}} M'_0$
Analogous to the previous case.

▷ **Exercise 4.3.** Prove that if $N \hookrightarrow_{\mathbf{v}} N'$, then $M[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M[X \leftarrow N']$.

▷ **Exercise 4.4.** Prove that if $X \notin \mathcal{FV}(L)$ then

$$K[X \leftarrow L][X' \leftarrow M[X \leftarrow L]] =_\alpha K[X' \leftarrow M][X \leftarrow L]$$

▷ **Exercise 4.5.** Prove that the transitive–reflexive closure of the parallel reduction $\hookrightarrow_{\mathbf{v}}$ is the same as $\twoheadrightarrow_{\mathbf{v}}$. This fact, along with Theorem 4.6, supports the proof of Theorem 4.5.

4.6 Observational Equivalence

Besides program evaluation, the transformation of programs plays an important role in the practice of programming and in the implementation of programming languages. For example, if M is a procedure, and if we think that N is a procedure that can perform the same computation as M but faster, then we would like to know whether M is really interchangeable with N. For one special instance of this problem, ISWIM clearly provides a solution: if both expressions are complete programs and we know that $M =_{\mathbf{v}} N$, then $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$ and we know that we can replace M by N.

In general, however, M and N are sub-expressions of some program, and we may not be able to evaluate them independently. Hence, what we really need is a general relation that explains when expressions are "equivalent in functionality".

To understand the notion of "equivalent in functionality", we must recall that the user of a program can only *observe* the output, and is primarily interested in the output. The programmer usually does not know the text of the program or any aspects of the text. In other words, such an observer treats programs as black boxes that produce some value in an unknown manner. It is consequently natural to say that the effect of an expression is the effect it has when it is a part of a program. Going even further, the comparison of two open expressions reduces to the question whether there is a way to tell apart the effects of the expressions. Or, put positively, if two expressions have the same effect, they are interchangeable from the standpoint of an external observer.

Before we continue, we need a notion of expressions contexts. The following grammar C defines a set of near-expressions. They are not quite expressions, because each member of C

contains a **hole**, written [ ], in the place of one subexpression.

$$
\begin{array}{rcl}
C & = & [\,] \\
 & | & (\lambda X.C) \\
 & | & (C\ M) \\
 & | & (M\ C) \\
 & | & (o^n\ M\ \ldots\ M\ C\ M\ \ldots\ M)
\end{array}
$$

$C[M]$ means "replace the [ ] in C with M"

For example, $(\lambda x.[\,])$ and $(+\ 1\ ((\lambda x.\ulcorner 0\urcorner)\ [\,]))$ are members of C, and

$$
\begin{array}{lcl}
(\lambda x.[\,])[(\lambda y.x)] & = & (\lambda x.(\lambda y.x)) \\
(+\ 1\ ((\lambda x.\ulcorner 0\urcorner)\ [\,]))[(z\ \ulcorner 12\urcorner)] & = & (+\ 1\ ((\lambda x.\ulcorner 0\urcorner)\ (z\ \ulcorner 12\urcorner)))
\end{array}
$$

Unlike substitution, filling a context hole can capture variables.

With this definition of context, we can express the notion of observational equivalence. Two expressions M and N are **observationally equivalent**, written $M \simeq_{\mathbf{v}} N$, if and only if they are indistinguishable in all contexts C.

$$M \simeq_{\mathbf{v}} N \quad \text{if} \quad eval_{\mathbf{v}}(C[M]) = eval_{\mathbf{v}}(C[N]) \quad \text{for all } C$$

The definition implies that if $M \simeq_{\mathbf{v}} N$, then $eval_{\mathbf{v}}(C[M])$ and $eval_{\mathbf{v}}(C[N])$ must be either both defined or both undefined for any given C.

Observational equivalence obviously extends program equivalence in terms of $eval_{\mathbf{v}}$; if programs M and N are observationally equivalent, then they are programs in the empty context, so $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$. But this is also the least we should expect from a theory of expressions with equivalent functionality. As the name already indicates, observational equivalence is an equivalence relation. Finally, if two expressions are observationally equivalent, then embedding the expressions in contexts should yield observationally equivalent expressions. After all, the added pieces are identical and should not affect the possible effects of the expressions on the output of a program. In short, observational equivalence is a **congruence relation**, i.e., its closure over contexts is also an equivalence relation.

Theorem 4.8: For any expressions M, N, and L,

1. $M \simeq_{\mathbf{v}} M$;
2. if $L \simeq_{\mathbf{v}} M$ and $M \simeq_{\mathbf{v}} N$, then $L \simeq_{\mathbf{v}} N$;
3. if $L \simeq_{\mathbf{v}} M$, then $M \simeq_{\mathbf{v}} L$; and
4. if $M \simeq_{\mathbf{v}} N$, then $C[M] \simeq_{\mathbf{v}} C[N]$ for all contexts C.

Proof for Theorem 4.8: Points 1 through 3 are trivial. As for point 4, assume $M \simeq_{\mathbf{v}} N$ and let C' be an arbitrary context. Then, $C'[C]$ is a context for M and N. By assumption $M \simeq_{\mathbf{v}} N$ and therefore,

$$eval_{\mathbf{v}}(C'[C[M]]) = eval_{\mathbf{v}}(C'[C[N]])$$

which is what we had to prove.

It turns out that observational equivalence is the largest congruence relation on expressions that satistifies our intutitive criteria.

Theorem 4.9: Let $\mathbf{R}$ be a congruence relation such that $M\,\mathbf{R}\,N$ for expression M and N implies $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$. If $M\,\mathbf{R}\,N$, then $M \simeq_{\mathbf{v}} N$.

Proof for Theorem 4.9: We assume $M \not\simeq_{\mathbf{v}} N$ and show $M \not\mathbf{R} N$. By assumption there exists a separating context C. That is, $eval_{\mathbf{v}}(C[M]) \neq eval_{\mathbf{v}}(C[N])$. Therefore, $C[M] \not\mathbf{R} C[N]$. Since $\mathbf{R}$ is supposed to be a congruence relation, we cannot have $M \equiv N$, which is a contradiction.

The proposition shows that observational equivalence is the most basic, the most fundamental congruence relation on expressions. All other relations only approximate the accuracy with which observational equivalence identifies expressions. It also follows from the proposition that ISWIM is sound with respect to observational equivalence. Unfortunately, it is also incomplete; the calculus cannot *prove* all observational equivalence relations.

Theorem 4.10 [Soundness, Incompleteness]: If we can show $M =_{\mathbf{v}} N$, then $M \simeq_{\mathbf{v}} N$. But $M \simeq_{\mathbf{v}} N$ does not imply that we can prove $M =_{\mathbf{v}} N$.

Proof for Theorem 4.10: By definition, $=_{\mathbf{v}}$ is a congruence relation. Moreover, $eval_{\mathbf{v}}$ is defined based on $=_{\mathbf{v}}$ such that if $M =_{\mathbf{v}} N$, then $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$.

For the inverse direction, we give a counter-example and sketch the proof that it is a counter-example. Consider the expressions $(\Omega\ (\lambda x.x))$ and Ω, where $\Omega = ((\lambda x.(x\ x)))\ (\lambda x.(x\ x)))$. Both terms diverge and are observationally equivalent. But both reduce to themselves only, and therefore cannot be provably $=_{\mathbf{v}}$.

To complete the proof of the Incompleteness Theorem, we still lack some tools. We will return to the proof in the next chapter, when we have developed some basic more knowledge about ISWIM.

▷ **Exercise 4.6.** Consider the following evaluation function $eval_0$, plus its associated observational equivalence relation $\simeq_0$:

$$eval_0(M) = \texttt{value} \quad \text{if} \quad M =_{\mathbf{v}} V \quad \text{for some } V$$

$$M \simeq_0 N \qquad \text{if} \quad eval_0(C[M]) = eval_0(C[N]) \quad \text{for all } C$$

Does $M \simeq_0 N$ imply anything about $M \simeq_{\mathbf{v}} N$? Sketch an argument for your answer.

4.7 History

In a series of papers in the mid-1960's [5], Landin expounded two important observations about programming languages. First, he argued that all programming languages share a basic set of facilities for specifying computation but differ in their choice of data and data primitives. The set of common facilities contains names, procedures, applications, exception mechanisms, mutable data structures, and possibly other forms of non-local control. Languages for numerical applications typically include several forms of numerical constants and large sets of numerical primitives, while those for string manipulation typically offer efficient string matching and manipulation primitives.

Second, he urged that programmers and implementors alike should think of a programming language as an advanced, symbolic form of arithmetic and algebra. Since all of us are used to calculating with numbers, booleans, and even more complex data structures from our days in kindergarten and high school, it should be easy to calculate with programs, too. Program evaluation, many forms of program editing, program transformations, and optimizations are just different, more elaborate forms of calculation. Instead of simple arithmetic expressions, such calculations deal with programs and pieces of programs.

Landin defined the programming language ISWIM. The basis of his design was Church's λ-calculus — a natural starting point, given Landin's insight on the central role of procedures

as a facility common to all languages. However, to support basic data and related primitives as well as assignments and control constructs, Landin extended the λ-calculus with appropriate constructions. He specified the semantics of the extended language with an abstract machine because he did not know how to extend the equational theory of the λ-calculus to a theory for the complete programming language. Indeed, it turned out that the λ-calculus does not even explain the semantics of the pure functional sub-language because ISWIM always evaluates the arguments to a procedure. Thus, Landin did not accomplish what he had set out to do, namely, to define the idealized core of all programming languages and an equational calculus that defines its semantics.

Starting with Plotkin's work on the relationship of abstract machines to equational calculi in the mid-1970's [8], the gap in Landin's work has been filled by a number of researchers, including Felleisen, Mason, Talcott, and their collaborators. Plotkin's work covered the basic functional sub-language of ISWIM, which requires the definition of a call-by-value variant of the λ-calculus. Felleisen and his co-workers extended the equational theory with axioms that account for several distinct kinds of imperative language facilities. Mason and Talcott investigated the use of equational theories for full ISWIM-like languages as a tool for program verification and transformational programming.

Although ISWIM did not become an actively used language, the philosophy of ISWIM lives on in modern programming languages, most notably, Scheme and ML, and its approach to language analysis and design applies to basically all programming languages. And one goal of this book is to illustrate the design, analysis and use of equational theories like the λ-calculus in the context of programming language design and analysis.

Chapter 5: Standard Reduction

The definition of the ISWIM evaluator via an equational proof system is elegant and flexible. A programmer can liberally apply the rules of the equational system in any order and at any place in the program, pursuing any strategy for such a calculation that seems profitable, then switching as soon as a different one appears to yield better results. For example, in cases involving the fixed point operator, the right-to-left direction of the $\beta_{\mathbf{v}}$-axiom is useful, whereas in other cases the opposite direction is preferable. However, this flexibility of the calculus is clearly not a good basis for implementing the evaluator $eval_{\mathbf{v}}$ as a computer program. Since we know from the Church-Rosser Theorem that a program is *equal* to a value precisely when it *reduces* to a value, we already know that we can rely on reductions in the search for a result. But even the restriction to reductions is not completely satisfactory because there are still too many redexes to choose from. An implementation would clearly benefit from knowing that picking a certain reduction step out of the set of possible steps guaranteed progress; otherwise, it would have to search too many different reduction paths.

The solution is to find a strategy that reduces a program to an answer if an answer is possible. In other words, given a program, it is possible to pick a redex according to a fixed strategy such that the reduction of the redex brings the evaluation one step closer to the result. The strategy is known as Curry-Feys Standardization. The strategy and its correctness theorem, the Curry-Feys Standard Reduction Theorem, are the subject of the first section.

The standard reduction strategy essentially defines a **textual machine** for ISWIM. In contrast to a real computer, the states of a textual machine are programs, and the machine instructions transform programs into programs. Nevertheless, the textual machine provides the foundation for a first implementation of the evaluator. As we will see in the next chapter, a systematic elimination of its inefficiencies and overlaps leads to machines with more efficient representations of intermediate states, and every step in this development is easy to justify and to formalize.

Another use of the textual machine concerns the analysis of the behavior of programs. For example, all program evaluations either end in a value, continue for ever, or reach a stuck state due to the misapplication of a primitive. Similarly, if a closed sub-expression plays a role during the evaluation, it must become the "current" instruction at some point during the evaluation. In the latter sections of this chapter, we prove theorems that formalize such properties.

5.1 Standard Reductions

Even after restricting our attention to reductions alone, it is not possible to build a naive evaluator, because there are still too many choices to make. Consider the following program:

$$(((\lambda xy.y)\ (\lambda x.\underline{((\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z))}))\ \ulcorner 7 \urcorner).$$

Both underlined sub-expressions are redexes. By reducing the larger one, the evaluation process clearly makes progress. After the reduction the new program is

$$((\lambda y.y)\ \ulcorner 7 \urcorner),$$

which is one reduction step away from the result, $\ulcorner 7 \urcorner$. However, if the evaluator were to choose the inner redex and were to continue in this way, it would not find the result of the program:

$$\begin{array}{ll} & ((\lambda xy.y)\ (\lambda x.((\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z)))\ \ulcorner 7 \urcorner) \\ \rightarrow_{\mathbf{v}} & ((\lambda xy.y)\ (\lambda x.((\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z))))\ \ulcorner 7 \urcorner) \\ \rightarrow_{\mathbf{v}} & ((\lambda xy.y)\ (\lambda x.((\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z)\ (\lambda z.z\ z\ z))))\ \ulcorner 7 \urcorner) \\ \rightarrow_{\mathbf{v}} & \ldots \end{array}$$

The problem with the second strategy is obvious: it does not find the result of the program because it reduces sub-expressions inside of λ-abstractions, and leaves the outer applications of the program intact. This observation also hints at a simple solution. A good evaluation strategy based on reductions should only pick redexes that are outside of abstractions. We make this more precise with a set of two rules:

1. If the expression is an application and all components are values, then, if the expression is a redex, it is the next redex to be reduced; if not, the program *cannot* have a value.

2. If the expression is an application but does not only consist of values, then pick one of the non-values and search for a potential redex in it.

The second rule is ambiguous because it permits distinct sub-expressions of a program to be candidates for further reduction. Since a deterministic strategy should pick a unique sub-expression as a candidate for further reduction, we *arbitrarily* choose to search for redexes in such an application from left to right:

2′. If the program is an application such that at least one of its sub-expressions is not a value, pick the leftmost non-value and search for a redex in there.

By following the above algorithm, we divide a program into an application consisting of values and a context. The shape of such a context is determined by our rules: it is either a hole, or it is an application and the sub-context is to the right of all values. Since these contexts play a special role in our subsequent investigations, we provide a formal definition for the set of **evaluation contexts** E.

$$
\begin{array}{rcl}
E & = & [\,] \\
 & | & (V\ E) \\
 & | & (E\ M) \\
 & | & (o^n\ V \ldots\ V\ E\ M \ldots\ M)
\end{array}
$$

To verify that our informal strategy picks a unique sub-expression from a program as a potential redex, we show that every program is either a value, or it is a unique evaluation context filled with an application that solely consists of values.

> **Theorem 5.1 [Unique Evaluation Contexts]**: For all M, either $M = V$, or there exists a unique evaluation context E such that $M = E[(V_1\ V_2)]$ or $M = E[(o^n\ V_1 \ldots\ V_n)]$
>
> **Proof for Theorem 5.1**: By induction on the structure of M. The base cases hold because they are values. Similarly, the $(\lambda X.N)$ case holds because it is a value. Otherwise, M is an operator application or function application.
>
> Assume that $M = (o^n\ N_1 \ldots\ N_n)$. If all $N_i \in V$, we are done. Otherwise there is a leftmost argument expression N_i (where $1 \leq i \leq n$) that is not a value. By induction, N_i can be partitioned into a unique evaluation context E' and some application L. Then $E = (o^n\ N_1 \ldots\ N_{i-1}\ E'\ N_{i+1} \ldots\ N_n)$ is an evaluation context, and $M = E[L]$. The context is unique since i is the minimal index such that $N_1, \ldots N_{i-1} \in V$ and E' was unique by induction.
>
> The case for function applications is analogous.

It follows from Theorem 5.1 that if $M = E[L]$ where L is an application consisting of values, then L is uniquely determined. If L is moreover a β_v or a δ redex, it is *the* redex that, according to our intuitive rules 1 and 2′, must be reduced. Reducing it produces a new program, which can be decomposed again. In short, to get an answer for a program: if it is a value, there

is nothing to do, otherwise decompose the program into an evaluation context and a redex, reduce the redex, and start over. To formalize this idea, we introduce a **standard reduction relation** $\longmapsto_{\mathbf{v}}$.

$$E[M] \longmapsto_{\mathbf{v}} E[M'] \quad \text{if} \quad M \,\mathbf{v}\, M'$$

We pronounce $M \longmapsto_{\mathbf{v}} N$ as "M standard-reduces to N." As usual, $\longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}}$ is the transitive–reflexive closure of $\longmapsto_{\mathbf{v}}$.

By Theorem 5.1, the standard reduction relation is a well-defined partial function. Based on it, our informal evaluation strategy can now easily be cast in terms of standard reduction steps. That is, a program can be evaluated by standard-reducing it to a value. But we need to verify formally that $\longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}}$ arrives at the same result as $\twoheadrightarrow_{\mathbf{v}}$.

> **Theorem 5.2 [Standard Reduction]**: $M \twoheadrightarrow_{\mathbf{v}} U$ if and only if $M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} V$ for some value V and $V \twoheadrightarrow_{\mathbf{v}} U$.
>
> In particular, if $U \in b$ then $M \twoheadrightarrow_{\mathbf{v}} U$ if and only if $M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} U$.

The separate values V and U are needed in the theorem's statement because a $\twoheadrightarrow_{\mathbf{v}}$ reduction might reduce expressions within a λ-abstraction. From the perspective of an evaluator, such reductions are useless, because they do not change the result `function`. We defer the proof of the theorem until the next section.

The alternative $eval^{\mathsf{s}}_{\mathbf{v}}$ function maps a program to its value using the standard reduction relation $\longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}}$, if a value exists.

$$eval^{\mathsf{s}}_{\mathbf{v}}(M) = \begin{cases} b & \text{if } M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} b \\ \texttt{function} & \text{if } M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} \lambda X.N \end{cases}$$

It is a simple corollary of Theorem 5.2 that this new evaluator function is the same as the original one.

> **Theorem 5.3**: $eval_{\mathbf{v}} = eval^{\mathsf{s}}_{\mathbf{v}}$.
>
> **Proof for Theorem 5.3**: Assume $eval_{\mathbf{v}}(M) = b$. By Church-Rosser and the fact that constants are normal forms with respect to $\mathbf{v}$, $M \twoheadrightarrow_{\mathbf{v}} b$. By Theorem 5.2, $M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} b$ and thus, $eval^{\mathsf{s}}_{\mathbf{v}}(M) = b$. Conversely, if $eval^{\mathsf{s}}_{\mathbf{v}}(M) = b$, then $M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} b$. Hence, $M =_{\mathbf{v}} b$ and $eval_{\mathbf{v}}(M) = b$.
>
> A similar proof works for the case where $eval_{\mathbf{v}}(M) = \texttt{function} = eval^{\mathsf{s}}_{\mathbf{v}}(M)$.

▷ **Exercise 5.1.** Show the reduction of

$$(+\ ((\lambda x.(\lambda y.y)\ x)\ (-\ \ulcorner 2 \urcorner \ulcorner 1 \urcorner))\ \ulcorner 8 \urcorner)$$

to a value with $\longmapsto_{\mathbf{v}}$. For each step, identify the evaluation context.

▷ **Exercise 5.2.** Suppose that we try to strengthen the evaluation function as follows:

$$eval^{\mathsf{s}}_{1}(M) = \begin{cases} b & \text{if } M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} b \\ \texttt{function1} & \text{if } M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} \lambda X.N \quad N \neq \lambda Y.N' \text{ for any } Y, N' \\ \texttt{function+} & \text{if } M \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{v}} \lambda X.\lambda Y.N \end{cases}$$

Is $eval^{\mathsf{s}}_{1}$ a function? If so, prove it. If not, provide a counter-example.

5.2 Proving the Standard Reduction Theorem

The proof of Theorem 5.2 (Standard Reduction) requires a complex argument. The difficult part of the proof is clearly the left-to-right direction; the right-to-left direction is implied by the fact that $\longmapsto_{\mathbf{v}}$ and $\longmapsto\!\!\!\!\rightarrow_{\mathbf{v}}$ are subsets of $\rightarrow_{\mathbf{v}}$ and $\rightarrow\!\!\!\!\rightarrow_{\mathbf{v}}$, respectively.

A naive proof attempt for the left-to-right direction could proceed by induction on the number of one-step reductions from M to U. Asssume the reduction sequence is as follows:

$$M = M_0 \rightarrow_{\mathbf{v}} M_1 \rightarrow_{\mathbf{v}} \ldots \rightarrow_{\mathbf{v}} M_m = U.$$

For $m = 0$ the claim vacuously holds, but when $m > 0$ the claim is impossible to prove by the simple-minded induction. While the inductive hypothesis says that for $M_1 \rightarrow\!\!\!\!\rightarrow_{\mathbf{v}} U$ there exists a value V such that $M_1 \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$, it is obvious that for many pairs M_0, M_1 it is *not* the case that $M_0 \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} M_1$. For example, the reduction step from M_0 to M_1 could be inside of a λ-abstraction, and the abstraction may be a part of the final answer.

The (mathematical) solution to this problem is to generalize the inductive hypothesis and the theorem. Instead of proving that programs reduce to values if and only if they reduce to values via the transitive closure of the standard reduction *function*, we prove that there is a canonical sequence of reduction steps for all reductions.

To state and prove the general theorem, we introduce the notion of a **standard reduction sequence**, $\mathcal{R}$. A standard reduction sequence generalizes the idea of a sequence of expressions that are related via the standard reduction function. In standard reduction sequences an expression can relate to its successor if a sub-expression in a λ-abstraction standard reduces to another expression. Similarly, standard reduction sequences also permit incomplete reductions, i.e., sequences that may *not* reduce a redex inside of holes of evaluation context for the rest of the sequence.

- $b \subset \mathcal{R}$
- $X \subset \mathcal{R}$
- If $M_1 \diamond \ldots M_m \in \mathcal{R}$, then
  $(\lambda X.M_1) \diamond \ldots (\lambda X.M_m) \in \mathcal{R}$
- If $M_1 \diamond \ldots M_m \in \mathcal{R}$ and $N_1 \diamond \ldots N_n \in \mathcal{R}$, then
  $(M_1\ N_1) \diamond \ldots (M_m\ N_1) \diamond (M_m\ N_2) \diamond \ldots (M_m\ N_n) \in \mathcal{R}$
- If $M_{i,1} \diamond \ldots M_{i,n_i} \in \mathcal{R}$ for $1 \leq i \leq m$ and $n_i \geq 1$, then

$$\begin{array}{l}(o^m\ M_{1,1}\ M_{2,1} \ldots\ M_{m,1}) \diamond \\ (o^m\ M_{1,2}\ M_{2,1} \ldots\ M_{m,1}) \diamond \\ \ldots \\ (o^m\ M_{1,n_1}\ M_{2,1} \ldots\ M_{m,1}) \diamond \\ (o^m\ M_{1,n_1}\ M_{2,2} \ldots\ M_{m,1}) \diamond \\ \ldots \\ (o^m\ M_{1,n_1}\ M_{2,n_2} \ldots\ M_{m,n_m}) \in \mathcal{R}\end{array}$$

- If $M_1 \diamond \ldots M_m \in \mathcal{R}$ and $M_0 \longmapsto_{\mathbf{v}} M_1$, then
  $M_0 \diamond M_1 \diamond \ldots M_m \in \mathcal{R}$

Clearly, the standard reduction sequences consisting of a single term are all ISWIM expressions.

We can now formulate the claim that is provable using a reasonably simple induction argument. The general idea for such a theorem is due to Curry and Feys, who proved it for Church's pure λ-calculus. The theorem for ISWIM is due to Plotkin.

Theorem 5.4: $M \twoheadrightarrow_{\mathbf{v}} N$ if and only if there exists $L_1 \diamond \ldots L_n \in \mathcal{R}$ such that $M = L_1$ and $N = L_n$.

Proof for Theorem 5.4: From right to left, the claim is a consequence of the fact that if K precedes L in a standard reduction sequence, then K reduces to L. This property clearly follows from the definition.

To prove the left-to-right direction, assume $M \twoheadrightarrow_{\mathbf{v}} N$. By the definition of $\twoheadrightarrow_{\mathbf{v}}$, this means that there exist expressions $M_1, \ldots M_m$ such that

$$M \to_{\mathbf{v}} M_1 \to_{\mathbf{v}} \ldots M_m \to_{\mathbf{v}} N$$

The critical idea is now the following: since the parallel reduction relation extends the one-step relation, it is also true that

$$M \hookrightarrow_{\mathbf{v}} M_1 \hookrightarrow_{\mathbf{v}} \ldots M_m \hookrightarrow_{\mathbf{v}} N$$

Moreover, such a sequence of parallel reduction steps can be transformed into a standard reduction sequence, based on the algorithm in the proof of Lemma 5.5, below.

The proof of the main lemma relies on a size function $|\_|$ for derivations of parallel reductions, that is, the argument that two terms are in the parallel reduction relation. The size of such derivations is approximately the number of **v**-redexes that an ordinary reduction between terms would have had to perform.

$$\begin{array}{lcl}
\#(X, M) & = & \text{the number of free } X \text{ in } M \\
|\, M \hookrightarrow_{\mathbf{v}} M \,| & = & 0 \\
|\, (o^n\ b_1 \ldots\ b_n) \hookrightarrow_{\mathbf{v}} \delta(o^n, b_1, \ldots b_n) \,| & = & 1 \\
|\, (\lambda X.M)N \hookrightarrow_{\mathbf{v}} M'[X \leftarrow V] \,| & = & s_1 + \#(X, N) \times s_2 + 1 \\
 & & \text{where } s_1 = |\, M \hookrightarrow_{\mathbf{v}} M' \,| \\
 & & \text{and } s_2 = |\, N \hookrightarrow_{\mathbf{v}} V \,| \\
|\, (M\ N) \hookrightarrow_{\mathbf{v}} (M'\ N') \,| & = & s_1 + s_2 \\
 & & \text{where } s_1 = |\, M \hookrightarrow_{\mathbf{v}} M' \,| \\
 & & \text{and } s_2 = |\, N \hookrightarrow_{\mathbf{v}} N' \,| \\
|\, (o^m\ M_1 \ldots M_m) \hookrightarrow_{\mathbf{v}} (o^m\ M'_1 \ldots M'_m) \,| & = & \Sigma_{i=1}^{m} s_i \\
 & & \text{where } s_i = |\, M_i \hookrightarrow_{\mathbf{v}} M'_i \,| \\
|\, (\lambda X.M) \hookrightarrow_{\mathbf{v}} (\lambda X.M') \,| & = & |\, M \hookrightarrow_{\mathbf{v}} M' \,|
\end{array}$$

Now we are ready to prove the main lemma.

Lemma 5.5: If $M \hookrightarrow_{\mathbf{v}} N$ and $N \diamond N_2 \diamond \ldots N_n \in \mathcal{R}$, then there is a $L_1 \diamond \ldots L_p \in \mathcal{R}$ such that $M = L_1$ and $L_p = N_n$.

Proof for Lemma 5.5: By lexicographic induction on the length n of the given standard reduction sequence, the size of the derivation $M \hookrightarrow_{\mathbf{v}} N$, and the structure of M. It proceeds by case analysis on the last step in the derivation of $M \hookrightarrow_{\mathbf{v}} N$:

- **Case** $M = N$
  This case is trivial.
- **Case** $M = (o^m\ b_1 \ldots\ b_m), N = \delta(o^m, b_1, \ldots b_m)$
  A δ-step that transforms the outermost application is also a standard reduction step. By combining this with the standard reduction sequence from N to N_n yields the required standard reduction sequence from M to N_n.

- **Case** $M = ((\lambda X.K)\ U)$, $N = L[X \leftarrow V]$, $K \hookrightarrow_{\mathbf{v}} L$, $U \hookrightarrow_{\mathbf{v}} V$
  M is also a β_{v}-redex, which as in the previous case, implies that a β_{v}-step from M to $K[X \leftarrow U]$ is a standard reduction step. By the assumptions and Theorem 4.7, the latter expression also parallel reduces to $L[X \leftarrow V]$. Indeed, we can prove a stronger version of Theorem 4.7, see Theorem 5.8 below, that shows that the size of this derivation is strictly smaller than the size of the derivation of $((\lambda x.K)\ U) \hookrightarrow_{\mathbf{v}} L[X \leftarrow V]$. Thus, the induction hypothesis applies and there must be a standard reduction sequence $L_2 \diamond \ldots L_p$ such that $K[X \leftarrow U] = L_2$ and $L_p = N_n$. Since $M \longmapsto_{\mathbf{v}} K[X \leftarrow U]$, the expression sequence $M \diamond L_2 \diamond \ldots L_p$ is the required standard reduction sequence.
- **Case** $M = (M'\ M'')$, $N = (N'\ N'')$, $M' \hookrightarrow_{\mathbf{v}} N'$, $M'' \hookrightarrow_{\mathbf{v}} N''$
  Since the standard reduction sequence $N \diamond N_2 \diamond \ldots N_n$ could be formed in two different ways, we must consider two subcases.
  - **Case** $N \longmapsto_{\mathbf{v}} N_2$
    This case relies on the following Lemma 5.6, which shows that if $M \hookrightarrow_{\mathbf{v}} N$ and $N \longmapsto_{\mathbf{v}} N_2$ then there exists an expression K such that $M \longmapsto_{\mathbf{v}} K$ and $K \hookrightarrow_{\mathbf{v}} N_2$. Now, $K \hookrightarrow_{\mathbf{v}} N_2$ and $N_2 \diamond N_3 \diamond \ldots N_n$ is a standard reduction sequence that is shorter than the original one. Hence, by the induction hypothesis there exists a standard reduction sequence $L_2 \diamond \ldots L_p$ such that $K = L_2$ and $L_p = N_n$. Moreover, $M \longmapsto_{\mathbf{v}} K$, and hence $M \diamond L_2 \diamond \ldots L_p$ is the desired standard reduction sequence.
  - **Case otherwise**
    $N' \diamond \ldots N'_k$ and $N'' \diamond \ldots N''_j$ are standard reduction sequences such that $N \diamond N_2 \diamond \ldots N_n$ is identical to $(N'\ N'') \diamond \ldots (N'_k\ N'') \diamond (N'_k\ N''_2) \diamond \ldots (N'_k\ N''_j)$. By the assumptions and the induction hypothesis, which applies because M' and M'' are proper subterms of M, there exist standard reduction sequences $L'_1 \diamond \ldots L'_{l_1}$ and $L''_1 \diamond \ldots L''_{l_2}$ such that $M' = L'_1$ and $L'_{l_1} = N'_k$ and $M'' = L''_1$ and $L''_{l_2} = N''_j$. Clearly, these two sequences form a single reduction sequence $L_1 \diamond \ldots L_p$ such that $L_1 = (L'_1\ L''_1) = (M'\ M'')$ and $L_p = (L'_{l_1}\ L''_{l_2}) = (N'_k\ N''_j) = N_n$, which is precisely what the lemma demands.
- **Case** $M = (o^m\ M_1 \ldots M_m)$, $N = (o^m\ N'_1 \ldots N'_m)$, $M_i \hookrightarrow_{\mathbf{v}} N_i$
  Again, the standard reduction sequence starting at N could be the result of two different formation rules. The proofs in both subcases though are completely analoguous to the two subcases in the previous case so that there is no need for further elaboration.
- **Case** $M = (\lambda X.K)$, $N = (\lambda X.L)$, $K \hookrightarrow_{\mathbf{v}} L$
  By the definition of standard reduction sequences, all of the expressions N_i are λ-abstractions, say, $N_i = \lambda X.N'_i$. Hence, $K \hookrightarrow_{\mathbf{v}} L$ and $L \diamond N'_2 \diamond \ldots N'_n$ is a standard reduction sequence. Since K is a proper sub-expression of M, the induction hypothesis applies and there must be a standard reduction sequence $L'_1 \diamond \ldots L'_p$ such that $K = L'_1$ and $L'_p = N'_n$. By taking $L_i = \lambda X.L'_i$, we get the desired standard reduction sequence.

These are all possible cases and we have thus finished the proof.

Lemma 5.6: If $M \hookrightarrow_{\mathbf{v}} N$ and $N \longmapsto_{\mathbf{v}} L$, then there exists an expression N^* such that $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N^*$ and $N^* \hookrightarrow_{\mathbf{v}} L$.

Proof for Lemma 5.6: By lexicographic induction on the size of the derivation of $M \hookrightarrow_{\mathbf{v}} N$ and the structure of M. It proceeds by case analysis on the last step in the parallel reduction derivation.

- **Case** $M = N$
  In this case, the conclusion vacuously holds.
- **Case** $M = (o^m\ b_1\ \ldots\ b_m)$**,** $N = \delta(o^m, b_1, \ldots b_m)$
  Since the result of a δ-step is a value, it is impossible that N standard reduces to some other term.
- **Case** $M = ((\lambda X.K)\ U)$**,** $N = L[X \leftarrow V]$**,** $K \hookrightarrow_{\mathbf{v}} L$**,** $U \hookrightarrow_{\mathbf{v}} V$
  M is also a β_{v}-redex, and therefore $M \longmapsto_{\mathbf{v}} K[X \leftarrow U]$. Next, by Lemma 4.7

  $$K[X \leftarrow U] \hookrightarrow_{\mathbf{v}} L[X \leftarrow V],$$

  and the derivation of this parallel reduction is smaller than the derivation of $M \hookrightarrow_{\mathbf{v}} N$ by Lemma 5.8. By induction hypothesis, there must be an expression N^* such that $K[X \leftarrow U] \longmapsto_{\mathbf{v}} N^* \hookrightarrow_{\mathbf{v}} L[X \leftarrow V]$. Hence we can prove the desired conclusion as follows:

  $$M \longmapsto_{\mathbf{v}} K[X \leftarrow U] \longmapsto_{\mathbf{v}} N^* \hookrightarrow_{\mathbf{v}} L[X \leftarrow V.]$$

- **Case** $M = (M'\ M'')$**,** $N = (N'\ N'')$**,** $M' \hookrightarrow_{\mathbf{v}} N'$**,** $M'' \hookrightarrow_{\mathbf{v}} N''$
  Here we distinguish three subcases according to the standard reduction step from N to L:
  - **Case** $N = ((\lambda X.K')\ N'')$ where $N'' \in V$
    That is, N is a β_{v}-redex and L is its contractum. By the assumed parallel reductions and Lemma 5.7 below, there exist $(\lambda X.K)$ and N^{**} such that $M' \longmapsto\!\!\!\!\to_{\mathbf{v}} (\lambda X.K) \hookrightarrow_{\mathbf{v}} (\lambda X.K')$ and $M'' \longmapsto\!\!\!\!\to_{\mathbf{v}} N^{**} \hookrightarrow_{\mathbf{v}} N''$. Hence,

    $$\begin{array}{rl} (M'\ M'') & \longmapsto\!\!\!\!\to_{\mathbf{v}}\ ((\lambda X.K)\ M'') \\ & \longmapsto\!\!\!\!\to_{\mathbf{v}}\ ((\lambda X.K)\ N^{**}) \\ & \longmapsto_{\mathbf{v}}\ K[X \leftarrow N^{**}] \\ & \hookrightarrow_{\mathbf{v}}\ K'[X \leftarrow N''], \end{array}$$

    which is precisely what the lemma claims.
  - **Case** $N = (E[K]\ N'')$
    Then, $L = (E[K']\ N'')$ where $(K, K'') \in \mathbf{v}$ and

    $$M' \hookrightarrow_{\mathbf{v}} E[K] \longmapsto_{\mathbf{v}} E[K'].$$

    By the induction hypothesis, which applies because M' is a proper subexpression of M, this means that there exists an expression N_1^* such that

    $$M' \longmapsto\!\!\!\!\to_{\mathbf{v}} N_1^* \hookrightarrow_{\mathbf{v}} E[K'],$$

    which implies that

    $$(M'\ M'') \longmapsto\!\!\!\!\to_{\mathbf{v}} (N_1^*\ M'') \hookrightarrow_{\mathbf{v}} (E[K']\ N'').$$

    In other words, $N^* = (N_1^*\ M'')$.
  - **Case** $N = (N'\ E[K])$ and $N' \in V$
    In this case, $L = (N'\ E[K'])$ where $(K, K'') \in \mathbf{v}$ and

    $$M'' \hookrightarrow_{\mathbf{v}} E[K] \longmapsto_{\mathbf{v}} E[K'].$$

    The induction hypothesis again implies the existence of an expression N_2^* such that

    $$M'' \longmapsto\!\!\!\!\to_{\mathbf{v}} N_2^* \hookrightarrow_{\mathbf{v}} E[K'].$$

Since M' may not be a value, we apply the following lemma to get a value N_1^* such that

$$M' \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N_1^* \hookrightarrow_{\mathbf{v}} N'.$$

Putting it all together we get,

$$\begin{array}{rcl} (M'\ M'') & \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} & (N_1^*\ M'') \\ & \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} & (N_1^*\ N_2^*) \\ & \longmapsto_{\mathbf{v}} & (N_1^*\ N_2^*) \\ & \hookrightarrow_{\mathbf{v}} & (N'\ E[K']). \end{array}$$

And thus, $N^* = (N_1^*\ N_2^*)$ in this last subcase.

- **Case** $M = (o^m\ M_1 \ldots M_m)$**,** $N = (o^m\ N_1' \ldots N_m')$**,** $M_i \hookrightarrow_{\mathbf{v}} N_i$
  The proof is analoguous to the preceding one though instead of a β_{v} step the first subcase is a δ-step. That is, all N_i' are basic constants and N standard reduces to a value. Since Lemma 5.7 shows that a term that parallel reduces to a basic constant also standard reduces to it, the rest of the argument is straightforward.
- **Case** $M = (\lambda X.K)$**,** $N = (\lambda X.L)$**,** $K \hookrightarrow_{\mathbf{v}} L$
  This case is again impossible because the standard reduction function is undefined on values.

This completes the case analysis and the proof.

Lemma 5.7: Let M be an application.

1. If $M \hookrightarrow_{\mathbf{v}} (\lambda X.N)$ then there exists an expression $(\lambda X.L)$ such that $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} (\lambda X.L) \hookrightarrow_{\mathbf{v}} (\lambda X.N)$.
2. If $M \hookrightarrow_{\mathbf{v}} N$ where $N = x$ or $N = b$ then $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N$.

Proof for Lemma 5.7: Both implications follow from an induction argument on the size of the derivation for the parallel reduction in the antecedent.

1. Only a parallel δ or a parallel β_{v} reduction can transform an application into a λ-abstraction. Clearly, δ reductions are also standard reductions and therefore the result is immediate in that case. Parallel β_{v} redexes are standard redexes. Thus, $M = ((\lambda Y.M')\ U)$, which parallel reduces to $(\lambda X.N) = N'[Y \leftarrow V]$ because $M' \hookrightarrow_{\mathbf{v}} N'$ and $U \hookrightarrow_{\mathbf{v}} V$ by Lemma 4.7. Hence,

   $$((\lambda Y.M')\ U) \longmapsto_{\mathbf{v}} M'[Y \leftarrow U] \hookrightarrow_{\mathbf{v}} N'[Y \leftarrow V] = (\lambda X.N)$$

   By Lemma 5.8, we also know that the latter derivation is shorter than the original parallel reduction step and therefore, by inductive hypothesis, there exists an expression $(\lambda X.L)$ such that

   $$((\lambda Y.M')\ U) \longmapsto_{\mathbf{v}} M'[Y \leftarrow U] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} (\lambda X.L) \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} (\lambda X.N)$$

2. Again, the only two parallel reductions that can transfrom a redex in the shape of an application into a constant or variable are δ and the β_{v} (parallel) reductions. Thus the argument proceeds as for the first part.

Lemma 5.8: If $M \hookrightarrow_{\mathbf{v}} N$ has size s_M and $U \hookrightarrow_{\mathbf{v}} V$ has size s_N then (i) $M[X \leftarrow U] \hookrightarrow_{\mathbf{v}} N[X \leftarrow V]$ and (ii) size s of the derivation of the latter is less than or equal to $s_M + \#(x, N) \times s_U$.

Remark 1: This lemma is a strengthening of Lemma 4.7. For completeness, we repeat the proof of the original lemma and add components about the size of the parallel reduction derivation.

Remark 2: The lemma implies that the size of the derivation of $((\lambda x.M)\ U) \hookrightarrow_{\mathbf{v}} N[X \leftarrow V]$ is strictly larger than the size of the derivation of $M[X \leftarrow U] \hookrightarrow_{\mathbf{v}} N[X \leftarrow V]$.

Proof for Lemma 5.8: The proof is an induction on M and proceeds by case analysis on the last step in the derivation of $M \hookrightarrow_{\mathbf{v}} N$:

- **Case** $M = N$
  In this special case, the claim says that if $U \hookrightarrow_{\mathbf{v}} V$ then $M[X \leftarrow U] \hookrightarrow_{\mathbf{v}} M[X \leftarrow V]$. *As to the claim about size: the assumption implies that* $s_M = 0$*, and therefore it must be true that* $s \leq \#(x, M) \times s_U$. The derivation of this specialized claim is an induction on the structure of M:
  - **Case** $M = b$
    $M[X \leftarrow U] = b \hookrightarrow_{\mathbf{v}} b = M[X \leftarrow V]$;
    size: $s = 0$;
  - **Case** $M = X$
    $M[X \leftarrow U] = U \hookrightarrow_{\mathbf{v}} V = M[X \leftarrow V]$;
    size: $s = s_U = \#(x, M) \times s_U$;
  - **Case** $M = Y \neq X$
    $M[X \leftarrow U] = y \hookrightarrow_{\mathbf{v}} y = M[X \leftarrow V]$;
    size: $s = 0$;
  - **Case** $M = (\lambda Y.K)$
    $M[X \leftarrow U] = (\lambda Y.K[X \leftarrow U)] \hookrightarrow_{\mathbf{v}} (\lambda Y.K[X \leftarrow V)\ =]M[X \leftarrow V]$ by induction hypothesis for K;
    size: $s = \#(x, K) \times s_U = \#(x, M) \times s_U$;
  - **Case** $M = (K\ L)$
    $M[X \leftarrow U] = (K[X \leftarrow U]\ L[X \leftarrow U)] \hookrightarrow_{\mathbf{v}} (K[X \leftarrow V]\ L[X \leftarrow V)\ =]M[X \leftarrow V]$ by induction hypothesis for K, L;
    size: $s = \#(x, K) \times s_U + \#(x, L) \times s_U = \#(x, M) \times s_U$.
- **Case** $M = (o^m\ b_1 \ldots b_m), N = \delta(o^m, b_1, \ldots b_m)$
  Here, M and N are closed, so $M[X \leftarrow U] = M$, $N[X \leftarrow V] = N$, and the result follows directly from the assumption.
  Size: $s = 0$.
- **Case** $M = ((\lambda y.K)\ W)$, $N = L[y \leftarrow W']$, $K \hookrightarrow_{\mathbf{v}} L$, $W \hookrightarrow_{\mathbf{v}} W'$
  A simple calculation shows that the claim holds:

$$\begin{array}{rcll} M[X \leftarrow U] & = & ((\lambda y.K[X \leftarrow U)]\ W[X \leftarrow U)] & \\ & \hookrightarrow_{\mathbf{v}} & L[X \leftarrow V][X \leftarrow W'y[] \leftarrow V] & \\ & = & L[y \leftarrow W'][X \leftarrow V] & (\dagger) \\ & = & N[X \leftarrow V] & \end{array}$$

  Equation (†) follows from a simple induction on the structure of L.
  The size of the derivation is calculated as follows. Let us assume the following conventions about sizes:

$$\begin{array}{ll} s_K: & K \hookrightarrow_{\mathbf{v}} L \\ s'_K: & K[X \leftarrow U] \hookrightarrow_{\mathbf{v}} L[X \leftarrow V] \\ s_W: & W \hookrightarrow_{\mathbf{v}} W' \\ s'_W: & W[X \leftarrow U] \hookrightarrow_{\mathbf{v}} W'[X \leftarrow V] \end{array}$$

By induction hypothesis,

$$\begin{array}{rcl} s'_K & \leq & s_K + \#(x, L) \times s_U \\ s'_W & \leq & s_W + \#(x, W') \times s_U \end{array}$$

Hence, the size for the entire derivation is

$$\begin{array}{rcl} s & = & s'_K + \#(y, L) \times s'_W + 1 \\ & \leq & s_K + \#(x, L) \times s_U + \#(y, L) \times (s_W + \#(x, W') \times s_U) + 1 \\ & = & s_K + \#(y, L) \times s_W + 1 + (\#(y, L) \times \#(x, W') + \#(x, L)) \times s_U \\ & = & s_M + \#(x, N) \times s_U \end{array}$$

- **Case** $M = (M_1\ M_2)$**,** $N = (N_1\ N_2)$**,** $M_i \hookrightarrow_{\mathbf{v}} N_i$
  By the induction hypothesis, $M_i[X \leftarrow U] \hookrightarrow_{\mathbf{v}} N_i[X \leftarrow V]$ for both $i = 1$ and $i = 2$. Thus,

$$\begin{array}{rcl} M[X \leftarrow U] & = & (M_1[X \leftarrow U]\ M_2[X \leftarrow U)] \\ & \hookrightarrow_{\mathbf{v}} & (N_1[X \leftarrow V]\ N_2[X \leftarrow V)] \\ & = & (N_1\ N_2)[X \leftarrow V] \\ & = & N[X \leftarrow V] \end{array}$$

  For the size argument, we again begin by stating some conventions:

$$\begin{array}{ll} s_i : & M_i \hookrightarrow_{\mathbf{v}} N_i \\ s'_i : & M_i[X \leftarrow U] \hookrightarrow_{\mathbf{v}} N_i[X \leftarrow V] \end{array}$$

  Thus, by induction hypothesis,

$$s'_i \leq s_i + \#(x, N_i) \times s_U.$$

  The rest follows from a simple calculation:

$$\begin{array}{rcl} s & = & s'_1 + s'_2 \\ & \leq & s_1 + \#(x, N_1) \times s_U + s_2 + \#(x, N_2) \times s_U \\ & = & s_1 + s_2 + (\#(x, N_1) + \#(x, N_2)) \times s_U \\ & = & s_M + \#(x, N) \times s_U \end{array}$$

- **Case** $M = (o^m\ M_1 \ldots M_m)$**,** $N = (o^m\ N'_1 \ldots N'_m)$**,** $M_i \hookrightarrow_{\mathbf{v}} N_i$
  Analogous to the previous case.
- **Case** $M = (\lambda x.K)$**,** $N = (\lambda x.L)$**,** $K \hookrightarrow_{\mathbf{v}} L$
  Analogous to the previous case.

These are all possible cases and we have thus finished the proof.

The reward of proving the standard reduction theorem is a proof of Theorem 5.2, the correctness proof for the textual machine.

Theorem 5.2: $M \rightarrow\!\!\!\!\rightarrow_{\mathbf{v}} U$ if and only if $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$ for some value V and $V \rightarrow\!\!\!\!\rightarrow_{\mathbf{v}} U$.

In particular, if $U \in b$ then $M \rightarrow\!\!\!\!\rightarrow_{\mathbf{v}} U$ if and only if $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} U$.

Proof for Theorem 5.2: To prove the left to right direction, assume that

$$M \rightarrow\!\!\!\!\rightarrow_{\mathbf{v}} U$$

and that M is not a value. It follows from Theorem 5.5 that there exists a standard reduction sequence $L_1 \diamond \ldots L_p$ such that $M = L_1$ and $L_p = U$. By induction on the length of the sequence, there exists an index i such that L_i is the first value in the sequence and

$$M = L_1 \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} L_i.$$

Set $V = L_i$; trivially, $L_i \rightarrow\!\!\!\!\rightarrow_{\mathbf{v}} U$. Moreover, constants can only be the end of a standard reduction sequence of values if the standard reduction sequence is a singleton. Hence, the second part of the theorem concerning basic constants obviously holds.

In summary, we have shown that the specification of the evaluation function based on the equational system is equivalent to an evaluator based on the standard reduction function. Since the latter is a truly algorithmic specification, we can now implement the evaluator easily.

5.3 Observational Equivalence

Two expressions are **of the same kind** if both are values or both are applications (non-values).

Lemma 5.9 [Activity]: If M is a closed expression and $eval_{\mathbf{v}}(C[M])$ exists, then

(i) either $eval_{\mathbf{v}}(C[M'])$ exists for all closed expressions M' that are of the same kind as M; or

(ii) for all closed expressions M' that are of the same kind as M, there is some evaluation context E such that

(a) if M is an application, then

$$C[M'] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[M']$$

(b) if M is a λ-abstraction, then for some U,

$$C[M'] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[(M'\ U)]$$

(c) if M is a constant, then for some primitive o^n and basic constants $b_1, \ldots b_n$,

$$C[M'] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[(o\ b_1\ \ldots\ M'\ \ldots\ b_n)]$$

Proof for Lemma 5.9: Intuitively, the proof proceeds by induction on the number of standard reduction steps in the reduction:

$$C[M] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$$

However, because M may be duplicated during $\beta_{\mathbf{v}}$-reductions, the induction hypothesis needs to be strengthened. The stronger claim is subject of the following lemma.

The formulation of an appropriate induction hypothesis for the Activity Lemma requires the notion of a multi-hole context.

$$\begin{array}{rcl} \overline{C} & = & [\,]_n \\ & | & M \\ & | & (\lambda x.\overline{C}) \\ & | & (\overline{C}\ \overline{C}) \\ & | & (o^n\ \overline{C} \ldots\ \overline{C}) \end{array}$$

A multi-hole context with m holes is well-formed if each hole $[\,]_i$ has a distinct label $i \in [1, m]$. The notation $\overline{C}[M_1, \ldots M_n]$ means the expression generated by replacing each $[\,]_i$ in $\overline{C}$ with M_i.

The stronger version of Lemma 5.9 re-states the claim for m expressions and contexts with m holes.

Lemma 5.10: If $M_1, \ldots M_m$ are closed expressions, $\overline{C}$ is a context with m holes, and $eval_{\mathbf{v}}(\overline{C}[M_1, \ldots M_m])$ exists, then

(i) either $eval_{\mathbf{v}}(\overline{C}[M'_1, \ldots M'_m])$ for all closed expressions $M'_1, \ldots M'_m$ that are of the same kind as $M_1, \ldots M_m$, respectively; or

(ii) for all closed expressions $M'_1, \ldots M'_m$ that are of the same kind as $M_1, \ldots M_m$, respectively, there is some evaluation context E such that for some $i \in [1, m]$,

(a) if M_i is an application, then

$$\overline{C}[M'_1, \ldots M'_m] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[M'_i]$$

(b) if M_i is a λ-abstraction, then for some value U,

$$\overline{C}[M'_1, \ldots M'_m] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[(M'_i\ U)]$$

(c) if M_i is a constant, then for some primitive o^n and basic constants $b_1, \ldots b_n$,

$$\overline{C}[M'_1, \ldots M'_m] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[(o\ b_1\ \ldots\ M'\ \ldots\ b_m)]$$

Proof for Lemma 5.10: The proof is an induction on the number of standard reduction steps for the original program. If $\overline{C}[M_1, \ldots M_m]$ is a value, part (i) of the conclusion clearly applies. Consequently, assume that

$$\overline{C}[M_1, \ldots M_m] \longmapsto_{\mathbf{v}} K \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$$

for some expression K and value V. By Theorem 5.1, the program can be partitioned into an evaluation context E' and a redex $(N\ L)$ such that

$$E'[(N\ L)] = \overline{C}[M_1, \ldots M_m],$$

or into an evaluation context E' and a redex $(o^n\ b'_1 \ldots\ b'_p)$ such that

$$E'[(o^n\ b'_1 \ldots\ b'_p)] = \overline{C}[M_1, \ldots M_m]$$

We deal with the first case; the proof of the second case proceeds in an analogous manner.

1. Some expression M_i contains, or is equal to, the redex $(N\ L)$. Since $(N\ L)$ is an application and E' an evaluation context, this is only possible if M_i is an application. If this is the case, the first clause of part (ii) of the lemma's conclusion holds: for any expressions $M'_1, \ldots M'_m$ of the correct kind,

$$E = \overline{C}[M'_1, \ldots M'_{i-1}, [\,]_i, \ldots M'_m]$$

is an evaluation context. The key to this claim is that the expressions $M'_1, \ldots M'_m$ are of the same kind as the expressions $M_1, \ldots M_m$, respectively. If

$$E' = E_1[(M_j\ E_2)]$$

where M_j, $1 \leq j \leq m$, is a value, and E_1 and E_2 are evaluation contexts, then if M'_j were an application, E would not be an evaluation context.

2. None of the expressions $M_1, \ldots M_m$ contains the redex, but the redex contains some of these expressions. Now we must consider two subcases:

   (a) Some λ-abstraction M_i is N. Now the second clause of part (ii) holds: for all closed expressions $M'_1, \ldots M'_m$,

   $$E = \overline{C}[M'_1, \ldots M'_{i-1}, [\,]_i, \ldots M'_m]$$

   and U differs from L only in occurrences of holes of $\overline{C}$. Again, as in the preceding case, E is an evaluation context because the expressions $M'_1, \ldots M'_m$ are of the right kind. Similarly, if $L = M_j$, for some j, $1 \leq j \leq m$, then if M'_j were not a value, U wouldn't be a value.

   (b) The value N is a λ-abstraction, $\lambda X.N'$, that may contain some of the expressions $M_1, \ldots M_m$. Now:

   $$\overline{C}[M_1, \ldots M_m] = E'[((\lambda X.N')\ L)] \longmapsto_{\mathbf{v}} E'[N'[L] \leftarrow x]$$

   As in the first subcase, since the expressions $M'_1, \ldots M'_m$ are of the right kind,

   $$\overline{C}[M'_1, \ldots M'_m] = E''[((\lambda X.N'')\ L')] \longmapsto_{\mathbf{v}} E''[N''[L'] \leftarrow x]$$

   for some evaluation context E'', expression N'', and value L' that differ accordingly from E', N', and L respectively. Since the expressions $M_1, \ldots M_m$ are closed, there is some context $\overline{C}'$ of q holes and a sequence of indices $1 \leq j_1, \ldots j_q \leq m$ such that

   $$E'[N'[L] \leftarrow x] = \overline{C}'[M_{j_1}, \ldots M_{j_q}]$$

   and

   $$E''[N''[L'] \leftarrow x] = \overline{C}'[M'_{j_1}, \ldots M'_{j_q}]$$

   The intermediate program satisfies the antecedent of our claim, the reduction

   $$\overline{C}'[M_{j_1}, \ldots, M_{j_q}] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$$

   is shorter, and therefore the induction hypothesis applies.

Equipped with the Activity Lemma, we are ready to finish Theorem 4.10 with the following lemma. It shows the observational equivalence of two expressions that are not provably equal.

Lemma 5.11: $\Omega \simeq_{\mathbf{v}} (\Omega\ (\lambda x.x))$

Proof for Lemma 5.11: Assume for some C that $eval_{\mathbf{v}}(C[\Omega])$ exists. By the Activity Lemma, it could be the case that for all closed expressions M there is some evaluation context such that

$$C[M] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} E[M]$$

But this contradicts the assumption that $C[\Omega]$ exists because $E[\Omega]$ is a diverging program. Hence, $eval_{\mathbf{v}}(C[M])$ exists for all closed expressions M and in particular for $M = (\Omega\ (\lambda x.x))$.

Conversely, assume that $eval_{\mathbf{v}}(C[(\Omega\ (\lambda x.x))])$ exists, then by an argument along the same lines, $eval_{\mathbf{v}}(C[\Omega])$ exists. Clearly, if one of the programs returns an abstraction, then so does the other, and both return the same basic constant. In short,

$$eval_{\mathbf{v}}(C[(\Omega\ (\lambda x.x))]) = eval_{\mathbf{v}}(C[\Omega])$$

This proves that $\Omega \simeq_{\mathbf{v}} (\Omega\ (\lambda x.x))$, because C is arbitrary.

5.4 Uniform Evaluation

Another question that can now be tackled is the following: what kind of behavior can we expect from a machine given an arbitrary program? The evaluator can diverge for two *different* reasons. First, every intermediate state of the evaluation may have a successor state. That is, the program is looping. Second, the evaluation may reach an intermediate state for which the standard reduction function is undefined:

$$M_1 \longmapsto_{\mathbf{v}} M_2 \longmapsto_{\mathbf{v}} \ldots \longmapsto_{\mathbf{v}} M_m$$

such that $\longmapsto_{\mathbf{v}}$ is undefined for M_m and M_m is not an answer. For this to be the case, M_m must contain an application consisting of values in the hole of an evaluation contexts such that neither the β_{v} nor the δ reduction applies, i.e.,

$$\begin{array}{lcll} & M_m & = & E[(U\ V)] \qquad \text{where } U \text{ is not a basic constant} \\ \text{or} & & & \\ & M_m & = & E[(o^n\ V_1 \ldots V_n)] \qquad \begin{array}{l}\text{where some } V_i \text{ is a } \lambda\text{-abstraction} \\ \text{or } \delta(o^n, V_1, \ldots, V_n) \\ \text{is undefined}\end{array} \end{array}$$

Adopting a machine-oriented view, programs for which $\longmapsto_{\mathbf{v}}$ is undefined are called **stuck states**. Stuck states play an important role in the analysis of machines and other programming language tools. A precise understanding of the stuck states of any given programming language is therefore crucial. We first give a formal definition of ISWIM stuck states.

An application M is **stuck** if

1. $M = (o^m\ b_1 \ldots\ b_m)$ and $\delta(o^m, b_1, \ldots b_m)$ is undefined; or
2. $M = (o^m\ b_1 \ldots\ b_{i-1}\ (\lambda X.N)\ b_{i+1} \ldots\ b_m)$; or
3. $M = (b\ V)$.

A program M is **stuck** if $M = E[N]$ for some evaluation context E and a stuck application N.

A program in pure ISWIM can only evaluate to a value, a stuck program, or diverge.

Lemma 5.12: If M is a closed expression, then either

1. $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$ for some value V;
2. $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N$ for some stuck program N;
3. for all N such that $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N$, there exists an L such that $N \longmapsto_{\mathbf{v}} L$.

Proof for Lemma 5.12: Assume there exists an N such that $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N$ and $\longmapsto_{\mathbf{v}}$ is undefined for N. That is,

$$M_1 \longmapsto_{\mathbf{v}} M_2 \longmapsto_{\mathbf{v}} M_3 \longmapsto_{\mathbf{v}} \ldots \longmapsto_{\mathbf{v}} M_m = N$$

By Unique Evaluation (Lemma 5.1), N is either a value, or there is an evaluation context E, possibly a primitive o^n, and values U, V, V_1, $\ldots$ V_n such that

$$N = E[(U\ V)] \text{ or } N = E[(o^n\ V_1\ \ldots\ V_n)]$$

If the applications in the hole are redexes, we have a contradiction to our assumption. Consequently, U is neither a λ-abstraction nor a primitive but is a basic constant. Similarly, some V_i must not be a basic constant or $\delta(o^n, V_1, \ldots V_n)$ is not defined. Hence, N is a stuck program. In conclusion, programs either diverge, or reduce to a value or to a stuck state.

Chapter 9 defines a language with a uniform treatment of stuck states.

Chapter 6: Machines

Every evaluation cycle in the standard-reduction textual machine performs three steps:

1. It tests whether the program is a value; if so, the machine stops.
2. If the program is an application, the machine partitions the program into an evaluation context, E, and an application, $(U\ V)$ or $(o^n\ V_1 \ldots V_n)$.
3. If the application is a $\beta_{\mathtt{v}}$ or δ redex, the machine constructs the contractum M and fills the evalutaion context E with the contractum M.

The result of such a step is the program $E[M]$, which is the next machine state. Now the evaluation cycle starts over.

An obvious inefficiency in this machine is the repeated partitioning of a program into an evaluation context and a redex. Clearly, the evaluation context of the $(n+1)$st step is often the same as, or at least, is closely related to the one for the nth step. For example, consider the evaluation of

$$(\texttt{add1}\ ((\lambda x.((\lambda y.((\lambda z.x)\ \ulcorner 3 \urcorner))\ \ulcorner 2 \urcorner))\ \ulcorner 1 \urcorner))$$

which is a fairly typical example:

$$\begin{array}{rl} & (\texttt{add1}\ \underline{((\lambda x.((\lambda y.((\lambda z.x)\ \ulcorner 3 \urcorner))\ \ulcorner 2 \urcorner))\ \ulcorner 1 \urcorner)}) \\ \longmapsto_{\mathtt{v}} & (\texttt{add1}\ \underline{((\lambda y.((\lambda z.\ulcorner 1 \urcorner)\ \ulcorner 3 \urcorner))\ \ulcorner 2 \urcorner)}) \\ \longmapsto_{\mathtt{v}} & (\texttt{add1}\ \underline{((\lambda z.\ulcorner 1 \urcorner)\ \ulcorner 3 \urcorner)}) \\ \longmapsto_{\mathtt{v}} & \underline{(\texttt{add1}\ \ulcorner 1 \urcorner)} \\ \longmapsto_{\mathtt{v}} & \ulcorner 2 \urcorner. \end{array}$$

The redexes in each intermediate state are underlined, the evaluation context is the part of the term that is not underlined. The evaluation context for the first three states is $(\texttt{add1}\ [\,])$. But the machine has to recompute the evaluation context at every stage.

Depending on the use of the standard-reduction machine, the efficiency may not be an issue. For example, if the goal of the machine is simply to formally define $eval^{\mathtt{s}}_{\mathtt{v}}$, then efficiency is of no interest. However, if we want to explain an *implementation* of a language to others, then we may want to formalize a more "realistic" definition of the language. In particular, no real implementation of an ISWIM-like language would find the context from scratch on each step.

6.1 CC Machine

To eliminate repeated context computations, we can separate program states into the "current subterm of interest" and the "current evaluation context." Putting the former into the hole of the latter yields the complete program. We will refer to the term of interest as the **control string** M and the evaluation context simply as the **context** E. These two elements are paired together to form the machine's state: $\langle M, E \rangle$. The machine is called the **CC machine**.

In addition to the reduction tasks of the textual machine, the CC machine is responsible for finding the redex. Initially, the entire program is the control string, and the initial evaluation context is the empty context. The machine's search instructions implement rules 1 and $2'$ from the beginning of the previous chapter. That is, if the control string is not an application that consists of values, the leftmost non-value becomes the control string, and the rest becomes a part of the evaluation context.

For example, if the control string is $((K\ L)\ N)$ and the evaluation context is E, then the machine must search for the redex in $(K\ L)$, and it must remember the shape of the rest of the

program. Consequently the next state of the machine must have $(K\ L)$ as the control string component and must refine the current context E to a context of the shape $E[([\,]\ N)]$. In other words, the CC machine must have a state transition of the form

$$\langle (M\ N), E\rangle \longmapsto_{\mathbf{v}} \langle M, E[([\,]\ N)]\rangle \quad \text{if} \quad M \notin V$$

Search rules for other shapes of applications are needed, too.

After the control string becomes a redex, the CC machine must simulate the actions of the textual machine on a redex. Thus it must have the following two instructions:

$$\begin{array}{lll} \langle (o^m\ b_1 \ldots\ b_m), E\rangle & \longmapsto_{\mathbf{v}} & \langle V, E\rangle \ \text{ where } V = \delta(o^m, b_1, \ldots, b_m) \qquad [\mathsf{cc}\delta] \\ \langle ((\lambda X.M)V), E\rangle & \longmapsto_{\mathbf{v}} & \langle M[X \leftarrow V], E\rangle \qquad [\mathsf{cc}\beta_{\mathsf{v}}] \end{array}$$

The result of such a transition can be a state that pairs a value with an evaluation contexts. In the textual machine, this corresponds to a step of the form $E[L] \longmapsto_{\mathbf{v}} E[V]$. The textual machine would actually divide $E[V]$ into a new redex L' and a new evaluation context E' that is distinct from E. But the textual machine clearly does not pick random pieces from the evaluation context to form the next redex. If $E = E^*[((\lambda X.M)\ [\,])]$, then $E' = E^*$ and $L' = ((\lambda X.M)\ V)$. That is, the new redex is formed of the innermost application in E, and E' is the rest of E.

Thus, for a faithful simulation of the textual machine, the CC machine needs a set of instructions that exploit the information surrounding the hole of the context when the control string is a value. In the running example, the CC machine would have to make a transition from

$$\langle V, E^*[((\lambda X.M)\ [\,])]\rangle$$

to

$$\langle ((\lambda X.M)\ V), E^*\rangle$$

At this point, the control string is an application again, and the search process can start over.

Putting it all together, the evaluation process on a CC machine consists of shifting pieces of the control string into the evaluation context such that the control string becomes a redex. Once the control string has turned into a redex, an ordinary contraction occurs. If the result is a value, the machine must shift pieces of the evaluation context back to the control string.

The $\longmapsto_{\mathsf{cc}}$ reduction relation on CC machine states is defined as follows:

$$\begin{array}{lll} \langle (M\ N), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E[([\,]\ N)]\rangle \qquad [\mathsf{cc1}] \\ \quad \text{if } M \notin V & & \\ \langle (V_1\ M), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E[(V_1\ [\,])]\rangle \qquad [\mathsf{cc2}] \\ \quad \text{if } M \notin V & & \\ \langle (o^n\ V_1 \ldots V_i\ M\ N \ldots), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E[(o^n\ V_1 \ldots V_i\ [\,]\ N \ldots)]\rangle \qquad [\mathsf{cc3}] \\ \quad \text{if } M \notin V & & \\ \langle ((\lambda X.M)\ V), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M[X \leftarrow V], E\rangle \qquad [\mathsf{cc}\beta_{\mathsf{v}}] \\ \langle (o^m\ b_1 \ldots\ b_m), E\rangle & \longmapsto_{\mathsf{cc}} & \langle V, E\rangle \qquad [\mathsf{cc}\delta] \\ & & \quad \text{where } V = \delta(o^m, b_1, \ldots b_m) \\ \langle V, E[(U\ [\,])]\rangle & \longmapsto_{\mathsf{cc}} & \langle (U\ V), E\rangle \qquad [\mathsf{cc4}] \\ \langle V, E[([\,]\ N)]\rangle & \longmapsto_{\mathsf{cc}} & \langle (V\ N), E\rangle \qquad [\mathsf{cc5}] \\ \langle V, E[(o^n\ V_1 \ldots V_i\ [\,]\ N \ldots)]\rangle & \longmapsto_{\mathsf{cc}} & \langle (o^n\ V_1 \ldots V_i\ V\ N \ldots), E\rangle \qquad [\mathsf{cc6}] \end{array}$$

$$eval_{\mathsf{cc}}(M) = \begin{cases} b & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle b, [\,]\rangle \\ \texttt{function} & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle \lambda X.N, [\,]\rangle \end{cases}$$

By the derivation of the CC machine, it is almost obvious that it faithfully implements the textual machine. Since evaluation on both machines is defined as a partial function from

programs to answers, a formal justification for this claim must show that the two functions are identical.

Theorem 6.1: $eval_{\mathsf{cc}} = eval^{\mathsf{s}}_{\mathbf{v}}$.

Proof for Theorem 6.1: We need to show that if $eval_{\mathsf{cc}}(M) = A$ then $eval^{\mathsf{s}}_{\mathbf{v}}(M) = A$ and vice versa. By the definition of the two evaluation functions, this requires a proof that

$$M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V \quad \text{if and only if} \quad \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle.$$

A natural approach to the proof of this claim is an induction of the length of the given transition sequence. However, since intermediate states have non-empty evaluation contexts, we will need a slightly stronger induction hypothesis:

$$E[M] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V \quad \text{if and only if} \quad \langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle.$$

The hypothesis clearly implies the general theorem. Thus the rest of the proof is a proof of this intermediate claim. The proof for each direction is an induction on the length of the reduction sequence.

To prove the left-to-right direction of the claim, assume $E[M] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$. If the reduction sequence is empty, the conclusion is immediate, so assume there is at least one step:

$$E[M] \longmapsto_{\mathbf{v}} E'[N] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$$

where $L\mathbf{v}N$ for some application expression L where $E'[L] = E[M]$. By the inductive hypothesis, $\langle N, E'\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle$. To conclude the proof, we must show that

$$\langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle N, E'\rangle$$

If L is inside M (or the same), then E' must be an extension of E (or the same). Otherwise, M must be a value, because it is inside the redex and also in an evalaution position. Thus, depending on the shape of M and E, there are four possibilities:

- **Case** $M = E''[L]$
  Then, $E' = E[E'']$. By Lemma 6.2 below, $\langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle L, E[E'']\rangle$ and, hence, by a $[\mathsf{cc}\delta]$ or $[\mathsf{cc}\beta_{\mathsf{v}}]$ transition, $\langle L, E[E'']\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle N, E[E'']\rangle$.
- **Case** $E = E'[((\lambda X.N')\ [\,])], M \in V$
  A first step puts the redex into the control position, which enables a $[\mathsf{cc}\beta_{\mathsf{v}}]$ reduction:

  $$\begin{array}{rcl} \langle M, E'[((\lambda X.N')\ [\,])]\rangle & \longmapsto_{\mathsf{cc}} & \langle((\lambda X.N')\ M), E'\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle N'[X \leftarrow M], E'\rangle \end{array}$$

  Since the assumptions imply that $N = N'[X \leftarrow M]$, this proves the conclusion.
- **Case** $E = E''[([\,]\ K)], M \in V$
  Depending on the nature of K, there are two different scenarios:
  1. If $K \in V$ then $L = (M\ K)$, which implies the conclusion.
  2. If $K \notin V$ then there exists an evaluation context E''' such that $K = E'''[L]$. Hence, $E' = E''[(M\ E'''[L])]$ and

     $$\begin{array}{rcl} \langle M, E''[([\,]\ K)]\rangle & \longmapsto_{\mathsf{cc}} & \langle(M\ K), E''\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle K, E''[(M\ [\,])]\rangle \\ & \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} & \langle L, E''[(M\ E''')]\rangle \text{ by Lemma 6.2} \\ & \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} & \langle N, E''[(M\ E''')]\rangle \\ & = & \langle N, E'\rangle. \end{array}$$

- **Case** $E = E'[(o^n\ b_1 \ldots\ [\,]\ \ldots\ b_n)], M \in b$
  Appropriate adaptations of the proofs of the second and third case prove the conclusion.

For the right-to-left direction, assume $\langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle$. The proof is a again an induction on the number of transition steps. If there are none, $M \in V$ and $E = [\,]$, which implies the claim. Otherwise, there is a first transition step:

$$\langle M, E\rangle \longmapsto_{\mathsf{cc}} \langle N, E'\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle$$

By inductive hypothesis, $E'[N] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$. The rest of the proof depends on which kind of transition the CC machine made to reach the state $\langle N, E'\rangle$. If the first transition was one of [cc1], ... [cc6], then $E[M] = E'[N]$, and the conclusion follows. If it is a [ccβ_{v}] or a [ccδ] transition, then $E[M] \longmapsto_{\mathbf{v}} E'[N]$, so $E[M] \longmapsto_{\mathbf{v}} E'[N] \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$.

Lemma 6.2: If $M = E'[L]$ and $L\ \mathbf{v}\ L'$ for some L', then

$$\langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle L, E[E']\rangle$$

Proof for Theorem 6.2: The proof is an induction on the structure of E'. If $E' = [\,]$ then the conclusion is immediate. Otherwise, E' has one of the following three shapes:

$$E' = (E''\ N)$$
$$E' = (V\ E'')$$
$$E' = (o^m\ V_1\ \ldots V_n\ E''\ N \ldots)$$

In each case, the machine can move the pieces surrounding E'' to the context component, e.g.,

$$\langle (E''[L]\ N), E\rangle \longmapsto_{\mathsf{cc}} \langle E''[L], E[([\,]\ N)]\rangle$$

Since E'' is a component of E', the conclusion follows from the inductive hypothesis.

▷ **Exercise 6.1.** Show the reduction of

$$\langle\, (((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1 \urcorner)\, , [\,]\rangle$$

to $\langle V, [\,]\rangle$ with $\longmapsto_{\mathsf{cc}}$.

▷ **Exercise 6.2.** Implement the CC machine in your favorite programming language.

6.2 SCC Machine

The CC machine faithfully immitates the textual machine by performing all of the work on the control string. Consider the following example:

$$((\lambda x.x)\ ((\lambda x.x)\ \ulcorner 5 \urcorner)),$$

consisting of an application of two applications, which evaluates as follows:

$$\begin{array}{lll} \langle ((\lambda x.x)\ ((\lambda x.x)\ \ulcorner 5 \urcorner)), [\,]\rangle & \longmapsto_{\mathsf{cc}} & \langle ((\lambda x.x)\ \ulcorner 5 \urcorner), ((\lambda x.x)\ [\,])\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle \ulcorner 5 \urcorner, ((\lambda x.x)\ [\,])\rangle \end{array}$$

At this point, the CC machine is forced to construct an application such that one of the application rules can put the second application in the control string register:

$$\begin{array}{lll} \ldots & \longmapsto_{\mathsf{cc}} & \langle((\lambda x.x)\ \ulcorner 5 \urcorner), [\,]\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle \ulcorner 5 \urcorner, [\,]\rangle. \end{array}$$

The next to last step is only needed because the CC machine only exploits information in the control string position. One way to simplify the CC machine is to combine the rules for values with the reduction of redexes such that the last two steps become one step.

A similar simplification is possible when the value returned belongs into the function position. A simple variation of the first example shows how the machine again moves information into the control string even though the next step is clear:

$$\begin{array}{lll} \langle(((\lambda x.x)\ (\lambda x.x))\ ((\lambda x.x)\ \ulcorner 5 \urcorner)), [\,]\rangle & \longmapsto_{\mathsf{cc}} & \langle((\lambda x.x)\ (\lambda x.x)), ([\,]\ ((\lambda x.x)\ \ulcorner 5 \urcorner))\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle(\lambda x.x), ([\,]\ ((\lambda x.x)\ \ulcorner 5 \urcorner))\rangle. \end{array}$$

Now the machine constructs an application only to move on to the evaluation of the argument expression:

$$\begin{array}{ll} \longmapsto_{\mathsf{cc}} & \langle((\lambda x.x)\ ((\lambda x.x)\ \ulcorner 5 \urcorner)), [\,]\rangle \\ \longmapsto_{\mathsf{cc}} & \langle((\lambda x.x)\ \ulcorner 5 \urcorner), ((\lambda x.x)\ [\,])\rangle \\ \ldots & \end{array}$$

Instead of constructing the application, the machine could always put the argument into the control register and put the value for the function position into the evaluation context. If the argument is a value, the modified machine can perform a reduction. If not, the current CC machine would have reached this state, too.

Once we have a machine that combines the recognition of a value with the reduction of a redex, when possible, we can also omit the specialized treatment of applications. The current CC machine has three distinct transitions for dealing with applications and three more for dealing with primitive applications. Side-conditions ensure that only one transition applies to any given application. These separate rules are only necessary because the machine performs all the work on the control string and uses the control stack only as a simple memory. Together with the perviously suggested simplifications, we can eliminate the side-conditions, the reduction rules, and the rule [cc2], which only exists to move arguments from evaluation contexts into the control string position.

The simplified CC machine has fewer transitions and no side-conditions. Given a state finding the applicable transition rule is easy. If the control string is an application, shift a fixed portion to the evaluation context and concentrate on one sub-expression. If it is a value, check the innermost application of the evaluation context. If the hole is in the last position of the application, perform a reduction, otherwise swap information between the control string register and the evaluation context register.

$$\begin{array}{lll} \langle(M\ N), E\rangle & \longmapsto_{\mathsf{scc}} & \langle M, E[([\,]\ N)]\rangle \quad [\mathsf{scc1}] \\ \langle(o^n\ \ M\ N \ldots), E\rangle & \longmapsto_{\mathsf{scc}} & \langle M, E[(o^n\ \ [\,]\ N \ldots)]\rangle \quad [\mathsf{scc2}] \\ \langle V, E[((\lambda X.M)\ [\,])]\rangle & \longmapsto_{\mathsf{scc}} & \langle M[X \leftarrow V], E\rangle \quad [\mathsf{scc3}] \\ \langle V, E[([\,]\ N)]\rangle & \longmapsto_{\mathsf{scc}} & \langle N, E[(V\ [\,])]\rangle \quad [\mathsf{scc4}] \\ \langle b, E[(o^n\ b_1 \ldots b_i\ [\,])]\rangle & \longmapsto_{\mathsf{scc}} & \langle V, E\rangle \quad [\mathsf{scc5}] \\ & & \text{where } \delta(o^n, b_1, \ldots, b_i, b) = V \\ \langle V, E[(o^n\ V_1 \ldots V_i\ [\,]\ N\ L \ldots)]\rangle & \longmapsto_{\mathsf{scc}} & \langle N, E[(o^n\ V_1 \ldots V_i\ V\ [\,]\ L \ldots)]\rangle \quad [\mathsf{scc6}] \end{array}$$

$$eval_{\mathsf{scc}}(M) = \begin{cases} b & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{scc}} \langle b, [\,]\rangle \\ \texttt{function} & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{scc}} \langle \lambda X.N, [\,]\rangle \end{cases}$$

The CC machine and the simplified version define the same evaluation function.

Theorem 6.3: $eval_{\mathsf{scc}} = eval_{\mathsf{cc}}$.

Proof for Theorem 6.3: The proof of this theorem is analoguous to the proof of Theorem 6.1. That is, unfolding the definition of the two evaluation functions leads to the proof obligation that

$$\langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle \qquad \text{if and only if} \qquad \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{scc}} \langle V, [\,]\rangle.$$

The correct stregthening of this claim to an inductive hypothesis extends it to arbitrary intermediate states:

$$\langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle V, [\,]\rangle \qquad \text{if and only if} \qquad \langle M, E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{scc}} \langle V, [\,]\rangle.$$

The key idea for the proof of the intermediate claim is that for any given program, the CC machine and the SCC machine quickly synchronize at some state provided the initial state leads to a result. More precisely, given $\langle M, E\rangle$ such that $E[M] = E'[L]$ and L is a **v**-redex, there exists a state $\langle N, E''\rangle$ such that

$$\langle M, E\rangle \longmapsto^{+}_{\mathsf{cc}} \langle N, E''\rangle \qquad \text{and} \qquad \langle M, E\rangle \longmapsto^{+}_{\mathsf{scc}} \langle N, E''\rangle$$

By definition, M is either a value or an expression that contains a redex. Moreover, if M is a value, the evaluation context must contain an innermost application: $E = E'[(o\ V_1 \ldots\ V_m\ [\,]\ N_1 \ldots\ N_n)]$, $E = E'[(V\ [\,])]$, or $E = E'[([\,]\ N)]$. Hence, the initial state $\langle M, E\rangle$ falls into one of the following four possibilities:

- **Case** $M = E'[L]$
  The CC machine evaluates the initial state as follows:

  $$\langle E'[L], E\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle L, E[E']\rangle \longmapsto_{\mathsf{cc}} \langle N, E[E']\rangle$$

  The SCC machine eventually reaches the same state, but the intermediate states depend on the shape of the redex L. If $L = ((\lambda X.L')\ L'')$, then the SCC machine reaches $\langle L'', E[E'[((\lambda X.L')\ [\,])]]\rangle$ as the next to last state; otherwise, the next to last state is $\langle b, E[E'[(o^n\ b_1 \ldots\ b_i\ [\,])]]\rangle$ where $L = (o^n\ b_1 \ldots b_i\ b)$.
- **Case** $M \in V, E = E'[((\lambda X.L')\ [\,])]$
  In this case, the CC machine creates a redex in the control component and reduces it:

  $$\begin{array}{rll} \langle M, E'[((\lambda X.L')\ [\,])]\rangle & \longmapsto_{\mathsf{cc}} & \langle ((\lambda X.L')\ M), E'\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle L'[X \leftarrow M], E'\rangle \end{array}$$

  The SCC machine avoids the first step and immediately goes into the same state.
- **Case** $M \in V, E = E'[([\,]\ N)]$
  If $N \in V$, the proof of this case proceeds along the lines of the second case. Otherwise,

  $$\langle M, E'[([\,]\ N)]\rangle \longmapsto_{\mathsf{cc}} \langle (M\ N), E'\rangle, \longmapsto_{\mathsf{cc}} \langle N, E'[(M\ [\,])]\rangle$$

  Again, the SCC machine skips the intermediate state.
- **Case** $M \in V, E = E'[(o^p\ V_1 \ldots V_m\ [\,]\ N_1 \ldots N_n)]$
  The transition sequences depend on the shape of the sequence $N_1, \ldots N_n$. There are three possible cases:

1. If the sequence $N_1, \ldots N_n$ is empty, $M, V_1, \ldots V_m$ must be basic constants and $\delta(o^p, V_1, \ldots V_m, V) = U$ for some value U. It follows that

$$\begin{array}{lll} \langle M, E'[(o^p\ V_1 \ldots V_n\ [\,])]\rangle & \longmapsto_{\mathsf{cc}} & \langle (o^p\ V_1 \ldots\ V_n\ M), E'\rangle \\ & \longmapsto_{\mathsf{cc}} & \langle U, E'\rangle \end{array}$$

   As in the previous case, the SCC machine reaches the final state by skiping the intermediate state.

2. If $N_1, \ldots N_n$ is not empty but $N_1, \ldots N_n \in V$, then

$$\begin{array}{l} \langle M, E'[(o^p\ V_1 \ldots\ V_m\ [\,]\ N_1 \ldots\ N_n)]\rangle \\ \quad \longmapsto_{\mathsf{cc}} \quad \langle (o^p\ V_1 \ldots\ V_m\ M\ N_1 \ldots\ N_n), E'\rangle \\ \quad \longmapsto_{\mathsf{cc}} \quad \langle U, E'\rangle. \end{array}$$

   Now, the SCC machine needs to perform a number of additional steps, which serve to verify that the additional arguments are values:

$$\begin{array}{l} \langle M, E'[(o^p\ V_1 \ldots V_m\ [\,]\ N_1 \ldots\ N_n)]\rangle \\ \quad \longmapsto_{\mathsf{scc}} \quad \langle N_1, E'[(o\ V_1 \ldots V_m\ M\ [\,]\ N_2 \ldots\ N_n)]\rangle \\ \quad \longmapsto\!\!\!\!\longmapsto_{\mathsf{scc}} \quad \ldots \\ \quad \longmapsto\!\!\!\!\longmapsto_{\mathsf{scc}} \quad \langle N_n, E'[(o^p\ V_1 \ldots\ N_{n-1}\ [\,])]\rangle \\ \quad \longmapsto_{\mathsf{scc}} \quad \langle U, E'\rangle. \end{array}$$

3. If $N_1, \ldots N_n$ is not empty and $N_i \notin V$, then both machines reach the intermediate state

$$\langle N_i, E'[(o^p\ V_1 \ldots\ V_n N_1 \ldots\ N_{i-1}\ [\,]\ N_{i+1} \ldots)]\rangle$$

   after an appropriate number of steps.

The rest of the proof is a straightforward induction on the number of transition steps. Assume $\langle M, E\rangle \longmapsto\!\!\!\!\longmapsto_{\mathsf{cc}} \langle V, [\,]\rangle$. If $M = V, E = [\,]$, the conclusion is immediate. Otherwise, $E[M]$ must contain a redex since the machine reaches a final state. By the above, there must be an intermediate state $\langle N, E'\rangle$ that both machines reach after some number of steps. By the inductive hypothesis, both machines also reach $\langle V, [\,]\rangle$ from this intermediate state. The proof of the opposite direction proceeds in the same fashion.

▷ **Exercise 6.3.** Show the reduction of

$$\langle\, (\,((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1 \urcorner)\,, [\,]\rangle$$

to $\langle V, [\,]\rangle$ with $\longmapsto_{\mathsf{scc}}$.

▷ **Exercise 6.4.** Implement the SCC machine.

6.3 CK Machine

Each step of an evaluation on the CC or simplified CC machine refers to the innermost application of the evaluation context—that is, the application that directly contains the hole. In the case that the control string is an application, a new innermost application is created in the context. In the case that the control string turns into a value, the next step is determined by considering the innermost application. In short, the transitions always access the evaluation

context from the inside and in a last-in, first-out fashion. Transition steps depend on the precise shape of the first element but not on the rest of the data structure.

The observation suggests that the evaluation context register should be a list of applications with a hole, so that the innermost part is easily accessed or added at the front of the list:

$$\begin{array}{rcl} \kappa & = & \kappa_s \ldots \\ \kappa_s & = & (V\ [\,]) \\ & | & ([\,]\ N) \\ & | & (o^n\ V \ldots\ V\ [\,]\ N \ldots) \end{array}$$

For readability and for ease of implementation, we tag each case of this construction and invert the list of values in a primitive application:

$$\begin{array}{rcl} \kappa & = & \mathtt{mt} \\ & | & \langle \mathtt{fun}, V, \kappa \rangle \\ & | & \langle \mathtt{arg}, N, \kappa \rangle \\ & | & \langle \mathtt{opd}, \langle V, \ldots, V, o^n \rangle, \langle N, \ldots \rangle, \kappa \rangle \end{array}$$

This data structure is called a **continuation**. The $\longmapsto_{\mathsf{ck}}$ relation maps an expression–continuation pair to a new expression–continuation pair.

$$\begin{array}{lcll} \langle (M\ N), \kappa \rangle & \longmapsto_{\mathsf{ck}} & \langle M, \langle \mathtt{arg}, N, \kappa \rangle \rangle & [\mathsf{ck1}] \\ \langle (o^n\ M\ N \ldots), \kappa \rangle & \longmapsto_{\mathsf{ck}} & \langle M, \langle \mathtt{opd}, \langle o^n \rangle, \langle N, \ldots \rangle, \kappa \rangle \rangle & [\mathsf{ck2}] \\ \langle V, \langle \mathtt{fun}, (\lambda X.M), \kappa \rangle \rangle & \longmapsto_{\mathsf{ck}} & \langle M[X \leftarrow V], \kappa \rangle & [\mathsf{ck3}] \\ \langle V, \langle \mathtt{arg}, N, \kappa \rangle \rangle & \longmapsto_{\mathsf{ck}} & \langle N, \langle \mathtt{fun}, V, \kappa \rangle \rangle & [\mathsf{ck4}] \\ \langle b, \langle \mathtt{opd}, \langle b_i, \ldots b_1, o^n \rangle, \langle \rangle, \kappa \rangle \rangle & \longmapsto_{\mathsf{ck}} & \langle V, \kappa \rangle & [\mathsf{ck5}] \\ & & \text{where } \delta(o^n, b_1, \ldots, b_i, b) = V & \\ \langle V, \langle \mathtt{opd}, \langle V', \ldots o^n \rangle, \langle N, L, \ldots \rangle, \kappa \rangle \rangle & \longmapsto_{\mathsf{ck}} & \langle N, \langle \mathtt{opd}, \langle V,\ V', \ldots o^n \rangle, \langle L, \ldots \rangle, \kappa \rangle \rangle & [\mathsf{ck6}] \end{array}$$

$$eval_{\mathsf{ck}}(M) = \begin{cases} b & \text{if } \langle M, \mathtt{mt} \rangle \longmapsto\!\!\!\!\!\rightarrow_{\mathsf{ck}} \langle b, \mathtt{mt} \rangle \\ \mathtt{function} & \text{if } \langle M, \mathtt{mt} \rangle \longmapsto\!\!\!\!\!\rightarrow_{\mathsf{ck}} \langle \lambda X.N, \mathtt{mt} \rangle \end{cases}$$

Given an SCC state, we can find a corresponding CK state and vice versa, based on a bijection between the set of evaluation contexts and the set of continuation codes:

$$\mathcal{T}_{\mathsf{ck}\to\mathsf{scc}}(\kappa) = \begin{cases} [\,] & \text{if } \kappa = \mathtt{mt} \\ E'[([\,]\ N)] & \text{if } \kappa = \langle \mathtt{arg}, N, \kappa' \rangle \\ E'[(V\ [\,])] & \text{if } \kappa = \langle \mathtt{fun}, V, \kappa' \rangle \\ E'[(o^n\ V_1 \ldots\ V_i\ [\,]\ N \ldots)] & \text{if } \kappa = \langle \mathtt{opd}, \langle V_i, \ldots V_1, o^n \rangle, \langle N, \ldots \rangle, \kappa' \rangle \end{cases}$$
where $E' = \mathcal{T}_{\mathsf{ck}\to\mathsf{scc}}(\kappa')$

$$\mathcal{T}_{\mathsf{scc}\to\mathsf{ck}}(E) = \begin{cases} \mathtt{mt} & \text{if } E = [\,] \\ \langle \mathtt{arg}, N, \kappa' \rangle & \text{if } E = E'[([\,]\ N)] \\ \langle \mathtt{fun}, V, \kappa' \rangle & \text{if } E = E'[(V\ [\,])] \\ \langle \mathtt{opd}, \langle V_i, \ldots, V_1, o^n \rangle, \langle N, \ldots \rangle, \kappa' \rangle & \text{if } E = E'[(o^n\ V_1 \ldots\ V_i\ [\,]\ N \ldots)] \end{cases}$$
where $\kappa' = \mathcal{T}_{\mathsf{scc}\to\mathsf{ck}}(E')$

Based on this bijection between CC and CK states, it is easy to prove the correctness of the evaluation function based on the CK machine.

Theorem 6.4: $eval_{\mathsf{ck}} = eval_{\mathsf{scc}}$.

Proof for Theorem 6.4: Based on the translations between states, we can check that

1. if $\langle M, E\rangle \longmapsto_{\mathsf{scc}} \langle N, E'\rangle$, then $\langle M, \mathcal{T}_{\mathsf{scc}\to\mathsf{ck}}(E)\rangle \longmapsto_{\mathsf{ck}} \langle N, \mathcal{T}_{\mathsf{scc}\to\mathsf{ck}}(E')\rangle$;
2. if $\langle M, \kappa\rangle \longmapsto_{\mathsf{ck}} \langle N, \kappa'\rangle$, then $\langle M, \mathcal{T}_{\mathsf{ck}\to\mathsf{scc}}(\kappa)\rangle \longmapsto_{\mathsf{scc}} \langle N, \mathcal{T}_{\mathsf{ck}\to\mathsf{scc}}(\kappa')\rangle$.

By trivial inductions on the length of the transition sequences, it follows that

$$\langle M, [\,]\rangle \longmapsto\!\!\!\!\to_{\mathsf{scc}} \langle V, [\,]\rangle \quad \text{if and only if} \quad \langle M, \mathtt{mt}\rangle \longmapsto\!\!\!\!\to_{\mathsf{ck}} \langle V, \mathtt{mt}\rangle,$$

which implies the theorem.

▷ **Exercise 6.5.** Show the reduction of

$$\langle\, (\,((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1\urcorner\,)\,, \mathtt{mt}\rangle$$

to $\langle V, \mathtt{mt}\rangle$ with $\longmapsto_{\mathsf{ck}}$.

▷ **Exercise 6.6.** Implement the CK machine.

6.4 CEK Machine

For all of the machines so far, the reduction of a β_{v}-redex requires the substitution of a value in place of all occurrences of an identifier. A naive implementation of this substituion would be expensive, especially as expressions become large. Furthermore, in the case that substitution procedures a non-value expression, the machine will imemdiately re-visit sub-expressions (to evaluate them) that were just visited by substituion.

Unlike the simplication we found for evaluation contexts, we cannot avoid substitution traversals and re-traversals by turning the expression inside-out. We can, however, delay the work of substituion until it is actually needed.

To represent delayed substitutions, every expression in the machine is replaced by a pair whose first component is an open term and whose second component is a function that maps all free variables to their correponding expressions. The pair is called a **closure**, and the function component is an **environment**.

Of course, environments cannot map variables to expressions, because this would re-introduce expressions into the machine; it must map variables to closures. Thus, environments $\mathcal{E}$ and closures c and v have mututally recursive definitions:

$$\begin{array}{lcl}
\mathcal{E} & = & \text{a function } \{\langle X, c\rangle, \ldots\} \\
c & = & \{\langle M, \mathcal{E}\rangle \mid \mathcal{FV}(M) \subset \mathrm{dom}(\mathcal{E})\} \\
v & = & \{\langle V, \mathcal{E}\rangle \mid \langle V, \mathcal{E}\rangle \in c\} \\
 & & \\
\mathcal{E}[X \leftarrow c] & = & \{\langle X, c\rangle\} \cup \{\langle Y, c'\rangle \mid \langle Y, c'\rangle \in \mathcal{E} \text{ and } Y \neq X\}
\end{array}$$

Based these defininitions, we can reformulate the CK machine into a machine that works on control strings with environments and continuation codes: the CEK machine. Continuations in this machine contain closures instead of expressions.

$$\begin{array}{rcl}
\overline{\kappa} & = & \mathtt{mt} \\
 & \mid & \langle \mathtt{fun}, v, \overline{\kappa}\rangle \\
 & \mid & \langle \mathtt{arg}, c, \overline{\kappa}\rangle \\
 & \mid & \langle \mathtt{opd}, \langle v, \ldots, v, o^n\rangle, \langle c, \ldots\rangle, \overline{\kappa}\rangle
\end{array}$$

The transitions are similar to CK transitions, except the expressions are replaced with closures, and substitution is replaced by environment update.

$$
\begin{array}{lll}
\langle\langle (M\ N), \mathcal{E}\rangle, \overline{\kappa}\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}\rangle, \langle \mathtt{arg}, \langle N, \mathcal{E}\rangle, \overline{\kappa}\rangle\rangle \quad \text{[cek1]} \\
\langle\langle (o^n\ M\ N \ldots), \mathcal{E}\rangle, \overline{\kappa}\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}\rangle, \langle \mathtt{opd}, \langle o^n\rangle, \langle\langle N, \mathcal{E}\rangle, \ldots\rangle, \overline{\kappa}\rangle\rangle \quad \text{[cek2]} \\
\langle\langle V, \mathcal{E}\rangle, \langle \mathtt{fun}, \langle (\lambda X_1.M), \mathcal{E}'\rangle, \overline{\kappa}\rangle\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}'[X_1 \leftarrow \langle V, \mathcal{E}\rangle]\rangle, \overline{\kappa}\rangle \quad \text{[cek3]} \\
\quad \text{if } V \notin X & & \\
\langle\langle V, \mathcal{E}\rangle, \langle \mathtt{arg}, \langle N, \mathcal{E}'\rangle, \overline{\kappa}\rangle\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle N, \mathcal{E}'\rangle, \langle \mathtt{fun}, \langle V, \mathcal{E}\rangle, \overline{\kappa}\rangle\rangle \quad \text{[cek4]} \\
\quad \text{if } V \notin X & & \\
\langle\langle b, \mathcal{E}\rangle, & \longmapsto_{\mathsf{cek}} & \langle\langle V, \emptyset\rangle, \overline{\kappa}\rangle \quad \text{[cek5]} \\
\ \langle \mathtt{opd}, \langle\langle b_i, \mathcal{E}_i\rangle, \ldots \langle b_1, \mathcal{E}_1\rangle, o^n\rangle, \langle\rangle, \overline{\kappa}\rangle\rangle & & \\
 & & \quad \text{where } \delta(o^n, b_1, \ldots, b_i, b) = V \\
\langle\langle V, \mathcal{E}\rangle, & \longmapsto_{\mathsf{cek}} & \langle\langle N, \mathcal{E}'\rangle, \quad \text{[cek6]} \\
\ \langle \mathtt{opd}, \langle v', \ldots o^n\rangle, \langle\langle N, \mathcal{E}'\rangle, c, \ldots\rangle, \overline{\kappa}\rangle\rangle & & \ \langle \mathtt{opd}, \langle\langle V, \mathcal{E}\rangle, v', \ldots o^n\rangle, \langle c, \ldots\rangle, \overline{\kappa}\rangle\rangle \\
\quad \text{if } V \notin X & & \\
\langle\langle X, \mathcal{E}\rangle, \overline{\kappa}\rangle & \longmapsto_{\mathsf{cek}} & \langle c, \overline{\kappa}\rangle \quad \text{[cek7]} \\
 & & \quad \text{where } \mathcal{E}(X) = c
\end{array}
$$

$$
eval_{\mathsf{cek}}(M) = \begin{cases} b & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle\langle b, \mathcal{E}\rangle, \mathtt{mt}\rangle \\ \mathtt{function} & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle\langle \lambda X.N, \mathcal{E}\rangle, \mathtt{mt}\rangle \end{cases}
$$

Given a CEK state, we can find a corresponding CK state with an "unload" function that replaces every closure in the state by the closed term that it represents:

$$
\mathcal{U}(\langle M, \{\langle X_1, c_1\rangle, \ldots \langle X_n, c_n\rangle\}\rangle) \quad = \quad M[X_1 \leftarrow \mathcal{U}(c_1)] \ldots [X_n \leftarrow \mathcal{U}(c_n)]
$$

Since closures also occur inside of CEK continuation codes, the translation of CEK states into CK states requires an additional traversal of CEK continuation codes to replace these closures:

$$
\mathcal{T}_{\mathsf{cek}\rightarrow\mathsf{ck}}(\overline{\kappa}) = \begin{cases} [\,] & \text{if } \overline{\kappa} = \mathtt{mt} \\ \langle \mathtt{arg}, \mathcal{U}(c), \kappa^*\rangle & \text{if } \overline{\kappa} = \langle \mathtt{arg}, c, \overline{\kappa}'\rangle \\ \langle \mathtt{fun}, \mathcal{U}(c), \kappa^*\rangle & \text{if } \overline{\kappa} = \langle \mathtt{fun}, c, \overline{\kappa}'\rangle \\ \langle \mathtt{opd}, \langle \mathcal{U}(c'), \ldots o^n\rangle, \langle \mathcal{U}(c), \ldots\rangle, \kappa^*\rangle & \\ \qquad \text{if } \overline{\kappa} = \langle \mathtt{opd}, \langle c', \ldots o^n\rangle, \langle c, \ldots\rangle, \overline{\kappa}'\rangle & \end{cases}
$$
$$
\text{where } \kappa^* = \mathcal{T}_{\mathsf{cek}\rightarrow\mathsf{ck}}(\overline{\kappa}')
$$

Unfortunately, it is impossible to define inverses of the $\mathcal{U}$ and $\mathcal{T}_{\mathsf{cek}\rightarrow\mathsf{ck}}$ functions; given a closed term, there are many closures that map back to the same term. As a result, the simple proof method of the preceding adequacy theorems for verifying the equality between the SCC evaluator and the CK evaluator fails here. However, as in the preceding sections, it is still the case that the CEK machine can simulate any given sequence of CK states if the initial state of the two machines are equivalent.

Theorem 6.5: $eval_{\mathsf{cek}} = eval_{\mathsf{ck}}$.

Proof for Theorem 6.5: Given the translation from CEK states, we can check that for any intermediate state $\langle c, \overline{\kappa}\rangle$ in the sequence of CEK states, there is a state $\langle c', \overline{\kappa}'\rangle$ such that

$$
\langle c, \overline{\kappa}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle c', \overline{\kappa}'\rangle
$$

and

$$
\langle \mathcal{U}(c), \mathcal{T}_{\mathsf{cek}\rightarrow\mathsf{ck}}(\overline{\kappa})\rangle \longmapsto_{\mathsf{ck}} \langle \mathcal{U}(c', \mathcal{T}_{\mathsf{cek}\rightarrow\mathsf{ck}}(\overline{\kappa}'))\rangle
$$

Hence, $\langle\langle M, \emptyset\rangle, \mathtt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle\langle V, \mathcal{E}\rangle, \mathtt{mt}\rangle$ implies $\langle M, \mathtt{mt}\rangle \longmapsto_{\mathsf{ck}} \langle \mathcal{U}(\langle V, \mathcal{E}\rangle), \mathtt{mt}\rangle$, which proves the left-to-right direction.

The right-to-left direction requires a stronger induction hypothesis than the left-to-right direction or the inductions of the preceding adequacy theorems because of the lack of a function from CK states to CEK states. In addition to the intial state of the transition sequence of the CK machine, we need to know the initial state of the CEK machine:

> For every CK state $\langle M_1, \kappa_1 \rangle$ and CEK state $\langle \langle M_1', \mathcal{E} \rangle, \overline{\kappa}_1' \rangle$ such that $M_1 = \mathcal{U}(\langle M_1', \mathcal{E} \rangle)$ and $\kappa_1 = \mathcal{T}_{\mathsf{cek}\to\mathsf{ck}}(\overline{\kappa}_1')$, if
>
> $$\langle M_1, \kappa_1 \rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{ck}} \langle V, \mathtt{mt} \rangle$$
>
> then for some closure cl with $\mathcal{U}(c) = V$,
>
> $$\langle \langle M_1', \mathcal{E} \rangle, \overline{\kappa}_1' \rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle c, \mathtt{mt} \rangle$$

The theorem follows from specializing the two initial states to $\langle M, \mathtt{mt} \rangle$ and $\langle \langle M, \emptyset \rangle, \mathtt{mt} \rangle$.

The proof of the claim is by induction on the length of the transition sequence in the CK machine. It proceeds by case analysis of the initial state of the CK machine:

- **Case** $M_1 = (L\ N)$
  The CK machine performs a [ck1] instruction:

  $$\langle (L\ N), \kappa_1 \rangle \longmapsto_{\mathsf{ck}} \langle L, \langle \mathtt{arg}, N, \kappa_1 \rangle \rangle$$

  By assumption, $\langle M_1', \mathcal{E} \rangle = \langle (L'\ N'), \mathcal{E} \rangle$ such that $L = \mathcal{U}(\langle L', \mathcal{E} \rangle)$ and $N = \mathcal{U}(\langle N', \mathcal{E} \rangle)$ since $\mathcal{U}$ preserves the shape of applications in the translation process. Hence,

  $$\langle \langle (L'\ N'), \mathcal{E} \rangle, \overline{\kappa}_1' \rangle \longmapsto_{\mathsf{ck}} \langle \langle L', \mathcal{E} \rangle, \langle \mathtt{arg}, \langle N', \mathcal{E} \rangle, \overline{\kappa}_1' \rangle \rangle.$$

  Clearly, $\mathcal{T}_{\mathsf{cek}\to\mathsf{ck}}(\langle \mathtt{arg}, \langle N', \mathcal{E} \rangle, \overline{\kappa}_1' \rangle) = \langle \mathtt{arg}, N, \kappa_1 \rangle$. Thus the inductive hypothesis applies and proves the claim for this case.

- **Case** $M_1 \in V, \kappa_1 = \langle \mathtt{fun}, (\lambda X.N), \kappa_2 \rangle$
  Here the CK transition sequence begins as follows:

  $$\langle M_1, \langle \mathtt{fun}, (\lambda X.N), \kappa_2 \rangle \rangle \longmapsto_{\mathsf{ck}} \langle N[X \leftarrow M_1], \kappa_2 \rangle$$

  Given the definition of $\mathcal{U}$, there are two kinds of CEK states that can map to the given CK state:

  - **Case** $M_1' \notin X$
    In this case, the CEK step directly corresponds to the CK step:

    $$\begin{array}{l} \langle \langle M_1', \mathcal{E} \rangle, \langle \mathtt{fun}, \langle (\lambda X.N), \mathcal{E}' \rangle, \overline{\kappa}_2 \rangle \rangle \\ \quad \longmapsto_{\mathsf{cek}} \quad \langle \langle N, \mathcal{E}'[X \leftarrow \langle M_1', \mathcal{E} \rangle] \rangle, \overline{\kappa}_2 \rangle \end{array}$$

  - **Case** $M_1' \in X$**,** $\mathcal{U}(\mathcal{E}(M_1')) = M_1$
    If the value is a variable, the CEK machine first looks up the value of the variable in the environment before it performs an appropriate transition:

    $$\begin{array}{l} \langle \langle M_1', \mathcal{E} \rangle, \langle \mathtt{fun}, \langle (\lambda X.N'), \mathcal{E}' \rangle, \overline{\kappa}_2 \rangle \rangle \\ \quad \longmapsto_{\mathsf{cek}} \quad \langle \mathcal{E}(M_1'), \langle \mathtt{fun}, \langle (\lambda X.N), \mathcal{E}' \rangle, \overline{\kappa}_2 \rangle \rangle \\ \quad \longmapsto_{\mathsf{cek}} \quad \langle \langle N, \mathcal{E}'[X \leftarrow \mathcal{E}(M_1')] \rangle, \overline{\kappa}_2 \rangle \end{array}$$

In both cases, the assumptions state $\mathcal{U}(\langle M_1', \mathcal{E}\rangle) = M_1$. Hence, to finish the case, we must show that

$$\mathcal{U}(\langle N', \mathcal{E}'[X \leftarrow c]\rangle) = N[X \leftarrow \mathcal{U}(c)]$$

where $cl = \langle M_1', \mathcal{E}\rangle$. From the definition of $\mathcal{U}$, if $\mathcal{E} = \{\langle X_1, c_1\rangle, \ldots, \langle X_1, c_1\rangle\}$,

$$\mathcal{U}(\langle \lambda X.N', \mathcal{E}'\rangle) = \lambda X.N'[X_1 \leftarrow \mathcal{U}(c_1)] \ldots [X_n \leftarrow \mathcal{U}(c_n)] = \lambda X.N,$$

which implies

$$N'[X_1 \leftarrow \mathcal{U}(c_1)] \ldots [X_n \leftarrow \mathcal{U}(c_n)] = N.$$

By the Substitution Lemma,

$$\begin{array}{rcl} N[X \leftarrow \mathcal{U}(c)] & = & N'[X_1 \leftarrow \mathcal{U}(c_1)] \ldots [X_n \leftarrow \mathcal{U}(c_n)][X \leftarrow \mathcal{U}(c)] \\ & = & \mathcal{U}(\langle N', \mathcal{E}[X \leftarrow c]\rangle). \end{array}$$

The rest follows again by induction on the length of the CK transition sequence.

The proof of the other cases are analoguous to the last one.

▷ **Exercise 6.7.** Show the reduction of

$$\langle\langle\,(\,((\lambda f.\lambda x.f\ x)\ \lambda y.(+\ y\ y))\ \ulcorner 1 \urcorner\,)\,, \emptyset\rangle, \mathtt{mt}\rangle$$

to $\langle\langle V, \mathcal{E}\rangle, \mathtt{mt}\rangle$ with $\longmapsto_{\mathsf{cek}}$.

▷ **Exercise 6.8.** Implement the CEK machine.

6.5 Machine Summary

CC machine

$$
\begin{array}{lcll}
\langle (M\ N), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E[([\,]\ N)]\rangle & [\mathsf{cc1}] \\
\quad \text{if } M \notin V & & & \\
\langle (V_1\ M), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E[(V_1\ [\,])]\rangle & [\mathsf{cc2}] \\
\quad \text{if } M \notin V & & & \\
\langle (o^n\ V_1 \ldots V_i\ M\ N \ldots), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E[(o^n\ V_1 \ldots V_i\ [\,]\ N \ldots)]\rangle & [\mathsf{cc3}] \\
\quad \text{if } M \notin V & & & \\
\langle ((\lambda X.M)\ V), E\rangle & \longmapsto_{\mathsf{cc}} & \langle M[X \leftarrow V], E\rangle & [\mathsf{cc}\beta_{\mathsf{v}}] \\
\langle (o^m\ b_1 \ldots\ b_m), E\rangle & \longmapsto_{\mathsf{cc}} & \langle V, E\rangle & [\mathsf{cc}\delta] \\
 & & \quad \text{where } V = \delta(o^m, b_1, \ldots b_m) & \\
\langle V, E[(U\ [\,])]\rangle & \longmapsto_{\mathsf{cc}} & \langle (U\ V), E\rangle & [\mathsf{cc4}] \\
\langle V, E[([\,]\ N)]\rangle & \longmapsto_{\mathsf{cc}} & \langle (V\ N), E\rangle & [\mathsf{cc5}] \\
\langle V, E[(o^n\ V_1 \ldots V_i\ [\,]\ N \ldots)]\rangle & \longmapsto_{\mathsf{cc}} & \langle (o^n\ V_1 \ldots V_i\ V\ N \ldots), E\rangle & [\mathsf{cc6}]
\end{array}
$$

$$
eval_{\mathsf{cc}}(M) = \begin{cases} b & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle b, [\,]\rangle \\ \texttt{function} & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cc}} \langle \lambda X.N, [\,]\rangle \end{cases}
$$

SCC machine

$$
\begin{array}{lcll}
\langle (M\ N), E\rangle & \longmapsto_{\mathsf{scc}} & \langle M, E[([\,]\ N)]\rangle & [\mathsf{scc1}] \\
\langle (o^n\ \ M\ N \ldots), E\rangle & \longmapsto_{\mathsf{scc}} & \langle M, E[(o^n\ \ [\,]\ N \ldots)]\rangle & [\mathsf{scc2}] \\
\langle V, E[((\lambda X.M)\ [\,])]\rangle & \longmapsto_{\mathsf{scc}} & \langle M[X \leftarrow V], E\rangle & [\mathsf{scc3}] \\
\langle V, E[([\,]\ N)]\rangle & \longmapsto_{\mathsf{scc}} & \langle N, E[(V\ [\,])]\rangle & [\mathsf{scc4}] \\
\langle b, E[(o^n\ b_1 \ldots b_i\ [\,])]\rangle & \longmapsto_{\mathsf{scc}} & \langle V, E\rangle & [\mathsf{scc5}] \\
 & & \quad \text{where } \delta(o^n, b_1, \ldots, b_i, b) = V & \\
\langle V, E[(o^n\ V_1 \ldots V_i\ [\,]\ N\ L \ldots)]\rangle & \longmapsto_{\mathsf{scc}} & \langle N, E[(o^n\ V_1 \ldots V_i\ V\ [\,]\ L \ldots)]\rangle & [\mathsf{scc6}]
\end{array}
$$

$$
eval_{\mathsf{scc}}(M) = \begin{cases} b & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{scc}} \langle b, [\,]\rangle \\ \texttt{function} & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{scc}} \langle \lambda X.N, [\,]\rangle \end{cases}
$$

CK machine

$$
\begin{array}{rcl}
\kappa & = & \texttt{mt} \\
 & | & \langle \texttt{fun}, V, \kappa\rangle \\
 & | & \langle \texttt{arg}, N, \kappa\rangle \\
 & | & \langle \texttt{opd}, \langle V, \ldots, V, o^n\rangle, \langle N, \ldots\rangle, \kappa\rangle
\end{array}
$$

$$
\begin{array}{lcll}
\langle (M\ N), \kappa\rangle & \longmapsto_{\mathsf{ck}} & \langle M, \langle \texttt{arg}, N, \kappa\rangle\rangle & [\mathsf{ck1}] \\
\langle (o^n\ M\ N \ldots), \kappa\rangle & \longmapsto_{\mathsf{ck}} & \langle M, \langle \texttt{opd}, \langle o^n\rangle, \langle N, \ldots\rangle, \kappa\rangle\rangle & [\mathsf{ck2}] \\
\langle V, \langle \texttt{fun}, (\lambda X.M), \kappa\rangle\rangle & \longmapsto_{\mathsf{ck}} & \langle M[X \leftarrow V], \kappa\rangle & [\mathsf{ck3}] \\
\langle V, \langle \texttt{arg}, N, \kappa\rangle\rangle & \longmapsto_{\mathsf{ck}} & \langle N, \langle \texttt{fun}, V, \kappa\rangle\rangle & [\mathsf{ck4}] \\
\langle b, \langle \texttt{opd}, \langle b_i, \ldots b_1, o^n\rangle, \langle\rangle, \kappa\rangle\rangle & \longmapsto_{\mathsf{ck}} & \langle V, \kappa\rangle & [\mathsf{ck5}] \\
 & & \quad \text{where } \delta(o^n, b_1, \ldots, b_i, b) = V & \\
\langle V, \langle \texttt{opd}, \langle V', \ldots o^n\rangle, \langle N, L, \ldots\rangle, \kappa\rangle\rangle & \longmapsto_{\mathsf{ck}} & \langle N, \langle \texttt{opd}, \langle V,\ V', \ldots o^n\rangle, \langle L, \ldots\rangle, \kappa\rangle\rangle & [\mathsf{ck6}]
\end{array}
$$

$$
eval_{\mathsf{ck}}(M) = \begin{cases} b & \text{if } \langle M, \texttt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{ck}} \langle b, \texttt{mt}\rangle \\ \texttt{function} & \text{if } \langle M, \texttt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{ck}} \langle \lambda X.N, \texttt{mt}\rangle \end{cases}
$$

CEK Machine

$$
\begin{array}{lcl}
\mathcal{E} & = & \text{a function } \{\langle X, c\rangle, \ldots\} \\
c & = & \{\langle M, \mathcal{E}\rangle \,|\, \mathcal{FV}(M) \subset \mathrm{dom}(\mathcal{E})\} \\
v & = & \{\langle V, \mathcal{E}\rangle \,|\, \langle V, \mathcal{E}\rangle \in c\}
\end{array}
$$

$$
\mathcal{E}[X \leftarrow c] \;=\; \{\langle X, c\rangle\} \cup \{\langle Y, c'\rangle \,|\, \langle Y, c'\rangle \in \mathcal{E} \text{ and } Y \neq X\}
$$

$$
\begin{array}{rcl}
\overline{\kappa} & = & \mathtt{mt} \\
& | & \langle \mathtt{fun}, v, \overline{\kappa}\rangle \\
& | & \langle \mathtt{arg}, c, \overline{\kappa}\rangle \\
& | & \langle \mathtt{opd}, \langle v, \ldots, v, o^n\rangle, \langle c, \ldots\rangle, \overline{\kappa}\rangle
\end{array}
$$

$$
\begin{array}{lcll}
\langle\langle (M\ N), \mathcal{E}\rangle, \overline{\kappa}\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}\rangle, \langle \mathtt{arg}, \langle N, \mathcal{E}\rangle, \overline{\kappa}\rangle\rangle & [\mathsf{cek1}] \\
\langle\langle (o^n\ M\ N \ldots), \mathcal{E}\rangle, \overline{\kappa}\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}\rangle, \langle \mathtt{opd}, \langle o^n\rangle, \langle\langle N, \mathcal{E}\rangle, \ldots\rangle, \overline{\kappa}\rangle\rangle & [\mathsf{cek2}] \\
\langle\langle V, \mathcal{E}\rangle, \langle \mathtt{fun}, \langle (\lambda X_1.M), \mathcal{E}'\rangle, \overline{\kappa}\rangle\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}'[X_1 \leftarrow \langle V, \mathcal{E}\rangle]\rangle, \overline{\kappa}\rangle & [\mathsf{cek3}] \\
\quad \text{if } V \notin X & & & \\
\langle\langle V, \mathcal{E}\rangle, \langle \mathtt{arg}, \langle N, \mathcal{E}'\rangle, \overline{\kappa}\rangle\rangle & \longmapsto_{\mathsf{cek}} & \langle\langle N, \mathcal{E}'\rangle, \langle \mathtt{fun}, \langle V, \mathcal{E}\rangle, \overline{\kappa}\rangle\rangle & [\mathsf{cek4}] \\
\quad \text{if } V \notin X & & & \\
\langle\langle b, \mathcal{E}\rangle, & \longmapsto_{\mathsf{cek}} & \langle\langle V, \emptyset\rangle, \overline{\kappa}\rangle & [\mathsf{cek5}] \\
\ \langle \mathtt{opd}, \langle\langle b_i, \mathcal{E}_i\rangle, \ldots \langle b_1, \mathcal{E}_1\rangle, o^n\rangle, \langle\rangle, \overline{\kappa}\rangle\rangle & & \quad \text{where } \delta(o^n, b_1, \ldots, b_i, b) = V & \\
\langle\langle V, \mathcal{E}\rangle, & \longmapsto_{\mathsf{cek}} & \langle\langle N, \mathcal{E}'\rangle, & [\mathsf{cek6}] \\
\ \langle \mathtt{opd}, \langle c', \ldots o^n\rangle, \langle\langle N, \mathcal{E}'\rangle, c, \ldots\rangle, \overline{\kappa}\rangle\rangle & & \ \langle \mathtt{opd}, \langle\langle V, \mathcal{E}\rangle, c', \ldots o^n\rangle, \langle c, \ldots\rangle, \overline{\kappa}\rangle\rangle & \\
\quad \text{if } V \notin X & & & \\
\langle\langle X, \mathcal{E}\rangle, \overline{\kappa}\rangle & \longmapsto_{\mathsf{cek}} & \langle c, \overline{\kappa}\rangle & [\mathsf{cek7}] \\
& & \quad \text{where } \mathcal{E}(X) = c &
\end{array}
$$

$$
eval_{\mathsf{cek}}(M) = \begin{cases} b & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle\langle b, \mathcal{E}\rangle, \mathtt{mt}\rangle \\ \mathtt{function} & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}} \langle\langle \lambda X.N, \mathcal{E}\rangle, \mathtt{mt}\rangle \end{cases}
$$

Chapter 7: SECD, Tail Calls, and Safe for Space

We derived the CEK machine by starting with the equational system $=_{\mathbf{v}}$, then refining it step-by-step to eliminate aspects of the reduction rules that were not easily mapped to small and efficient sequences of processor instructions. With each refinement step, we were able to prove that the new evaluator was equivalent to the old evaluator. Consequently, $eval_{\mathbf{v}} = eval_{\mathsf{cek}}$, so our theorems about the original evaluator hold for the CEK machine.

In this chapter, we consider aspects of evaluation beyond the simple result of the $eval_{\mathbf{v}}$ function. Since we are trying to work towards a realistic implmentation of ISWIM, we must consider resource-usage issues as well as efficiency issues. For example, Ω not only loops forever, but it does so in $\rightarrow_{\mathbf{v}}$ without ever growing. In other words, Ω corresponds to a Java `for` loop, as opposed to a Java method that always calls itselfs. The difference is that the self-calling Java method will eventually exhaust the stack.

To clarify this point, we embark on a slight detour: we consider a different machine formulation of ISWIM, called the SECD machine, that evaluates expressions to the same result as $eval_{\mathbf{v}}$, but which has different resource-usage properties. We then consider the implications of the CEK machine's resource usage for a programmer, and the implications of the mechanism. In particular, we discuss the notion of tail calls, and of continuations as values.

7.1 SECD machine

The **SECD machine** does not work directly on source ISWIM expressions. Instead, to use the SECD machine, we must first compile ISWIM source expressions to a sequence of "bytecodes" that constitute a control string $\hat{C}$:

$$
\begin{array}{rcl}
\hat{C} & = & \epsilon \\
 & | & b\ \hat{C} \\
 & | & X\ \hat{C} \\
 & | & \mathtt{ap}\ \hat{C} \\
 & | & \mathtt{prim}_{o^n}\ \hat{C} \\
 & | & \langle X, \hat{C} \rangle\ \hat{C}
\end{array}
$$

$$
\begin{array}{lcl}
[\![b]\!]_{\mathsf{secd}} & = & b \\
[\![X]\!]_{\mathsf{secd}} & = & X \\
[\![(M_1\ M_2)]\!]_{\mathsf{secd}} & = & [\![M_1]\!]_{\mathsf{secd}}\ [\![M_2]\!]_{\mathsf{secd}}\ \mathtt{ap} \\
[\![(o^n\ M_1 \ldots\ M_n)]\!]_{\mathsf{secd}} & = & [\![M_1]\!]_{\mathsf{secd}} \ldots\ [\![M_n]\!]_{\mathsf{secd}}\ \mathtt{prim}_{o^n} \\
[\![(\lambda X.M)]\!]_{\mathsf{secd}} & = & \langle X, [\![M]\!]_{\mathsf{secd}} \rangle
\end{array}
$$

Each evaluation step for the SECD machine applies to a tuple of four elements:

$\hat{S}$	a stack for values $\hat{V}$
$\hat{\mathcal{E}}$	an environment binding variables X to values $\hat{V}$
$\hat{C}$	a control string
$\hat{D}$	a "dump" representing a saved state: $\langle \hat{S}, \hat{\mathcal{E}}, \hat{C}, \hat{D} \rangle$

In the terms of a conventional imperative language (e.g., Java, C), the $\hat{S}$ part corresponds to the method- or procedure-local stack, and the $\hat{D}$ part is the rest of the stack. The sets $\hat{S}$, $\hat{\mathcal{E}}$,

$\hat{D}$, and $\hat{V}$ are defined as follows:

$$\begin{array}{rcl}
\hat{S} & = & \epsilon \\
 & | & \hat{V}\ S \\
\hat{\mathcal{E}} & = & \text{a function } \{\langle X, \hat{V}\rangle, \ldots\} \\
\hat{D} & = & \epsilon \\
 & | & \langle \hat{S}, \hat{\mathcal{E}}, \hat{C}, \hat{D}\rangle \\
\hat{V} & = & b \\
 & | & \langle\langle X, \hat{C}\rangle, \hat{\mathcal{E}}\rangle
\end{array}$$

The $\longmapsto_{\mathsf{secd}}$ relation defines a one-step evaluation on SECD states:

$$\begin{array}{lcll}
\langle \hat{S}, \hat{\mathcal{E}}, b\ \hat{C}, \hat{D}\rangle & \longmapsto_{\mathsf{secd}} & \langle b\ \hat{S}, \hat{\mathcal{E}}, \hat{C}, \hat{D}\rangle & [\mathsf{secd1}] \\
\langle \hat{S}, \hat{\mathcal{E}}, X\ \hat{C}, \hat{D}\rangle & \longmapsto_{\mathsf{secd}} & \langle \hat{V}\ \hat{S}, \hat{\mathcal{E}}, \hat{C}, \hat{D}\rangle & [\mathsf{secd2}] \\
 & & \quad \text{where } \hat{V} = \hat{\mathcal{E}}(X) & \\
\langle b_n\ \ldots\ b_1\ \hat{S}, \hat{\mathcal{E}}, \mathtt{prim}_{o^n}\ \hat{C}, \hat{D}\rangle & \longmapsto_{\mathsf{secd}} & \langle \hat{V}\ \hat{S}, \hat{\mathcal{E}}, \hat{C}, \hat{D}\rangle\rangle & [\mathsf{secd3}] \\
 & & \quad \text{where } \hat{V} = \delta(o^n, b_1, \ldots b_n) & \\
\langle \hat{S}, \hat{\mathcal{E}}, \langle X, C'\rangle\ C, D\rangle & \longmapsto_{\mathsf{secd}} & \langle\langle\langle X, C'\rangle, \hat{\mathcal{E}}\rangle\ \hat{S}, \hat{\mathcal{E}}, \hat{C}, \hat{D}\rangle & [\mathsf{secd4}] \\
\langle \hat{V}\ \langle\langle X, C'\rangle, \hat{\mathcal{E}}'\rangle\ \hat{S}, \hat{\mathcal{E}}, \mathtt{ap}\ C, D\rangle & \longmapsto_{\mathsf{secd}} & \langle \epsilon, \hat{\mathcal{E}}'[X \leftarrow \hat{V}], C', \langle \hat{S}, \hat{\mathcal{E}}, C, D\rangle\rangle & [\mathsf{secd5}] \\
\langle \hat{V}\ \hat{S}, \hat{\mathcal{E}}, \emptyset, \langle \hat{S}', \hat{\mathcal{E}}', C', D\rangle\rangle & \longmapsto_{\mathsf{secd}} & \langle \hat{V}\ \hat{S}', \hat{\mathcal{E}}', C', D\rangle & [\mathsf{secd6}]
\end{array}$$

$$eval_{\mathsf{secd}}(M) = \begin{cases} b & \text{if } \langle \epsilon, \emptyset, [\![M]\!]_{\mathsf{secd}}, \epsilon\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{secd}} \langle b, \hat{\mathcal{E}}, \epsilon, \epsilon\rangle \\ \mathtt{function} & \text{if } \langle \epsilon, \emptyset, [\![M]\!]_{\mathsf{secd}}, \epsilon\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{secd}} \langle\langle\langle X, \hat{C}\rangle, \hat{\mathcal{E}}'\rangle, \hat{\mathcal{E}}, \epsilon, \epsilon\rangle \end{cases}$$

The SECD and CEK machines produce the same result for any expression that has a result.

Claim 7.1: $eval_{\mathsf{secd}} = eval_{\mathsf{cek}}$.

We will not prove this claim. For our purposes, the more interesting fact is that CEK and SECD machines produce the same result in different ways. In particular, a programmer might *notice* the difference when using a real machine with finite resources.

7.2 Context Space

The difference between SECD and CEK is in how context is accumulated and saved while subexpressions are evaluated.

- In the SECD machine, context is created by function calls, where the current stack, environment, and control are packaged into a dump. The stack provides a working area for assembling application arguments..

  This view of context accumulation is natural when approaching computation as in Java, Pascal, or C.

- In the CEK machine, context is created when evaluating an application function or argument, regardless of whether the function or argument is a complex expression. (The argument stack is subsumed by the continuation.) Function application always shrinks the context, rather than expanding it.

  This view of context accumulation is natural when approaching computation as in the λ-calculus, Scheme, or ML.

Due to this difference, the SECD and CEK machines behave differently on recursive programs. As an extreme example, consider the evaluation of Ω:

$$(\lambda x.x\ x)(\lambda x.x\ x)$$

Both machines loop forever on this expression. However, the nature of the infinite loop is different. The CEK machine's state will never grow beyond a certain size while evaluating:

$$\begin{array}{lll}
\langle\langle((\lambda x.x\ x)(\lambda x.x\ x)), \emptyset\rangle, \mathtt{mt}\rangle & & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.x\ x), \emptyset\rangle, \langle \mathtt{arg}, \langle(\lambda x.x\ x), \emptyset\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.x\ x), \emptyset\rangle, \langle \mathtt{fun}, \langle(\lambda x.x\ x), \emptyset\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(x\ x), \{\langle x, c\rangle\}\rangle, \mathtt{mt}\rangle & \text{where } c = \langle(\lambda x.x\ x), \emptyset\rangle \\
\longmapsto_{\mathsf{cek}} & \langle\langle x, \{\langle x, c\rangle\}\rangle, \langle \mathtt{arg}, \langle x, \{\langle x, c\rangle\}\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.x\ x), \emptyset\rangle, \langle \mathtt{arg}, \langle x, \{\langle x, c\rangle\}\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle x, \{\langle x, c\rangle\}\rangle, \langle \mathtt{fun}, \langle(\lambda x.x\ x), \emptyset\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.x\ x), \emptyset\rangle, \langle \mathtt{fun}, \langle(\lambda x.x\ x), \emptyset\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(x\ x), \{\langle x, c\rangle\}\rangle, \mathtt{mt}\rangle & \text{same as five steps back} \\
& \ldots &
\end{array}$$

In contrast, the state of the SECD machine will grow in size forever:

$$\begin{array}{lll}
\langle\epsilon, \emptyset, [\![(\lambda x.x\ x)(\lambda x.x\ x)]\!]_{\mathsf{secd}}, \epsilon\rangle & & \\
= & \langle\epsilon, \emptyset, \langle x, x\ x\ \mathtt{ap}\rangle\ \langle x, x\ x\ \mathtt{ap}\rangle\ \mathtt{ap}, \epsilon\rangle & \\
\longmapsto_{\mathsf{secd}} & \langle\langle\langle x, x\ x\ \mathtt{ap}\rangle, \emptyset\rangle, \emptyset, \langle x, x\ x\ \mathtt{ap}\rangle\ \mathtt{ap}, \epsilon\rangle & \\
\longmapsto_{\mathsf{secd}} & \langle\langle\langle x, x\ x\ \mathtt{ap}\rangle, \emptyset\rangle\ \langle\langle x, x\ x\ \mathtt{ap}\rangle, \emptyset\rangle, \emptyset, \mathtt{ap}, \epsilon\rangle & \\
\longmapsto_{\mathsf{secd}} & \langle\epsilon, \{\langle x, \hat{V}\rangle\}, x\ x\ \mathtt{ap}, \langle\epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle & \text{where } \hat{V} = \langle\langle x, x\ x\ \mathtt{ap}\rangle, \emptyset\rangle \\
\longmapsto_{\mathsf{secd}} & \langle\hat{V}, \{\langle x, \hat{V}\rangle\}, x\ \mathtt{ap}, \langle\epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle & \\
\longmapsto_{\mathsf{secd}} & \langle\hat{V}\ \hat{V}, \{\langle x, \hat{V}\rangle\}, \mathtt{ap}, \langle\epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle & \\
\longmapsto_{\mathsf{secd}} & \langle\epsilon, \{\langle x, \hat{V}\rangle\}, x\ x\ \mathtt{ap}, \langle\epsilon, \{\langle x, \hat{V}\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle\rangle & \text{same } \hat{S}, \hat{\mathcal{E}}, \text{ and } \hat{C}\text{; larger } \hat{D} \\
& \ldots & \\
\longmapsto_{\mathsf{secd}} & \langle\epsilon, \{\langle x, \hat{V}\rangle\}, x\ x\ \mathtt{ap}, \langle\epsilon, \{\langle x, \hat{V}\rangle\}, \epsilon, \langle\epsilon, \{\langle x, \hat{V}\rangle\}, \epsilon, \langle\epsilon, \emptyset, \epsilon, \epsilon\rangle\rangle\rangle\rangle & \\
& \ldots &
\end{array}$$

In other words, the CEK machine evaluates Ω as a loop, while the SECD machine evaluates Ω as infinite function recursion as in Java or C.

The notion of saving context on argument evaluations, rather than on function calls, corresponds to **tail calls** (or **tail recursion**) in programming languages. For example, the following program runs forever in Scheme, without running out of stack space:

```
(define (f x) (f x))
(f 'ignored)
```

More generally, the semantics of Scheme must be understood in terms of a CEK-like model, and not an SECD-like model.

Of course, the relevance of tail calls extends beyond infinite loops. Since real machines generally have a finite amount of memory, the difference between loops and stack-based recursion is important for large (but bounded) recursions as well.

For example, every C programmer knows to implement a function that processes a long list as a loop, instead of with recursion, to avoid overflowing the stack. In languages that support tail calls, a recursive definition can be just as efficient as a loop. Languages such as Scheme and ML do not even provide a looping construct, since it is unnecessary.

▷ **Exercise 7.1.** Which of the following expressions loop with a bounded size in the CEK machine?

- $\mathsf{Y_v}\ (\lambda f.\lambda x.f\ x)\ \ulcorner 0 \urcorner$
- $\mathsf{Y_v}\ (\lambda f.\lambda x.\mathsf{if0}\ x\ (f\ \ulcorner 10 \urcorner)\ (f\ (-\ x\ \ulcorner 1 \urcorner)))\ \ulcorner 0 \urcorner$
- $\mathsf{Y_v}\ (\lambda f.\lambda x.\mathsf{if0}\ x\ (f\ \ulcorner 10 \urcorner)\ (-\ (f\ x)\ \ulcorner 1 \urcorner))\ \ulcorner 0 \urcorner$

▷ **Exercise 7.2.** Which of the following expressions execute in the CEK machine with a bounded size that is independent of n, for a non-negative n?

- $\mathsf{Y_v}\ \mathsf{mkodd}\ \ulcorner n \urcorner$ where $\mathsf{mkodd} \doteq \lambda f.\lambda x.\mathsf{if0}\ x\ \mathsf{false}\ (\mathsf{not}\ (f\ (-\ x\ 1)))$
- $\mathsf{Y_v}\ \mathsf{mkoe}\ \mathsf{false}\ \ulcorner n \urcorner$ where $\mathsf{mkoe} \doteq \lambda f.\lambda z.\lambda x.\mathsf{if0}\ x\ z\ (f\ (\mathsf{not}\ z)\ (-\ x\ 1))$

7.3 Environment Space

The CEK machine behaves quantitatively better than the SECD machine, but is it good enough? Remember that the CEK machine is meant as a faithful implementation of $\longmapsto_{\mathbf{v}}$. We might then ask whether $\longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}}$ finds a value for an expression in the same amount of space (asymptotically) as $\longmapsto\!\!\!\!\rightarrow_{\mathbf{v}}$.

The answer, it turns out, is that $\longmapsto\!\!\!\!\rightarrow_{\mathsf{cek}}$ might use more space. Consider the following reduction sequence, where F is shorthand for $(\lambda f.(\lambda x.((f\ f)\ (\lambda x.x))))$:

$$
\begin{array}{ll}
 & ((\mathsf{F}\ \mathsf{F})\ \ulcorner 0 \urcorner) \\
\longmapsto_{\mathbf{v}} & ((\lambda x.((\mathsf{F}\ \mathsf{F})\ (\lambda x.x)))\ \ulcorner 0 \urcorner) \\
\longmapsto_{\mathbf{v}} & ((\mathsf{F}\ \mathsf{F})\ (\lambda x x)) \\
\longmapsto_{\mathbf{v}} & ((\lambda x.((\mathsf{F}\ \mathsf{F})\ (\lambda x.x)))\ (\lambda x x)) \\
\longmapsto_{\mathbf{v}} & ((\mathsf{F}\ \mathsf{F})\ (\lambda x x)) \\
\longmapsto_{\mathbf{v}} & ((\lambda x.((\mathsf{F}\ \mathsf{F})\ (\lambda x.x)))\ (\lambda x x)) \\
\longmapsto_{\mathbf{v}} & \ldots
\end{array}
$$

Clearly, this sequence continues forever, and the overall expression is never larger than the fourth step.

The CEK machine, however, grows indefinitely:

$$
\begin{array}{lll}
 & \langle\langle((\mathsf{F}\ \mathsf{F})\ \ulcorner 0 \urcorner), \emptyset\rangle, \mathtt{mt}\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\mathsf{F}\ \mathsf{F}), \emptyset\rangle, \langle \mathtt{arg}, \langle \ulcorner 0 \urcorner, \emptyset\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{arg}, \langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{arg}, \langle \ulcorner 0 \urcorner, \emptyset\rangle, \mathtt{mt}\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{fun}, \langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{arg}, \langle \ulcorner 0 \urcorner, \emptyset\rangle, \mathtt{mt}\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.((f\ f)\ (\lambda x.x))), \mathcal{E}_0\rangle, \langle \mathtt{arg}, \langle \ulcorner 0 \urcorner, \emptyset\rangle, \mathtt{mt}\rangle\rangle & \mathcal{E}_0 = \{\langle f, \langle \mathsf{F}, \emptyset\rangle\rangle\} \\
\longmapsto_{\mathsf{cek}} & \langle\langle \ulcorner 0 \urcorner, \emptyset\rangle, \langle \mathtt{fun}, \langle(\lambda x.((f\ f)\ (\lambda x.x))), \mathcal{E}_0\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle((f\ f)\ (\lambda x.x)), \mathcal{E}_1\rangle, \mathtt{mt}\rangle & \mathcal{E}_1 = \{\langle x, \langle \ulcorner 0 \urcorner, \emptyset\rangle\rangle\} \cup \mathcal{E}_0 \\
\longmapsto_{\mathsf{cek}} & \langle\langle(f\ f), \mathcal{E}_1\rangle, \langle \mathtt{arg}, \langle(\lambda x.x), \mathcal{E}_1\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle f, \mathcal{E}_1\rangle, \langle \mathtt{arg}, \langle f, \mathcal{E}_1\rangle, \langle \mathtt{arg}, \langle(\lambda x.x), \mathcal{E}_1\rangle, \mathtt{mt}\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{arg}, \langle f, \mathcal{E}_1\rangle, \langle \mathtt{arg}, \langle(\lambda x.x), \mathcal{E}_1\rangle, \mathtt{mt}\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle f, \mathcal{E}_1\rangle, \langle \mathtt{fun}, \langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{arg}, \langle(\lambda x.x), \mathcal{E}_1\rangle, \mathtt{mt}\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{fun}, \langle \mathsf{F}, \emptyset\rangle, \langle \mathtt{arg}, \langle(\lambda x.x), \mathcal{E}_1\rangle, \mathtt{mt}\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.((f\ f)\ (\lambda x.x))), \mathcal{E}_2\rangle, \langle \mathtt{arg}, \langle(\lambda x.x), \mathcal{E}_1\rangle, \mathtt{mt}\rangle\rangle & \mathcal{E}_2 = \{\langle f, \langle \mathsf{F}, \emptyset\rangle\rangle\} \\
\longmapsto_{\mathsf{cek}} & \langle\langle(\lambda x.x), \mathcal{E}_1\rangle, \langle \mathtt{fun}, \langle(\lambda x.((f\ f)\ (\lambda x.x))), \mathcal{E}_2\rangle, \mathtt{mt}\rangle\rangle & \\
\longmapsto_{\mathsf{cek}} & \langle\langle((f\ f)\ (\lambda x.x)), \mathcal{E}_3\rangle, \mathtt{mt}\rangle & \mathcal{E}_3 = \{\langle x, \langle(\lambda x.x), \mathcal{E}_1\rangle\rangle\} \cup \mathcal{E}_2 \\
\longmapsto_{\mathsf{cek}} & \ldots &
\end{array}
$$

On first glance, at the step that creates $\mathcal{E}_3$, the machine state is the same size as the step that creates $\mathcal{E}_1$. The definition of $\mathcal{E}_3$, however, refers to $\mathcal{E}_1$, which means that $\mathcal{E}_3$ stands for a larger environment than $\mathcal{E}_1$. Similarly, $\mathcal{E}_5$ will build on $\mathcal{E}_3$, and so on, so that the environment keeps growing, which means that the overall machine state keeps growing.

In the closure $\langle(\lambda x.x), \mathcal{E}_1\rangle$, none of the bindings in $\mathcal{E}_1$ are actually needed, because $(\lambda x.x)$ has no free variables. In other words, substitutions were delayed and accumulated as $\mathcal{E}_1$, but the substitutions would not have affected the expression if they had been performed immediately.

If the substitions were actually needed, then reductions with $\longmapsto_{\mathbf{v}}$ would grow, too. For example, replacing F in the example with $(\lambda f.(\lambda x.((f\ f)\ (\lambda y.x))))$ leads to an infinitely growing state with both $\longmapsto_{\mathbf{v}}$ and $\longmapsto_{\mathsf{cek}}$. In that case, the x in $(\lambda y.x)$ is free, so an immediate substution for x in $(\lambda y.x)$ changes the expression to keep the same information that is collected in $\mathcal{E}_1$ (and then $\mathcal{E}_3$, etc.).

In general, to repair the CEK machine and obtain the same space consumption as $\longmapsto_{\mathbf{v}}$, we can change closure formation to discard unnecessary subsitions from an environment. Given an expression M and an environment $\mathcal{E}$, the closure should be $\langle M, \mathcal{E}'\rangle$ where $\mathcal{E}'$ is like $\mathcal{E}$, but with its domain restricted to $\mathcal{FV}(M)$. For example, closing $(\lambda x.x)$ with $\mathcal{E}_1$ should form $\langle(\lambda x.x), \emptyset\rangle$.

A machine with the same space consumption as $\longmapsto_{\mathbf{v}}$ is called **safe for space**. Some practical implementations of Scheme-like languages are not safe for space; however, practical implementations do prune the environment when forming a closure that corresponds to a procedure value. Since recursion requires procedure closures, pruning for procedures is enough to make tail calls effective; programs that run in constant space with $\longmapsto_{\mathbf{v}}$ also run in constant space in a machine that prunes only procedure-closure environments (but programs that run in $\mathrm{O}(n)$ with $\longmapsto_{\mathbf{v}}$ may, for example, run in $\mathrm{O}(n^2)$ space in the same machine).

7.4 History

The SECD machine is from Landin's original paper on ISWIM [5]. Plotkin showed that an evaluator based on the SECD machine is equivalent to call-by-value lambda calculus [8].

Chase [3] is usually credited as the first to observe space problems when implementing an ISWIM-like language through a CEK-like interpreter. Appel [1] formalized the notion of safe-for-space. Clinger [4] connected the notions of tail recursion and space safety in a more implementation-neutral formalization, similar to the connection here between $\longmapsto_{\mathbf{v}}$ and $\longmapsto_{\mathsf{cek}}$.

Chapter 8: Continuations

The CEK machine explains how a language like ISWIM can be implemented on a realistic machine:

- Each of three parts of state in the CEK machine (the current expression, environment, and continuation) can be stored in a register.
- Each CEK step can be implemented with a fairly simple sequence of processor instructions. Environment lookup has a fairly obvious implementation.

We have neglected the allocation aspects of the machine (for now), but we've seen that the machine state's size is bounded when it should be (if environments are pruned appropriately).

As we developed the machine, not only do theorems about evaluation with $eval_{\mathbf{v}}$ carry over, but the programmer's understanding of evaluation with $\longmapsto_{\mathbf{v}}$ is fairly intact. This is important to our goal of decribing languages and evaluation in an understandable way.

There is, however, one "strange" aspect of the machine: the K register. The C and E registers correspond closely to constructs in the language. The C register contains an expression, which programmers can form at will, and the E register contains an environment, which a programmer can capture using a λ (to form a closure). But the K register does not correspond to anything that the programmer can write or capture within the language.

What if we add a construct to the language for capturing the current continuation? The only use for a continuation is in the last slot of the machine, so the only useful operation a program might perform on a continution is to use it—replacing the current continuation.

To investigate the use of continuations as values, we extend ISWIM with `letcc` and `cc` forms:

$$\begin{array}{rcl} M & = & \ldots \\ & | & (\texttt{letcc}\ X\ M) \\ & | & (\texttt{cc}\ M\ M) \end{array}$$

The new `letcc` form grabs the current continuation and binds it to a variable while evaluting an expression. The new `cc` form replaces the current continuation with the value of its first subexpression, and supplies the value of its second subexpression to the newly restored continuation.

8.1 Saving Contexts

Since a continuation is just an inside-out evaluation context, we can rephrase the above descriptions in terms of evalaution contexts. From this view, evalaution contexts are now values (and therefore expressions); we'll wrap context values with $\lfloor\ \rfloor$ so that they are not confused with active contexts, and so they do not interfere with hole replacement:

$$\begin{array}{rcl} M & = & \ldots \\ & | & \lfloor E \rfloor \end{array} \qquad \begin{array}{rcl} V & = & \ldots \\ & | & \lfloor E \rfloor \end{array}$$

In terms of evaluation contexts, the `letcc` form grabs the current context. A use of the `cc` form later will restore the grabbed context, regardless of whatever might be happening at that later point. The entire context at the later point is discarded; for example, if the evaluator was in the process of evaluating some argument to a primitive, then we forget about the primitive application, and everything around it, and return instead to whatever we were doing when the context was grabbed.

Here's a simple example use of `letcc` and `cc`:

$$\begin{array}{l}(+\ \ulcorner 1\urcorner \\ \quad (\texttt{letcc}\ x \\ \quad\quad (+\ (\lambda y.y) \\ \quad\quad\quad (\texttt{cc}\ x\ \ulcorner 12\urcorner))))\end{array}$$

In the inner addition expression, the first argument is a function. Evaluating the example expression starts evaluating the addition expression, but before the addition can discover that the first argument is not a number, the (`cc` x $\ulcorner 12\urcorner$) expression "jumps" out of the addition. The result will be $\ulcorner 13\urcorner$.

Let's walk through it slowly. To evaluate the outside addition, we first evaluate 1 (which has the value 1), and then consider the `letcc` expression. At that point, the evaluation context is E_0 and the redex is M_0:

$$E_0 = \begin{array}{l}(+\ \ulcorner 1\urcorner \\ \quad [\,])\end{array} \qquad M_0 = \begin{array}{l}(\texttt{letcc}\ x \\ \quad (+\ (\lambda y.y) \\ \quad\quad (\texttt{cc}\ x\ \ulcorner 12\urcorner)))\end{array}$$

The `letcc` expression binds the context E_0 to the variable x via replacement. Then the `letcc` disappears, in the same way that application makes λ disappear. Thus, the entire expression is as follows:

$$\begin{array}{l}(+\ \ulcorner 1\urcorner \\ \quad (+\ (\lambda y.y) \\ \quad\quad (\texttt{cc}\ \lfloor E_o \rfloor\ \ulcorner 12\urcorner))))\end{array}$$

Paritioning this expression produces a context E_1 and redex M_1:

$$E_1 = \begin{array}{l}(+\ \ulcorner 1\urcorner \\ \quad (+\ (\lambda y.y) \\ \quad\quad [\,]))\end{array} \qquad M_1 = (\texttt{cc}\ \lfloor E_0 \rfloor\ \ulcorner 12\urcorner)$$

The context E_1 turns out not to matter, because the `cc` expression throws it away and replaces it with E_0. The `cc` expression also provides $\ulcorner 12\urcorner$ as the value to go into the hole of the restored context, so the expression to evaluate becomes

$$\begin{array}{l}(+\ \ulcorner 1\urcorner \\ \quad \ulcorner 12\urcorner)\end{array}$$

The final result is $\ulcorner 13\urcorner$.

The example above shows how continuations can be used to define an error-handling mechanism. Continuations can also be used for implementing cooperative threads ("coroutines"), or for saving and restoring speculative computations. Some languages, notably Scheme, provide constructs like `letcc` and `cc`. Furthermore, some Scheme implementations define the exception mechanism in terms of continuations.

8.2 Revised Texual Machine

We can define the semantics of `letcc` and `cc` by extending our E grammar and $\longmapsto_{\mathbf{v}}$ relation:

$$\begin{array}{rcl} E & = & \ldots \\ & | & (\texttt{cc}\ E\ M) \\ & | & (\texttt{cc}\ V\ E) \end{array}$$

$$\begin{array}{lcll} E[M] & \longmapsto_{\mathbf{v}} & E[M'] & \text{if} \quad M \,\mathbf{v}\, M' \\ E[(\texttt{letcc}\ X\ M)] & \longmapsto_{\mathbf{v}} & E[M[X \leftarrow \lfloor E \rfloor]] & \\ E[(\texttt{cc}\ \lfloor E' \rfloor\ V)] & \longmapsto_{\mathbf{v}} & E'[V] & \end{array}$$

With these extensions, we can demonstrate formally the reduction of the example from the previous section. The redex is underlied in each step:

$$\begin{array}{ll}
& (+\ \ulcorner 1 \urcorner\ \underline{(\texttt{letcc}\ x\ (+\ (\lambda y.y)\ (\texttt{cc}\ x\ \ulcorner 12 \urcorner)))}) \\
\longmapsto_{\mathbf{v}} & \underline{(+\ \ulcorner 1 \urcorner\ (+\ (\lambda y.y)\ (\texttt{cc}\ \lfloor (+\ \ulcorner 1 \urcorner\ [\,])\rfloor\ \ulcorner 12 \urcorner))}) \\
\longmapsto_{\mathbf{v}} & \underline{(+\ \ulcorner 1 \urcorner\ \ulcorner 12 \urcorner)} \\
\longmapsto_{\mathbf{v}} & \ulcorner 13 \urcorner
\end{array}$$

8.3 Revised CEK Machine

To update the CEK machine, we need new $\longmapsto_{\mathsf{cek}}$ relations. We also need to extend the $\overline{\kappa}$ grammar in the same way the we extended the E grammar, and we need to add continuations (instead of $\lfloor E \rfloor$) to the set of values.

$$\begin{array}{rcl}
\overline{\kappa} & = & \ldots \\
& | & \langle \texttt{ccval}, \langle M, \mathcal{E} \rangle, \overline{\kappa} \rangle \\
& | & \langle \texttt{cc}, \overline{\kappa}, \overline{\kappa} \rangle \\
V & = & \ldots \\
& | & \overline{\kappa}
\end{array}$$

$$\begin{array}{lcll}
\langle\langle (\texttt{letcc}\ X\ M), \mathcal{E} \rangle, \overline{\kappa} \rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E}[X \leftarrow \langle \overline{\kappa}, \emptyset \rangle] \rangle, \overline{\kappa} \rangle & [\mathsf{cek8}] \\
\langle\langle (\texttt{cc}\ M\ N), \mathcal{E} \rangle, \overline{\kappa} \rangle & \longmapsto_{\mathsf{cek}} & \langle\langle M, \mathcal{E} \rangle, \langle \texttt{ccval}, \langle N, \mathcal{E} \rangle, \overline{\kappa} \rangle \rangle & [\mathsf{cek9}] \\
\langle\langle \overline{\kappa}', \mathcal{E} \rangle, \langle \texttt{ccval}, \langle N, \mathcal{E}' \rangle, \overline{\kappa} \rangle \rangle & \longmapsto_{\mathsf{cek}} & \langle\langle N, \mathcal{E}' \rangle, \langle \texttt{cc}, \overline{\kappa}', \overline{\kappa} \rangle \rangle & [\mathsf{cek10}] \\
\langle\langle V, \mathcal{E} \rangle, \langle \texttt{cc}, \overline{\kappa}', \overline{\kappa} \rangle \rangle & \longmapsto_{\mathsf{cek}} & \langle\langle V, \mathcal{E} \rangle, \overline{\kappa}' \rangle & [\mathsf{cek11}] \\
\quad \text{if } V \notin X & & &
\end{array}$$

▷ **Exercise 8.1.** Show the evalution of

$$\langle\langle (+\ \ulcorner 1 \urcorner\ (\texttt{letcc}\ x\ (+\ (\lambda y.y)\ (\texttt{cc}\ x\ \ulcorner 12 \urcorner)))), \emptyset \rangle, \texttt{mt} \rangle$$

to $\langle\langle V, \mathcal{E} \rangle, \texttt{mt} \rangle$ with the revised $\longmapsto_{\mathsf{cek}}$.

▷ **Exercise 8.2.** The above extension of CEK with continuations could be more efficient with simple changes. In particular, the $\overline{\kappa}$ in $\langle \texttt{cc}, \overline{\kappa}, \overline{\kappa}' \rangle$ is never used. Suggest revisions to the $\overline{\kappa}$ grammar and the $\longmapsto_{\mathsf{cek}}$ rules to streamline the machine.

▷ **Exercise 8.3.** Given the following abbreviations:

$$\begin{array}{lcl}
\mathsf{mkodd} & \doteq & \lambda f.\lambda x.\mathsf{if0}\ x\ \mathsf{false}\ (\mathsf{not}\ (f\ (-\ x\ 1))) \\
\mathsf{next} & \doteq & \lambda k.\lambda x.(\texttt{cc}\ k\ (\mathsf{mkpair}\ k\ x)) \\
\mathsf{call} & \doteq & \lambda f.\lambda y.(f\ (\mathsf{next}\ (\mathsf{fst}\ y))\ (\mathsf{snd}\ y)) \\
\mathsf{Z} & \doteq & \lambda f.\lambda x.(\mathsf{call}\ f\ (\texttt{letcc}\ k\ (\mathsf{mkpair}\ k\ x)))
\end{array}$$

What is the value of (Z mkodd 1)? Justify your answer, but don't try to write down a complete CEK reduction.

▷ **Exercise 8.4.** Given the following abbreviation:

$$\mathsf{callcc} \doteq \lambda f.(\texttt{letcc}\ k\ (f\ (\lambda y.(\texttt{cc}\ k\ y))))$$

What does ((callcc callcc) (callcc callcc)) do? Justify your answer.

▷ **Exercise 8.5.** Is the evaluation of ((callcc callcc) (callcc callcc)) bounded in the revised CEK

machine? Justify your answer.

▷ **Exercise 8.6.** Is the evaluation of $((\lambda y.((\mathsf{callcc}\ \mathsf{callcc})\ y))\ (\mathsf{callcc}\ \mathsf{callcc}))$ bounded in the revised CEK machine? Justify your answer.

Chapter 9: Errors and Exceptions

An ISWIM program might diverge for two distinct reasons. First, a program's standard reduction sequence may have a successor for every intermediate state. This situation corresponds to a genuine infinite loop in the program, which, in general, cannot be detected. Second, a program's standard reduction sequence may end in a stuck state, that is, a program that is not a value and has no successor. One typical example is the application of a division primitive to 0; another one is the use of a numeral in function position.

An interpreter that diverges when a program reaches a stuck state leaves the programmer in the dark about the execution of his program. The interpreter should instead signal an error that indicates the source of the trouble. One way to specify the revised interpreter for ISWIM is to add the element `err` to the set of answers and to add a single clause to the definition of the evaluator:

$$eval(M) = \begin{cases} b & \text{if } M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} b \\ \texttt{function} & \text{if } M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} \lambda X.N \\ \texttt{err} & \text{if } M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} N \text{ and } N \text{ is stuck} \end{cases}$$

By Lemma 5.12, which shows that the three enumerated cases plus divergence are the only possible ways for the evaluation of a program to proceed, we know that the extended evaluator is still a partial function from programs to answers. However, beyond programming errors, it is often necessary to terminate the execution of a program in a controlled manner. For example, a program may encode an input–output process for a certain set of data but may have to accept data from some larger set. If the input data for such a program are outside of its proper domain, it should signal this event and terminate in a predictable manner.

In plain ISWIM, a programmer can only express the idea of signaling an error by writing down an erroneous expression like (`add1` $\lambda y.y$). We would like instead a linguistic tool for programmers to express error termination directly. We solve this problem by adding error constructs to the syntax of ISWIM. In the extended language a programmer can explicitly write down `err` with the intention that the evaluation of `err` terminates the execution of the program. We treat this extended version of ISWIM in the first section of this chapter.

When a language contains constructs for signaling errors, it is also natural to add constructs that detect whether an error occurred during the evaluation of a subprogram such that the program can take remedial action. These **exception-handling** constructs are highly desirable in practice and are provided by many programming languages. Besides handling unpreventable errors, exception-handling can be used to exit from loops once the result has been detected, or to implement some form of backtracking. In the second and later sections, we examine a variety of exception handling mechanisms.

9.1 Errors

Since a program can encounter many different types of error situations, we add an undetermined set of errors to ISWIM. Each element of this set plays the role of a syntactic construct. We assume that the set is non-empty but make no other restrictions for now.

M	$=$	$\ldots$	same as plain ISWIM
	$\mid$	$\texttt{err}_l$	
l	$=$	a label	

9.1.1 Calculating with Error ISWIM

Roughly speaking, if a sub-expression signals an error, the error should eliminate any computation that would have beeen executed if the evaluation of the sub-expression had not signalled

an error. A precise specification of the behavior of $\texttt{err}_l$ in terms of calculation rules requires an analysis of how $\texttt{err}_l$ interacts with other syntactic constructions. Take application as an example. If a function is applied to $\texttt{err}_l$, the appliction must be terminated. We can express this requirement with the following informal notions of reduction:

$$\begin{array}{lcl} ((\lambda x.M)\ \texttt{err}_l) & \to & \texttt{err}_l \\ (o^n\ V \ldots\ \texttt{err}_l\ N\ \ldots) & \to & \texttt{err}_l \end{array}$$

The rules clarify that $\texttt{err}_l$ is *not* an ordinary value; otherwise applications like $((\lambda x.\ulcorner 5\urcorner)\ \texttt{err}_l)$ would reduce to $\ulcorner 5\urcorner$. The case for $\texttt{err}_l$ in function position is similarly straightforward. When $\texttt{err}_l$ occurs in function position, there is no function to be applied so the application also reduces to $\texttt{err}_l$:

$$(\texttt{err}_l\ N) \quad \to \quad \texttt{err}_l$$

Next we need to consider the situation when $\texttt{err}_l$ occurs as the body of a procedure. Since a procedure like $(\lambda x.\texttt{err}_l)$ may never be invoked, the $\texttt{err}_l$ should not propagate through the abstraction. The hidden $\texttt{err}_l$ should only become visible when the λ-abstraction is applied to a value. In other words, $(\lambda x.\texttt{err}_l)$ is an irreducible value.

Given that the evaluation context surrounding a redex represents the rest of the computation, a natural formalization of the behavior of $\texttt{err}_l$ says that it erases evaluation contexts. Thus, we introduce the following **error** reduction:

$$E[\texttt{err}_l] \quad \textbf{error} \quad \texttt{err}_l$$

In addition to occurrences of $\texttt{err}_l$ in the original program, stuck expressions should now reduce to some error value. Consider the application of some basic constant b to an arbitrary value V. It should reduce to an error that signals that b cannot be used as a procedure, perhaps using b as the label:

$$(b\ V) \quad \to \quad \texttt{err}_b$$

One question is whether the generated error should also depend on the nature of V, i.e., $\texttt{err}_{bV}$. Suppose an application of b to U for $V \neq U$ reduces to a distinct error, $\texttt{err}_{bU}$. Then, if it is also the case that $V \to_{\mathbf{v}} U$, the calculus proves

$$\texttt{err}_{bV} =_{\mathbf{v}} (b\ V) =_{\mathbf{v}} (b\ U) =_{\mathbf{v}} \texttt{err}_{bU}$$

That is, the two distinct errors are provably equal even though they are not identical and were assumed to be distinct. Implementations of programming languages avoid this problem by signaling an error that only depends on the basic constant in the function position, and not on the argument.

Following the same reasoning, an application of a primitive operation to a value other than a constant must also reduce to an error, independent of the actual arguments. For example, applying the addition primitive to any abstraction yields the error constant

$$(+\ (\lambda xy.x)\ 0) \quad \to \quad \texttt{err}_+$$

and

$$(+\ (\lambda xy.y)\ 0) \quad \to \quad \texttt{err}_+$$

independent of the argument.

Finally, we must consider the application of primitive operations to a bad set of basic constants. Since the application is unique and the error message depends on the constant, we assume that the δ function returns an appropriate error message for bad inputs to a primitive operation. We restrict the δ reduction (as opposed to the δ function) to produce values, however, and introduce a new δ_{err} reduction for errors.

The new reductions define **Error ISWIM**:

$$
\begin{array}{llll}
(o^m\ b_1 \ldots\ b_m) & \delta & V & \text{if} \quad \delta(o^m, b_1, \ldots b_m) = V \\
\\
(o^m\ b_1 \ldots\ b_m) & \delta_{\mathsf{err}} & \mathtt{err}_l & \text{if} \quad \delta(o^m, b_1, \ldots b_m) = \mathtt{err}_l \\
(o^m\ V_1 \ldots\ (\lambda X.N)\ \ldots\ V_m) & \delta_{\mathsf{err}} & \mathtt{err}_{o^m} & \\
(b\ V) & \delta_{\mathsf{err}} & \mathtt{err}_b &
\end{array}
$$

$$\mathbf{e} = \beta_{\mathsf{v}} \cup \delta \cup \delta_{\mathsf{err}} \cup \mathbf{error}$$

An application M is **faulty** if $M\,\delta_{\mathsf{err}}\,\mathtt{err}_l$ for some $\mathtt{err}_l$.

$$A_{\mathbf{e}} = b \cup \{\mathtt{function}\} \cup \mathtt{err}$$

$$eval_{\mathbf{e}}(M) = \begin{cases} b & \text{if } M =_{\mathbf{e}} b \\ \mathtt{function} & \text{if } M =_{\mathbf{e}} \lambda X.N \\ \mathtt{err}_l & \text{if } M =_{\mathbf{e}} \mathtt{err}_l \end{cases}$$

▷ **Exercise 9.1.** Show two different reductions of

$$(+\ (-\ 4\ \mathtt{err_a})\ \mathtt{err_b})$$

to $\mathtt{err}_l$ with $\rightarrow_{\mathbf{e}}$. Indicate the redex (the expression related by $\mathbf{e}$) for each step. Is the resulting l unique?

9.1.2 Consistency for Error ISWIM

We need to check that the addition of $\mathtt{err}_l$ does not destroy the key properties of the evaluator. First, we would like to know that the evaluator (and thus the programming language) is still a function. That is, our goal is to establish a consistency theorem for Error ISWIM.

> **Theorem 9.1 [Consistency with Errors]**: The relation $eval_{\mathbf{e}}$ is a partial function.

The proof of this theorem has the same structure as the proof of Theorem 4.3, the corresponding theorem for $eval_{\mathbf{v}}$. We first prove a Church-Rosser property, that is, we show that provably equal results reduce to some common contractum. The theorem follows from Church-Rosser property. The proof of the property relies on the fact that the diamond property holds for the underlying reduction. The diamond property for δ and β_{v} extended to the larger syntax obviously holds; an adaptation of the proof for ISWIM to Error ISWIM is straightforward. But the calculus for Error ISWIM also includes reductions that create and move $\mathtt{err}_l$. Fortunately, it is possible to combine the diamond theorems for different reductions on the same language, a method that we will use to establish the diamond property for $\mathbf{e}$.

We first deal with the extension of the ISWIM calculus to the larger syntax. We define $\mathbf{w}$ as the extension of $\mathbf{v}$ to the full set of Error ISWIM expressions.

$$\mathbf{w} = \delta \cup \beta_{\mathsf{v}}$$

Lemma 9.2 [Diamond Property for $\twoheadrightarrow_{\mathbf{w}}$]: If $L \twoheadrightarrow_{\mathbf{w}} M$ and $L \twoheadrightarrow_{\mathbf{w}} N$, then there exists an expression K such that $M \twoheadrightarrow_{\mathbf{w}} K$ and $N \twoheadrightarrow_{\mathbf{w}} K$.

The proof is essentially the same as for $\twoheadrightarrow_{\mathbf{v}}$. Next, we turn to the analysis of the two new notions of reduction. We define **f** as the rest of **e**:

$$\mathbf{f} = \mathbf{error} \cup \delta_{\mathsf{err}}$$

Unlike **v** or **w**, this notion of reduction never duplicates expressions or creates new opportunities to reduce sub-expressions. We might therefore expect that the one-step relation satisfies the diamond property, but, unfortunately, it does not hold. Consider the reduction from $E[\mathtt{err}_l]$ to $\mathtt{err}_l$. If $E[\mathtt{err}_l]$ also reduces to $E'[\mathtt{err}_l]$ for some evaluation context E', then $E'[\mathtt{err}_l]$ directly reduces to $\mathtt{err}_l$. Hence it is not the one-step reduction $\rightarrow_{\mathbf{f}}$ but its reflexive closure $\rightarrow_{\mathbf{f}}^0$ that satisfies the diamond property.

Lemma 9.3 [Diamond Property for $\twoheadrightarrow_{\mathbf{f}}$]: If $L \twoheadrightarrow_{\mathbf{f}} M$ and $L \twoheadrightarrow_{\mathbf{f}} N$ then there exists an expression K such that $M \twoheadrightarrow_{\mathbf{f}} K$ and $N \twoheadrightarrow_{\mathbf{f}} K$.

Proof for Lemma 9.3: We first prove the claim for a subset of the relation: If $L \rightarrow_{\mathbf{f}}^0 M$ and $L \rightarrow_{\mathbf{f}}^0 N$ then there exists an expression K such that $M \rightarrow_{\mathbf{f}}^0 K$ and $N \rightarrow_{\mathbf{f}}^0 K$. The lemma clearly follows from this claim. The proof proceeds by case analysis of the reduction from L to M:

- **Case L is faulty;** $M = \mathtt{err}_l$
  Clearly, if L is faulty and $L \rightarrow_{\mathbf{f}}^0 N$ then N is faulty. Hence, $K = \mathtt{err}_l$ is the desired fourth expression.
- **Case** $L = E[\mathtt{err}_l]; M = \mathtt{err}_l$
  It is easy to check that any **f**-reduction applied to $E[\mathtt{err}_l]$ yields an expression of the shape $E'[\mathtt{err}_l]$ for some evaluation context E', which implies that the reduction **error** applies to N.
- **Case** $L = M$
  Set $K = N$.
- **Case** $L = C[L']$**,** $M = C[\mathtt{err}_l]$**,** $L'\ \mathbf{f}\ \mathtt{err}_l$
  If $N = C'[L']$ for some context C', then $K = C'[\mathtt{err}_l]$ because $C[L'] \rightarrow_{\mathbf{f}}^0 C'[L']$ implies that for all N', $C[N'] \rightarrow_{\mathbf{f}}^0 C'[N']$. Otherwise $N = C'[\mathtt{err}_l]$ by the definition of compatibility, in which case $K = M = N$.

▷ **Exercise 9.2.** Prove that if L is faulty and $L \rightarrow_{\mathbf{f}} N$, then N is faulty and reduces to the same error element.

▷ **Exercise 9.3.** Prove that If $L = E[\mathtt{err}_l]$ and $L \rightarrow_{\mathbf{f}} M$, then there exists an evaluation context E' such that $M = E'[\mathtt{err}_l]$.

After establishing that each of the notions of reduction **w** and **f** satisfy the diamond property, we need to show that these results can be combined. Intuitively, the combination of **w** and **f** should have the diamond property because the two notions of reduction do not interfere with each other. That is, if two reductions apply to a term, the redexes do not overlap and the reduction of one redex may destroy but not otherwise affect the other redex. Technically speaking, the two one-step reductions commute.

If we can show that the one-step relations based on **w** and **f** commute, the commutation for their transitive closures clearly follows. However, the one-step relations do not commute. Consider the expression

$$((\lambda f.(f\ (f\ \ulcorner 0 \urcorner))))\ (\lambda x.\underline{(\mathtt{add1}\ (\lambda x.x))})$$

Reducing the underlined δ_{err}-redex first yields

$$((\lambda f.(f\ (f\ \ulcorner 0 \urcorner)))\ (\lambda x.\mathtt{err}_{\mathtt{add1}})) \to_{\mathbf{e}} ((\lambda x.\mathtt{err}_{\mathtt{add1}})\ ((\lambda x.\mathtt{err}_{\mathtt{add1}})\ \ulcorner 0 \urcorner))$$

but reducing the β_{v}-redex first requires *two* δ_{err}-reductions to reach the same state:

$$\begin{array}{rl} & ((\lambda x.\underline{(\mathtt{add1}\ (\lambda x.x))})\ ((\lambda x.\underline{(\mathtt{add1}\ (\lambda x.x))})\ \ulcorner 0 \urcorner)) \\ \to_{\mathbf{e}} & \underline{((\lambda x.\mathtt{err}_{\mathtt{add1}})\ ((\lambda x.(\mathtt{add1}\ (\lambda x.x)))\ \ulcorner 0 \urcorner))} \\ \to_{\mathbf{e}} & ((\lambda x.\mathtt{err}_{\mathtt{add1}})\ ((\lambda x.\mathtt{err}_{\mathtt{add1}})\ \ulcorner 0 \urcorner)) \end{array}$$

Fortunately, it is still possible to extend a commutation result for the one-step relations to the full reductions when only one direction needs multiple steps. The proof of the following lemma illustrates the idea.

Lemma 9.4: The reductions $\twoheadrightarrow_{\mathbf{w}}$ and $\twoheadrightarrow_{\mathbf{f}}$ commute.

Proof for Lemma 9.4: The first step of the proof shows that for all L, M, and N such that $L \to_{\mathbf{w}} M$ and $L \to_{\mathbf{f}} N$, there exists a K such that $M \twoheadrightarrow_{\mathbf{f}} K$ and $N \to_{\mathbf{w}}^{K}$.

The proof of the claim is an induction on the derivation of $L \to_{\mathbf{w}} M$:

- **Case** $L = ((\lambda x.L')\ V)$**,** $M = L'[X \leftarrow V]$
  Only two subcases are possible because a value cannot reduce to $\mathtt{err}_l$:
  - **Case** $N = ((\lambda x.L'')\ V)$**,** $L' \to_{\mathbf{f}} L''$
    If we can show that $M[X \leftarrow V] \to_{\mathbf{f}} M'[X \leftarrow V]$ if $M \to_{\mathbf{f}} M'$, then we can take $K = L''[X \leftarrow V]$.
  - **Case** $N = ((\lambda x.L')\ V')$**,** $V \to_{\mathbf{f}} V'$
    If we can show that $M[X \leftarrow V] \twoheadrightarrow_{\mathbf{f}} M[X \leftarrow V']$ if $V \to_{\mathbf{f}} V'$, then we can take $K = L'[X \leftarrow V']$.
- **Case** $L\,\delta\,M$
  Then, by definition of δ and δ_{err}, it is impossible that $L \to_{\mathbf{f}} N$.
- **Case** $L = C[L']$**,** $M = C[M']$**,** $L'\,\mathbf{w}\,M'$
  If $C[L'] \to_{\mathbf{f}} C[L'']$, the claim follows from the inductive hypothesis. Otherwise, $C[L'] \to_{\mathbf{f}} C'[L']$ and $K = C'[M']$ is the correct choice.

The rest follows by a simple induction on the length of the two reduction sequences.

Based on the preceding lemma and the two diamond lemmas, we can now prove that the reduction generated by **e** satisfies the crucial diamond lemma.

Lemma 9.5 [Diamond Property for $\twoheadrightarrow_{\mathbf{e}}$]: If $L \twoheadrightarrow_{\mathbf{e}} M$ and $L \twoheadrightarrow_{\mathbf{e}} N$, then there exists an expression K such that $M \twoheadrightarrow_{\mathbf{e}} K$ and $N \twoheadrightarrow_{\mathbf{e}} K$.

Proof for Lemma 9.5: For the cases where $L = M$ or $L = N$, the lemma obviously holds. Thus assume $L \neq M$ and $L \neq N$. The proof is an induction on the product of the reduction steps from L to M and L to N. Pick an $M_1 \neq L$ and an $N_1 \neq L$ such that the reduction steps from L to M_1 and N_1 are either **f** or **w** steps. Four cases are possible:

- **Case** $L \twoheadrightarrow_{\mathbf{f}} M_1, L \twoheadrightarrow_{\mathbf{w}} N_1$
  The lemma follows by the commutation property for the two reductions.
- **Case** $L \twoheadrightarrow_{\mathbf{w}} M_1, L \twoheadrightarrow_{\mathbf{f}} N_1$
  Again, the lemma is a consequence of the commutation property.
- **Case** $L \twoheadrightarrow_{\mathbf{f}} M_1, L \twoheadrightarrow_{\mathbf{f}} N_1$
  The dimaond property for $\twoheadrightarrow_{\mathbf{f}}$ implies the lemma.
- **Case** $L \twoheadrightarrow_{\mathbf{w}} M_1, L \twoheadrightarrow_{\mathbf{w}} N_1$
  This case holds due to the dimaond property for $\twoheadrightarrow_{\mathbf{w}}$.

9.1.3 Standard Reduction for Error ISWIM

As discussed in previous chapters, a specification of the evaluator based on a calculus does not lend itself immediately to a good implementation. A better specification defines a strategy that picks a unique redex from a program and rewrites the program until it turns into an answer. For ISWIM, the correct strategy selects the leftmost-outermost redex, reduces it, and continues the evaluation with the resulting program. Technically, recall that the deterministic ISWIM evaluator partitions the program into an evaluation context E and a $\mathbf{v}$-redex M. If N is the contractum of M, the next state in the evaluation sequence is $E[N]$: $E[M] \longmapsto_{\mathbf{v}} E[N]$. The strategy works because for every program that is not a value, there is a unique partitioning of the program into an evaluation context and a $\mathbf{v}$-redex.

The unique partitioning is not so easily available for Error ISWIM. To see why, note that we can demonstrate two possible reductions with $\rightarrow_{\mathbf{e}}$ for $(+\ \ulcorner 1 \urcorner\ (+\ \ulcorner 2 \urcorner\ \mathtt{err}_a))$:

$$\frac{\dfrac{\dfrac{(+\ 1\ (+\ \ulcorner 2 \urcorner\ [])) \in E}{(+\ 1\ (+\ \ulcorner 2 \urcorner\ \mathtt{err}_a))\ \mathbf{error}\ \mathtt{err}_a}}{(+\ 1\ (+\ \ulcorner 2 \urcorner\ \mathtt{err}_a))\ \mathbf{e}\ \mathtt{err}_a} \qquad [] \in C}{(+\ \ulcorner 1 \urcorner\ (+\ \ulcorner 2 \urcorner\ \mathtt{err}_a)) \rightarrow_{\mathbf{e}} \mathtt{err}_a}$$

and

$$\frac{\dfrac{\dfrac{(+\ \ulcorner 2 \urcorner\ []) \in E}{(+\ \ulcorner 2 \urcorner\ \mathtt{err}_a)\ \mathbf{error}\ \mathtt{err}_a}}{(+\ \ulcorner 2 \urcorner\ \mathtt{err}_a)\ \mathbf{e}\ \mathtt{err}_a} \qquad (+\ \ulcorner 1 \urcorner\ []) \in C}{(+\ \ulcorner 1 \urcorner\ (+\ \ulcorner 2 \urcorner\ \mathtt{err}_a)) \rightarrow_{\mathbf{e}} (+\ \ulcorner 1 \urcorner\ \mathtt{err}_a)}$$

Of course, we do not expect $\rightarrow_{\mathbf{e}}$ to be a function, so the above result is not surprising. But it turns out that both $[]$ and $(+\ \ulcorner 1 \urcorner\ [])$ are in E as well as C, so there are two ways to match $E[M]$ and $E[N]$ with $M\,\mathbf{e}\,N$. Consequently, we will not define $\longmapsto_{\mathbf{e}}$ as the closure of $\mathbf{e}$ over E. To avoid non-determinism, we define $\longmapsto_{\mathbf{e}}$ to directly propagate errors, and close in general only over the other relations, β_{v}, δ, and δ_{err}:

$$\tilde{\mathbf{e}} = \beta_{\mathsf{v}} \cup \delta \cup \delta_{\mathsf{err}}$$

$$\begin{array}{lcll} E[M] & \longmapsto_{\mathbf{e}} & E[N] & \text{if} \quad M\,\tilde{\mathbf{e}}\,N \\ E[\mathtt{err}_l] & \longmapsto_{\mathbf{e}} & \mathtt{err}_l & \end{array}$$

We define an evaluator for this reduction, as usual:

$$eval^{\mathsf{s}}_{\mathbf{e}}(M) = \begin{cases} b & \text{if } M \longmapsto\!\!\!\!\rightarrow_{\mathbf{e}} b \\ \mathtt{function} & \text{if } M \longmapsto\!\!\!\!\rightarrow_{\mathbf{e}} \lambda X.N \\ \mathtt{err}_l & \text{if } M \longmapsto\!\!\!\!\rightarrow_{\mathbf{e}} \mathtt{err}_l \end{cases}$$

To state the unique partitioning theorem that ensures $\longmapsto_{\mathbf{e}}$ and $eval^{\mathsf{s}}_{\mathbf{e}}$ are functions, we must restrict the shape of redex expressions M_{re} to match the way $\longmapsto_{\mathbf{e}}$ is defined.

$$\begin{array}{rcl} M_{\mathsf{re}} & = & (V\ V) \\ & | & (o^m\ V \ldots\ V) \\ & | & \mathtt{err}_l \end{array}$$

Lemma 9.6 [Unique Evaluation Contexts for Error ISWIM]: Every closed Error ISWIM expression M is either a value, or there exists a unique evaluation context E and a standard redex M_{re} such that $M = E[M_{\mathsf{re}}]$.

Proof for Lemma 9.6: The proof is similar to that of Lemma 5.1.

The lemma validates that the new specification of the evaluator, $eval^{\mathsf{s}}_{\mathbf{e}}$, is a partial function. But, we really want to know that the new evaluator function is equal to the old one and can be used in its place. This is also true, though we will not prove it here.

Theorem 9.7: $eval^{\mathsf{s}}_{\mathbf{e}} = eval_{\mathbf{e}}$

▷ **Exercise 9.4.** Adapt the proof of Theorem 5.5 to prove Theorem 9.7.

▷ **Exercise 9.5.** If $\mathtt{err}_l$ contains only one element, $\mathtt{err}_0$, a feasible alternative for propagating $\mathtt{err}_l$ through applications is the following notion of reduction:

$$
\begin{array}{rcl}
E_{\mathsf{a}} & = & [\,] \\
& | & (M\ E_{\mathsf{a}}) \\
& | & (E_{\mathsf{a}}\ M) \\
& | & (o^m\ M \ldots\ M\ E_{\mathsf{a}}\ M\ \ldots M)
\end{array}
$$

$$
\begin{array}{lcll}
E[M] & \longmapsto_{\mathbf{e}'} & E[N] & \text{if} \quad M \,\tilde{\mathbf{e}}\, N \\
E_{\mathsf{a}}[\mathtt{err}_l] & \longmapsto_{\mathbf{e}'} & \mathtt{err}_l &
\end{array}
$$

Demonstrate that $\longmapsto_{\mathbf{e}'}$ is a not function (despite the use of $\longmapsto$). Is $eval^{\mathsf{s}}_{\mathbf{e}'}$, defined in terms of $\longmapsto_{\mathbf{e}'}$, a function? Is $eval^{\mathsf{s}}_{\mathbf{e}'}$ a function when $\mathtt{err}_l$ contains at least two elements?

9.1.4 Relating ISWIM and Error ISWIM

The extension of a programming language almost immediately raises the question whether programs of the original language are executed correctly. Given a fixed set of constants and primitive operations, including a fixed δ function, Error ISWIM is an extension of ISWIM that affects the behavior of diverging programs but not that of terminating ones. More precisely, an execution of a terminating ISWIM program on the Error ISWIM evaluator yields the same answer as an execution on the ISWIM evaluator; and the execution of diverging (formerly stuck) programs may terminate with an error.

Theorem 9.8: For any M,

1. $eval_{\mathbf{v}}(M) = A$ implies that $eval_{\mathbf{e}}(M) = A$;
2. $eval_{\mathbf{e}}(M) = A$ and $A \neq \mathtt{err}_l$ implies that $eval_{\mathbf{v}}(M) = A$;
3. $eval_{\mathbf{e}}(M) = \mathtt{err}_l$ implies that $eval_{\mathbf{v}}(M)$ diverges; and
4. if $eval_{\mathbf{e}}(M)$ is undefined, then $eval_{\mathbf{v}}(M)$ is undefined.

Proof for Theorem 9.8: For the proof of the first claim assume $M =_{\mathbf{v}} V$. Since **v** interpreted on the extended language is a strict subset of **e**, it is clearly true that $M =_{\mathbf{e}} V$.

For the proof of the second claim, recall that by Theorem 9.7, $eval_{\mathbf{e}} = eval^{\mathsf{s}}_{\mathbf{e}}$. Hence, assume $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{e}} V$. The assumption implies $V \neq \mathtt{err}_l$. But then none of the standard reduction steps from M to V can be a step based on δ_{err} or **error**. We prove this statement by induction on the length of the standard reduction sequence from M to V:

- **Case** $M = V$
  The claim holds.

- **Case** $M \longmapsto_{\mathbf{e}} M_1 \longmapsto\!\!\!\!\to_{\mathbf{e}} V$
  We know that $M = E[M']$ for some evaluation context E and an $\mathbf{e}$-redex M'. If $M'\,\delta_{\mathsf{err}}\,\mathtt{err}_l$ or $M' = \mathtt{err}_l$, then $M \longmapsto_{\mathbf{e}} \mathtt{err}_l$, contradicting $V \neq \mathtt{err}_l$. Hence, $M \longmapsto_{\mathbf{v}} M_1$, and by induction, $M_1 \longmapsto\!\!\!\!\to_{\mathbf{v}} V$, so the claim holds.

The last two claims follow from the observation that if for some ISWIM program M such that $M \longmapsto\!\!\!\!\to_{\mathbf{e}} N$, then $M \longmapsto\!\!\!\!\to_{\mathbf{v}} N$ and N is in ISWIM if the standard transition reduces a β_{v} or a δ redex. Hence, if $M \longmapsto\!\!\!\!\to_{\mathbf{e}} \mathtt{err}_l$ then $M \longmapsto\!\!\!\!\to_{\mathbf{v}} E[N] \longmapsto_{\mathbf{e}} E[\mathtt{err}_l] \longmapsto_{\mathbf{e}} \mathtt{err}_l$ for some evaluation context E and a faulty application N. Hence, $eval_{\mathbf{v}}(M)$ is undefined. Finally, if $M \longmapsto\!\!\!\!\to_{\mathbf{e}} N$ implies that $N \longmapsto_{\mathbf{e}} L$, then all transitions are according to δ or β_{v}, and thus $eval_{\mathbf{v}}(M)$ is undefined.

A similar question concerning the relationship between the two languages is whether program transformations that are valid for ISWIM are also valid in the extended setting. Technically speaking, we are asking whether an observational equivalence about two ISWIM expressions is still an observational equivalence in Error ISWIM. Not surprisingly, the answer is no. Reducing all stuck expressions to `err` means that all stuck expressions are provably equal, which clearly is not true for languages like Error ISWIM. We begin with the formal definition of observational equivalence for Error ISWIM.

$$M \simeq_{\mathbf{e}} N \quad \text{if} \quad eval_{\mathbf{e}}(C[M]) = eval_{\mathbf{e}}(C[N]) \quad \text{for all } C$$

The following theorem formalizes the discussed relationship between the observational equivalence relations of ISWIM and Error ISWIM.

Theorem 9.9: For and M and N,

1. $M \simeq_{\mathbf{e}} N$ implies $M \simeq_{\mathbf{v}} N$; but
2. $M \simeq_{\mathbf{v}} N$ does not imply $M \simeq_{\mathbf{e}} N$.

Proof for Theorem 9.9: The first part is obvious. Any ISWIM context that observationally separates two ISWIM expressions is also an Error ISWIM context.

For the second part, recall that

$$(\lambda fx.((f\ x)\ \Omega)) \simeq_{\mathbf{v}} (\lambda fx.\Omega)$$

But in Error ISWIM the two expressions are clearly distinguishable with

$$C = ([]\ (\lambda x.\mathtt{err}_l)\ (\lambda x.x)).$$

Whereas $eval_{\mathbf{e}}(C[(\lambda fx.((f\ x)\ \Omega))]) = \mathtt{err}_l$, $eval_{\mathbf{e}}(C[(\lambda fx.\Omega)])$ does not exist.

The proof of the theorem reveals why Error ISWIM can distinguish more ISWIM expressions than ISWIM itself. Some expressions that used to be observationally equivalent to divergence are now equal to errors. But this also means that the only way such expressions can be distinguished is via programs that eventually return errors.

Theorem 9.10: If M and N are such that $M \simeq_{\mathbf{v}} N$ and $M \not\simeq_{\mathbf{e}} N$, then there exists a context C over Error ISWIM such that $C[M]$ and $C[N]$ are programs and one of the following conditions hold:

1. There exist errors $\mathtt{err}_{l_1}$ and $\mathtt{err}_{l_2}$ with $\mathtt{err}_{l_1} \neq \mathtt{err}_{l_2}$ and

$$eval_{\mathbf{e}}(C[M]) = \mathtt{err}_{l_1} \text{ and } eval_{\mathbf{e}}(C[N]) = \mathtt{err}_{l_2}$$

2. There exists an error $\mathtt{err}_{l_1}$ and

$$eval_{\mathbf{e}}(C[M]) = \mathtt{err}_{l_1} \text{ and } eval_{\mathbf{e}}(C[N]) \text{ diverges}$$

3. There exists an error $\mathtt{err}_{l_2}$ and

$$eval_{\mathbf{e}}(C[M]) \text{ diverges and } eval_{\mathbf{e}}(C[N]) = \mathtt{err}_{l_2}$$

Proof for Theorem 9.10: Let M and N be such that $M \simeq_{\mathbf{v}} N$ and $M \not\simeq_{\mathbf{e}} N$. Assume, without loss of generality, that M and N can be distinguished by observing answers distinct from errors. For example, there may exist an Error ISWIM context C such that $C[M]$ and $C[N]$ are programs and for some basic constants b_M and b_N,

$$eval_{\mathbf{e}}(C[M]) = b_1, eval_{\mathbf{e}}(C[N]) = b_2, \text{ and } b_1 \neq b_2$$

Since C is an Error ISWIM context, it may contain several error elements. However, these error elements clearly cannot play any role in the evaluation because once a program is an evaluation context filled with some $\mathtt{err}_l$, the answer must be $\mathtt{err}_l$. Thus, let C' be like C except that all occurrences of error elements are replaced by Ω. But then, by the following lemma,

$$eval_{\mathbf{e}}(C[M]) = eval_{\mathbf{e}}(C'[M]) = b_1$$

and

$$eval_{\mathbf{e}}(C[N]) = eval_{\mathbf{e}}(C'[N]) = b_2,$$

which means that some ISWIM context can already observe differences between M and N: $M \not\simeq_{\mathbf{v}} N$. This contradiction to the assumptions of the lemma proves that the context C cannot exist.

We have left to prove that if a program has a proper answer, occurrences of error elements in the program can be ignored. The corresponding lemma is a version of the Activity Lemma (Lemma 5.9) for Error ISWIM.

Lemma 9.11: Let C be an m-hole context over Error ISWIM. If $eval_{\mathbf{e}}(C[\mathtt{err}_{l_1}, \ldots \mathtt{err}_{l_m}]) = a$ and $a \notin \mathtt{err}_l$, then $eval_{\mathbf{e}}(C[\Omega, \ldots \Omega]) = a$.

Proof for Theorem 9.11: By assumption, $C[\mathtt{err}_{l_1}, \ldots \mathtt{err}_{l_m}] \longmapsto\!\!\!\!\rightarrow_{\mathbf{e}} V$ for some value $V \not\equiv \mathtt{err}_l$. We will prove by induction on the length of the reduction sequence that $C[\Omega, \ldots \Omega]$ reduces to a value. If the original program is a value, the claim clearly holds. Thus, assume that for some program M,

$$C[\mathtt{err}_{l_1}, \ldots \mathtt{err}_{l_m}] \longmapsto_{\mathbf{e}} M \longmapsto\!\!\!\!\rightarrow_{\mathbf{e}} V$$

By the definition of standard reduction, there must be some evaluation context E and standard redex M_{re} such that

$$C[\mathtt{err}_{l_1}, \ldots \mathtt{err}_{l_m}] = E[M_{\mathsf{re}}]$$

The standard redex M_{re} cannot be an error element and it cannot be a δ_{err} redex because both would contradict the assumption. But then M_{re} reduces to some contractum N such that for some n-ary context C',

$$M = C'[\mathtt{err}_{l'_1}, \ldots \mathtt{err}_{l'_n}]$$

where $1'_1, \ldots l'_n$ is a (possibly repetitive) permutation of $l_1, \ldots l_m$. This clearly means that

$$C[\Omega, \ldots \Omega] \longmapsto_{\mathbf{e}} C'[\Omega, \ldots \Omega]$$

and, by inductive hypothesis,

$$C[\Omega, \ldots \Omega] \longmapsto_{\mathbf{e}} V'$$

for some value V'. Moreover, if V was a basic constant, then $V' = V$ and if V was a λ-abstraction then so is V'. Hence,

$$eval_{\mathbf{e}}(C[\Omega, \ldots \Omega]) = a$$

▷ **Exercise 9.6.** Suppose that we weaken the evaluator so that it does not distinguish result errors:

$$eval_{\hat{\mathbf{e}}}(M) = \begin{cases} b & \text{if } M =_{\mathbf{e}} b \\ \texttt{function} & \text{if } M =_{\mathbf{e}} \lambda X.N \\ \texttt{err} & \text{if } M =_{\mathbf{e}} \texttt{err}_l \end{cases}$$

$$M \simeq_{\hat{\mathbf{e}}} N \quad \text{if} \quad eval_{\hat{\mathbf{e}}}(C[M]) = eval_{\hat{\mathbf{e}}}(C[N]) \quad \text{for all } C$$

Is $\simeq_{\hat{\mathbf{e}}}$ also weakened, in the sense that $M \simeq_{\hat{\mathbf{e}}} N$ does not imply $M \simeq_{\mathbf{e}} N$?

▷ **Exercise 9.7.** Use the proof of Theorem 9.9 to show that Error ISWIM can reveal the order in which a procedure invokes its parameters that are procedures. Hint: Separate the cases where $\texttt{err}_l$ contains only $\texttt{err}_0$, and the case where $\texttt{err}_l \supseteq \{\texttt{err}_{l_1}, \texttt{err}_{l_2}\}$ with $\texttt{err}_{l_1} \neq \texttt{err}_{l_2}$.

▷ **Exercise 9.8.** Use Theorem 9.1 to prove that for any ISWIM expressions M and N, $M =_{\mathbf{v}} N$ implies $M =_{\mathbf{e}} N$.

Conclude that for all M and N, $M =_{\mathbf{e}} N$ implies $M \simeq_{\mathbf{v}} N$. Hint: use the Church-Rosser theorem for ISWIM, Theorem 4.4.

Also prove that $M =_{\mathbf{e}} N$ implies $M \simeq_{\mathbf{e}} N$. Hint: compare to Theorem 4.10.

9.2 Exceptions and Handlers

In addition to constructs for signaling errors, many programming languages provide facilities for dealing with errors that arise during the evaluation of a sub-expression. Such constructs are referred to as **error handlers** or **exception handlers**. A typical syntax for error handlers is

$$(\texttt{catch } M_1 \texttt{ with } \lambda X.M_2)$$

where the expression M_1 is the protected **body** and $\lambda X.M_2$ is the **handler**. The evaluation of such an expression starts with the body. If the body returns some value, the value becomes the answer of the complete expression. If the evaluation of the body signals an error, the handler is used to determine a result.

Exception handlers are useful in many situations. For example, the evaluation of an arithmetic expression may signal a division-by-zero error or an overflow error, yet it may suffice to return a symbolic constant `infinity` if this happens. Similarly, if there are two methods for computing a result, but the first and faster one may signal an exception; a program can attempt to compute the answer with the first method, but when an exception occurs, control is transferred to the second method of computation.

Provided that exceptions come with additional information, the exception handler's action may also depend on what kind of exception occured. Indeed, this option increases the usefulness of exception handlers dramatically, because the handler not only knows that an exception occurred, but also what kind of exception occurred. We will assume that the exceptions are represented as basic constants, and that the basic constant in an `throw` application is the information that a handler receives.

Handler ISWIM extends plain ISWIM with `throw` and `catch` forms:

M	$=$	$\ldots$	same as plain ISWIM
	$\mid$	$(\texttt{throw}\ b)$	
	$\mid$	$(\texttt{catch}\ M\ \texttt{with}\ \lambda X.M)$	
C	$=$	$\ldots$	same as plain ISWIM
	$\mid$	$(\texttt{catch}\ C\ \texttt{with}\ \lambda X.M)$	
	$\mid$	$(\texttt{catch}\ M\ \texttt{with}\ \lambda X.C)$	

For an illustration of the versatility of exception handlers, we consider a procedure that multiplies the elements of a list of numbers. A naive version of the procedure traverses the list and multiplies the elements one by one:

$$\Pi \doteq \mathsf{Y_V}(\lambda\pi.\ \lambda l.\quad (\mathsf{if0}\quad (\mathsf{isnull}\ l)\ \ulcorner 1\urcorner \\ (*\ (\mathsf{car}\ l)\ (\pi\ (\mathsf{cdr}\ l)))))$$

In Handler ISWIM, the procedure can check for an occurrence of $\ulcorner 0\urcorner$ in the list and can exit the potentially deep recursion immediately, because the result will be $\ulcorner 0\urcorner$. Exiting can be implemented by raising an exception that is handled by a surrounding handler:

$$\Pi^0 \doteq \lambda l.(\texttt{catch}\ (\Pi'\ l)\ \texttt{with}\ \lambda x.x)$$

where

$$\Pi' \doteq \mathsf{Y_V}(\lambda\pi.\ \lambda l.\quad (\mathsf{if0}\quad (\mathsf{isnull}\ l)\qquad\qquad\qquad \ulcorner 1\urcorner \\ (\mathsf{if0}\quad (\mathsf{car}\ l)\qquad\qquad\qquad (\texttt{throw}\ \ulcorner 0\urcorner) \\ (*\ (\mathsf{car}\ l)\ (\pi\ (\mathsf{cdr}\ l)))))$$

In this example, signaling an exception indicates that the proper result was found, and thus speeds up the ordinary computation.

9.2.1 Calculating with Handler ISWIM

The calculus for Handler ISWIM must clearly incorporate the basic axioms for procedures and primitive operations of the Error ISWIM calculus. Also, errors should still eliminate pending computations except for error handlers. However, the notion of a pending computation has changed, and it now depends on the behavior of `catch` expressions. During the evaluation of the protected body, the rest of the evaluation temporarily means the rest of the evaluation of the body. If this evaluation process returns a value, this value is the result of the entire `catch` expression. Othwerwise, if the second evaluation signals an error, the error is not propagated; instead, the handler is applied to the basic constant associated with the error.

The formal characterization of this sketch is straightforward. Returning a value from the evaluation of the body corresponds to a notion of reduction that returns the first sub-expression of a handler expression when both sub-expressions are values:

$$(\texttt{catch}\ U\ \texttt{with}\ \lambda X.M) \to U$$

A notion of reduction that describes the error handling component of the behavior says that if the first sub-expression reduces to a `throw` expression, its basic constant becomes the argument of the handler:

$$(\texttt{catch}\ (\texttt{throw}\ b)\ \texttt{with}\ \lambda X.M) \to ((\lambda X.M)\ b)$$

No other reductions dealing with `catch` expressions are needed. The second notion of reduction also points out that the hole of an evaluation context, which represents the pending computation of a redex, cannot be inside the body of a `catch` expression. Otherwise, error propagation through evaluation contexts would circumvent error handlers.

The notions of reduction for error elements have the almost same shape as the ones for Error ISWIM. The following defines the calculus for Handler ISWIM. To deal with exceptions raised by primitive operators, we assume a bijection $\mathcal{EM}$ mapping members of $\mathtt{err}_l$ to members of b, and we change δ_{err} accordingly. We also change the evaluation function to include information about the basic constant raised for an escaping exception.

$$
\begin{array}{llll}
(o^m\ b_1 \ldots\ b_m) & \delta_{\mathsf{err}} & (\mathtt{throw}\ b) & \text{if } \delta(o^m, b_1, \ldots b_m) = \mathtt{err}_l \\
 & & & \text{and where } b = \mathcal{EM}(\mathtt{err}_l) \\
(o^m\ V_1 \ldots\ (\lambda X.N)\ \ldots\ V_m) & \delta_{\mathsf{err}} & (\mathtt{throw}\ b) & \text{where } b = \mathcal{EM}(\mathtt{err}_{o^m}) \\
(b\ V) & \delta_{\mathsf{err}} & (\mathtt{throw}\ b') & \text{where } b' = \mathcal{EM}(\mathtt{err}_b)
\end{array}
$$

$$
\begin{array}{lll}
E[(\mathtt{throw}\ b)] & \mathbf{throw} & (\mathtt{throw}\ b) \\
(\mathtt{catch}\ V\ \mathtt{with}\ \lambda X.M) & \mathbf{return} & V \\
(\mathtt{catch}\ (\mathtt{throw}\ b)\ \mathtt{with}\ \lambda X.M) & \mathbf{catch} & ((\lambda X.M)\ b)
\end{array}
$$

$$\mathbf{h} = \beta_{\mathsf{v}} \cup \delta \cup \delta_{\mathsf{err}} \cup \mathbf{throw} \cup \mathbf{return} \cup \mathbf{catch}$$

$$A_{\mathbf{h}} = b \cup \{\mathtt{function}\} \cup \mathtt{err}_b$$

$$eval_{\mathbf{h}}(M) = \begin{cases} b & \text{if } M =_{\mathbf{h}} b \\ \mathtt{function} & \text{if } M =_{\mathbf{h}} \lambda X.N \\ \mathtt{err}_b & \text{if } M =_{\mathbf{h}} (\mathtt{throw}\ b) \end{cases}$$

9.2.2 Consistency for Handler ISWIM

Theorem 9.12 [Consistency with Handlers]: The relation $eval_{\mathbf{h}}$ is a partial function.

Proof for Theorem 9.12: The relations **catch** and **return** trivially satisfy the diamond property. It is also straightforward to show that the proof of the dimaond property for the notion of reduction **e** (but using **throw** instead of **error**) extends to the full syntax. If the reduction based on *eltn* commutes with the one based on **catch** and **return**, we have the diamond property for the full language. To prove the commutativity of **catch** and **return** with the old reductions, there are two cases

- **Case** $C[(\mathtt{catch}\ (\mathtt{throw}\ b)\ \mathtt{with}\ \lambda X.M)] \to C[((\lambda X.M)\ b)]$
  Then, a reduction step according to one of δ, β_{v}, δ_{err}, or **r** either modifies the context C or the handler V:
  - **Case** $C[(\mathtt{catch}\ (\mathtt{throw}\ b)\ \mathtt{with}\ \lambda X.M)]$
    $\to C'[(\mathtt{catch}\ (\mathtt{throw}\ b)\ \mathtt{with}\ \lambda X.M)]$
    The common contractum for $C'[(\mathtt{catch}\ \mathtt{throw}\ b\ \mathtt{with}\ \lambda X.M)]$ and $C[((\lambda X.M)\ b)]$ is $C'[((\lambda X.M)\ b)]$.
  - **Case** $C[(\mathtt{catch}\ (\mathtt{throw}\ b)\ \mathtt{with}\ \lambda X.M)]$
    $\to C[(\mathtt{catch}\ (\mathtt{throw}\ b)\ \mathtt{with}\ \lambda X.M')]$
    In this case, the expressions reduce to $C[((\lambda X.M')\ b)]$.

  In both cases, the reductions to the common contractum always require one step, which shows that the one-step reduction relations and hence their transitive closures commute.

- **Case** $C[(\texttt{catch}\ U\ \texttt{with}\ \lambda X.M)] \to C[U]$
  In this case, a reduction can affect C, U, and V, which requires an analysis of three subcases. Otherwise the proof of this case proceeds as the first one.

The consistency of the calculus and the functionhood of $eval_{\mathbf{h}}$ directly follow from the diamond property according to the familiar proof strategy.

9.2.3 Standard Reduction for Handler ISWIM

The definition of a deterministic evaluation function for Handler ISWIM can no longer rely on the simple algorithm for ISWIM and Error ISWIM. We need a revised notion of evaluation contexts, because we need to be able to reduce the body expression in

$$(\texttt{catch}\ \underline{((\lambda x.x)\ \ulcorner 0 \urcorner)}\ \texttt{with}\ \lambda x.x)$$

However, we do not want to extend the definition of E with $(\texttt{catch}\ E\ \texttt{with}\ \lambda X.M)$, because that would allow

$$(\texttt{catch}\ \underline{(\texttt{throw}\ b)}\ \texttt{with}\ \lambda x.x) \to (\texttt{throw}\ b)$$

by **throw**.

The solution is to define a new kind of evaluation context, E_{h}. Meanwhile, unlike in ISWIM or Error ISWIM, E contexts do not represent the entire rest of a computation relative to a redex. Instead, they only represent rest of the computation up to the closest `catch` expression. An E_{h} context will represent the rest of the computation, including all the pending `catch` expressions.

$$\begin{array}{rcl} E_{\mathsf{h}} & = & [\,] \\ & | & (E_{\mathsf{h}}\ M) \\ & | & (V\ E_{\mathsf{h}}) \\ & | & (o^m\ V_1\ \ldots V_{i-1}\ E_{\mathsf{h}}\ M_{i+1}\ \ldots\ M_m) \\ & | & (\texttt{catch}\ E_{\mathsf{h}}\ \texttt{with}\ \lambda X.M) \end{array}$$

As with Error ISWIM, we must define the standard reduction relation so that it avoids ambiguities due to multiple matching **throw** reductions. Instead of one special rule, two are needed; one for an uncaught exception, and one within a `catch` expression:

$$\begin{array}{lcl} E[(\texttt{throw}\ b)] & \longmapsto_{\mathbf{h}} & (\texttt{throw}\ b) \\ E_{\mathsf{h}}[(\texttt{catch}\ E[(\texttt{throw}\ b)] & \longmapsto_{\mathbf{h}} & E_{\mathsf{h}}[(\texttt{catch}\ (\texttt{throw}\ b) \\ \qquad \texttt{with}\ \lambda X.M)] & & \qquad\qquad \texttt{with}\ \lambda X.M)] \end{array}$$

The complete definition follows.

$$\tilde{\mathbf{h}} = \beta_{\mathsf{v}} \cup \delta \cup \delta_{\mathsf{err}} \cup \mathbf{return} \cup \mathbf{catch}$$

$$\begin{array}{lcll} E_{\mathsf{h}}[M] & \longmapsto_{\mathbf{h}} & E_{\mathsf{h}}[N] & \text{if } M\ \tilde{\mathbf{h}}\ N \\ E[(\texttt{throw}\ b)] & \longmapsto_{\mathbf{h}} & (\texttt{throw}\ b) & \\ E_{\mathsf{h}}[(\texttt{catch}\ E[(\texttt{throw}\ b)] & \longmapsto_{\mathbf{h}} & E_{\mathsf{h}}[(\texttt{catch}\ (\texttt{throw}\ b) & \\ \qquad \texttt{with}\ \lambda X.M)] & & \qquad\qquad \texttt{with}\ \lambda X.M)] & \end{array}$$

$$eval^{\mathsf{s}}_{\mathbf{h}}(M) = \begin{cases} b & \text{if } M \longmapsto\!\!\!\!\to_{\mathbf{h}} b \\ \texttt{function} & \text{if } M \longmapsto\!\!\!\!\to_{\mathbf{h}} \lambda X.N \\ \texttt{err}_b & \text{if } M \longmapsto\!\!\!\!\to_{\mathbf{h}} (\texttt{throw}\ b) \end{cases}$$

The correctness proof of this new specification of the evaluator consists of the usual steps, though it is more complex to show the "unique partitioning" lemma. The redex grammar M_{rh} for the lemma omits `throw` expressions.

$$\begin{array}{rcl} M_{\mathsf{rh}} & = & (V\ V) \\ & | & (o^m\ V \ldots\ V) \\ & | & (\texttt{catch}\ V\ \texttt{with}\ \lambda X.M) \\ & | & (\texttt{catch}\ (\texttt{throw}\ b)\ \texttt{with}\ \lambda X.M) \end{array}$$

Lemma 9.13 [Unique Standard Contexts]: For every closed Handler ISWIM expression M, one of the following is true for a unique E_{h}, E, and M_{rh}:

- M is a value.
- There exist a standard context E_{h} and a standard redex M_{rh} such that $M = E_{\mathsf{h}}[M_{\mathsf{rh}}]$.
- There exists a context E such that $M = E[(\texttt{throw}\ b)]$.
- There exist a standard context E_{h} and context E such that $M = E_{\mathsf{h}}[(\texttt{catch}\ E[(\texttt{throw}\ b)]\ \texttt{with}\ \lambda X.M)]$.

Second, we must show that the evaluator based on the standard reduction function is equal to the evaluator based on the calculus. We state the theorem without proof.

Theorem 9.14: $eval^{\mathbf{s}}_{\mathbf{h}} = eval_{\mathbf{h}}$

9.2.4 Observational Equivalence of Handler ISWIM

Observational equivalence of Handler ISWIM is defined as usual.

$$M \simeq_{\mathbf{h}} N \quad \text{if} \quad eval_{\mathbf{h}}(C[M]) = eval_{\mathbf{h}}(C[N]) \quad \text{for all } C$$

It is obvious that any ISWIM expressions that can be distinguished in Error ISWIM can also be distinguished in Handler ISWIM. Moreover, the result of these distinguishing programs does not have to be an error anymore. With error handlers, all error results can be turned into basic constants.

Theorem 9.15: $M \simeq_{\mathbf{e}} N$ implies $M \simeq_{\mathbf{h}} N$. Moreover, the analogue of Theorem 9.10 does not hold for Handler ISWIM.

Proof for Theorem 9.15: The first part is trivial. For the second part, assume that C is a distinguishing context in Error ISWIM for M and N. Take

$$C' = (\texttt{catch}\ C\ \texttt{with}\ \lambda x.x)$$

as the distinguishing context in Handler ISWIM.

Conjecture 9.16: $M \simeq_{\mathbf{h}} N$ implies $M \simeq_{\mathbf{e}} N$.

Conjecture 9.17: $\{(M,N) \mid M \simeq_{\mathbf{v}} N \text{ and } M \not\simeq_{\mathbf{e}} N\}$ is the same as $\{(M,N) \mid M \simeq_{\mathbf{v}} N \text{ and } M \not\simeq_{\mathbf{h}} N\}$.

Note: The first conjecture implies the second.

▷ **Exercise 9.9.** Suppose that we weaken the evaluator so that it does not distinguish result errors:

$$eval_{\hat{\mathbf{h}}}(M) = \begin{cases} b & \text{if } M =_{\mathbf{h}} b \\ \texttt{function} & \text{if } M =_{\mathbf{h}} \lambda X.N \\ \texttt{err} & \text{if } M =_{\mathbf{h}} (\texttt{throw}\ b) \end{cases}$$

$$M \simeq_{\hat{\mathbf{h}}} N \quad \text{if} \quad eval_{\hat{\mathbf{h}}}(C[M]) = eval_{\hat{\mathbf{h}}}(C[N]) \quad \text{for all } C$$

Is $\simeq_{\hat{\mathbf{h}}}$ also weakened, in the sense that $M \simeq_{\hat{\mathbf{h}}} N$ does not imply $M \simeq_{\mathbf{h}} N$?

▷ **Exercise 9.10.** Here is an even weaker evaluator for Handler ISWIM:

$$eval_3(M) = \begin{cases} b & \text{if } M =_{\mathbf{h}} b \\ \texttt{function} & \text{if } M =_{\mathbf{h}} \lambda X.N \\ b & \text{if } M =_{\mathbf{h}} (\texttt{throw}\ b) \end{cases}$$

It differs from the original evaluator in that it maps all programs to basic constants or `function`. In particular, the modified evaluator ignores the difference between errors and basic constants.

Check that the modified evaluator is still a function. Are the two observational equivalence relations, defined by $eval_{\mathbf{h}}$ and $eval_3$, the same?

9.3 Machines for Exceptions

As we saw in previous chapters, a textual abstract machine, like the standard reduction function, is a good tool for understanding and analyzing the mechanical behavior of programs. It is more algorithmic than a calculus, yet it avoids the use of auxiliary data structures like activation records, stacks, etc. For a good implementation, though, is still too abstract, hiding too many inefficiencies, so we have also looked at more concrete machines, such as the CC machine. Starting from the standard reduction function for Handler ISWIM, we briefly sketch the changes to the CC machine for ISWIM that are necessary to accomodate errors and error handlers.

9.3.1 The Handler-Extended CC Machine

The rules for $\longmapsto_{\mathbf{h}}$ in Section 9.2.3 (page 101) are an extension of the rules for $\longmapsto_{\mathbf{v}}$. The basic difference is that $\longmapsto_{\mathbf{h}}$ distinguishes two notions of the rest of a computation; an evaluation context represents the rest of the computation up to the closest `catch` expression, and a standard redex denotes the rest of the entire computation relative to some standard redex.

A naive adaptation of the strategy of the CC machine suggests a separation of the standard context from the control string to avoid the repeated partitioning task. Machine states for the revised CC machine will be pairs consisting of closed expressions and standard contexts. The instructions shift information from the control string to the context until a standard redex is found. If the standard redex is a δ- or a β_{v}-redex, the machine proceeds as before. If the redex is a δ_{err}-redex, the machine places an appropriate `throw` expression in its control register. Finally, if the control string is an `err` expression, then the machine erases an appropriate portion of the standard context: up to the closest handler, if it exists, or the entire context otherwise.

The following rules extend a variant of $\longmapsto_{\mathsf{cc}}$ where each E in the original set of rules is replaced by E_{h}:

$$\begin{array}{lcll}
\langle M, E_{\mathsf{h}}\rangle & \longmapsto_{\mathsf{cc}} & \langle (\texttt{throw}\ b), E_{\mathsf{h}}\rangle & [\mathsf{cc7}] \\
\quad \text{if } M\ \delta_{\mathsf{err}}\ (\texttt{throw}\ b) & & & \\
\langle (\texttt{throw}\ b), E\rangle & \longmapsto_{\mathsf{cc}} & \langle (\texttt{throw}\ b), [\,]\rangle & [\mathsf{cc8}] \\
\langle (\texttt{catch}\ M\ \texttt{with}\ \lambda X.N), E_{\mathsf{h}}\rangle & \longmapsto_{\mathsf{cc}} & \langle M, E_{\mathsf{h}}[(\texttt{catch}\ [\,]\ \texttt{with}\ \lambda X.N)]\rangle & [\mathsf{cc9}] \\
\langle V, E_{\mathsf{h}}[(\texttt{catch}\ [\,]\ \texttt{with}\ \lambda X.N)]\rangle & \longmapsto_{\mathsf{cc}} & \langle V, E_{\mathsf{h}}\rangle & [\mathsf{cc10}] \\
\langle (\texttt{throw}\ b), E_{\mathsf{h}}[(\texttt{catch}\ E\ \texttt{with}\ \lambda X.N)]\rangle & \longmapsto_{\mathsf{cc}} & \langle (\texttt{throw}\ b), E_{\mathsf{h}}[(\texttt{catch}\ [\,]\ \texttt{with}\ \lambda X.N)]\rangle & [\mathsf{cc11}] \\
\langle (\texttt{throw}\ b), E_{\mathsf{h}}[(\texttt{catch}\ [\,]\ \texttt{with}\ \lambda X.N)]\rangle & \longmapsto_{\mathsf{cc}} & \langle ((\lambda X.N)\ b), E_{\mathsf{h}}\rangle & [\mathsf{cc12}]
\end{array}$$

$$eval_{\mathsf{cc+h}}(M) = \begin{cases} b & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\to_{\mathsf{cc}} \langle b, [\,]\rangle \\ \texttt{function} & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\to_{\mathsf{cc}} \langle \lambda X.N, [\,]\rangle \\ \texttt{err}_b & \text{if } \langle M, [\,]\rangle \longmapsto\!\!\!\!\to_{\mathsf{cc}} \langle (\texttt{throw}\ b), [\,]\rangle \end{cases}$$

For a proof of correctness of this machine, it suffices to check that the six new transitions faithfully implement the five new standard reduction transitions. This check is a straightforward exercise.

Theorem 9.18: $eval_{\mathsf{cc+h}} = eval^{\mathsf{s}}_{\mathbf{h}}$

▷ **Exercise 9.11.** Prove Theorem 9.18.

▷ **Exercise 9.12.** Prove that it is possible to eliminate [cc8] by wraping the initial program in a handler.

9.3.2 The CCH Machine

One major bottleneck of the naive CC machine for Handler ISWIM is obvious. Although the machine maintains the standard context as a separate data structure, the transition implementing **error** must perform a complex analysis of this data structure. More precisely, it must partition the standard context to determine the "current" evaluation context, that is, the largest possible evaluation context surrounding the hole. This traversal of the standard context, however, is a repetition of the traversal that the machine performed in the search of the redex. If it kept the current evaluation context around as a separate data structure, the **error** instruction could simply replace the current evaluation context with an empty context.

The idea of separating the "current" evaluation context from the rest of the standard context suggests that the entire standard context be represented as a stack of evaluation contexts. Each element is the evaluation context between two currently active `catch` expressions. In order to handle errors correctly, the evaluation contexts should be paired with their respective handlers. When the machine encounters a `catch` expression, it pushes the current evaluation context and the new handler on the stack:

$$\langle (\texttt{catch}\ M\ \texttt{with}\ \lambda X.N), E, H\rangle \to \langle M, [\,], \langle \lambda X.N,\ E\rangle\ H\rangle$$

When the body of a `catch` expression is eventually reduced to a value, the machine pops the stack

$$\langle V, [\,], \langle \lambda X.N,\ E\rangle H\rangle \to \langle V, E, H\rangle$$

that is, it throws away the current handler and reinstalls the surrounding evaluation context. The instruction for handling errors is analoguous. The H register is a stack of handler–context pairs:

$$\begin{array}{rcl} H & = & \epsilon \\ & | & \langle \lambda X.M,\ E\rangle\ H \end{array}$$

We call the new machine the **CCH machine**, where the "H" stands for "handler stack." The first six CC instructions, the ones for for plain ISWIM expressions, essentially remain the same; they do not use the additional register in any way and do not alter it. (The handler-extended CC machine uses E_{h} in place of E for the original CC rules, but the CCH machine continues to use E in the rules, because the handler part of the context has been moved into the handler stack, H.)

$$
\begin{array}{lcll}
\langle (MN), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle M, E[([\,]\ N)], H\rangle & [\mathsf{cch1}] \\
\quad \text{if } M \notin V & & & \\
\langle (VM), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle M, E[(V\ [\,])], H\rangle & [\mathsf{cch2}] \\
\quad \text{if } M \notin V & & & \\
\langle (o^n\ V_1 \ldots V_i\ M\ N \ldots), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle M, E[(o^n\ V_1 \ldots V_i\ [\,]\ N \ldots)], H\rangle & [\mathsf{cch3}] \\
\quad \text{if } M \notin V & & & \\
\langle ((\lambda X.M)\ V), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle M[X \leftarrow V], E, H\rangle & [\mathsf{cch}\beta_{\mathsf{v}}] \\
\langle (o^m\ b_1 \ldots\ b_m), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle V, E, H\rangle & [\mathsf{cch}\delta_{\mathsf{v}}] \\
 & & \quad \text{where } \delta(o^m, b_1, \ldots b_m) = V & \\
\langle V, E[(U\ [\,])], H\rangle & \longmapsto_{\mathsf{cch}} & \langle (U\ V), E, H\rangle & [\mathsf{cch4}] \\
\langle V, E[([\,]\ N)], H\rangle & \longmapsto_{\mathsf{cch}} & \langle (V\ N), E, H\rangle & [\mathsf{cch5}] \\
\langle V, E[(o^n\ V_1 \ldots V_i\ [\,]\ N \ldots)], H\rangle & \longmapsto_{\mathsf{cch}} & \langle (o^n\ V_1 \ldots V_i\ V\ N \ldots), E, H\rangle & [\mathsf{cch6}] \\
\langle M, E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle (\texttt{throw}\ b), E, H\rangle & [\mathsf{cch7}] \\
\quad \text{if } M\delta_{\mathsf{err}}(\texttt{throw}\ b) & & & \\
\langle (\texttt{throw}\ b), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle (\texttt{throw}\ b), [\,], H\rangle & [\mathsf{cch8}] \\
\langle (\texttt{catch}\ M\ \texttt{with}\ \lambda X.N), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle M, [\,], \langle \lambda X.N,\ E\rangle\ H\rangle & [\mathsf{cch9}] \\
\langle V, [\,], \langle U,\ E\rangle\ H]\rangle & \longmapsto_{\mathsf{cch}} & \langle V, E, H\rangle & [\mathsf{cch10}] \\
\langle (\texttt{throw}\ b), E, H\rangle & \longmapsto_{\mathsf{cch}} & \langle (\texttt{throw}\ b), [\,], H\rangle & [\mathsf{cch11}] \\
\langle (\texttt{throw}\ b), [\,], \langle U,\ E\rangle\ H\rangle & \longmapsto_{\mathsf{cch}} & \langle (U\ b), E, H\rangle & [\mathsf{cch12}]
\end{array}
$$

$$
eval_{\mathsf{cch}}(M) = \begin{cases} b & \text{if } \langle M, [\,], \epsilon\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cch}} \langle b, [\,], \epsilon\rangle \\ \texttt{function} & \text{if } \langle M, [\,], \epsilon\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cch}} \langle \lambda X.N, [\,], \epsilon\rangle \\ \texttt{err}_b & \text{if } \langle M, [\,], \epsilon\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cch}} \langle (\texttt{throw}\ b), [\,], \epsilon\rangle \end{cases}
$$

For a correctness proof of the CCH machine, we must find a relationship between the states of the extended CC machine and those of the CCH machine. The idea for the conversions is the above-mentioned representation invariant: the standard context of a CC machine state is represented as a stack of handlers and evaluation contexts. Conversely, given such a stack, we can simply create a single standard context that creates the stack. Based on this idea, it is easy to prove the equivalence between the two machines.

Theorem 9.19: $eval_{\mathsf{cch}} = eval_{\mathsf{cc+h}}$

▷ **Exercise 9.13.** Prove Theorem 9.19.

▷ **Exercise 9.14.** Implement the extended CC and CCH machines.

▷ **Exercise 9.15.** Derive extended CK and CEK machines based on the CCH machine.

▷ **Exercise 9.16.** Design a modified CCH machine that reserves a new register for the current error handler. Prove its equivalence with the above CCH machine.

Chapter 10: Imperative Assignment

In an imperative language, implementing a loop typically requires variable assignment. For example, the following C implementation of `sum` assigns to the variables `n` and `i`:

```
int sum(int n) {
  int i = 0;
  while (n > 0) {
    i = i + n;
    n = n - 1;
  }
  return i;
}
```

In a purely functional style, we can implement the same loop using tail recursion and accumulator arguments, as in the following Scheme function:

```
(define (sum n i)
  (if (zero? n)
      i
      (sum (- n 1) (+ i n))))
```

In general, however, state is not so easily eliminated. Adding assignments or mutation operators provides a whole new set of interesting programming paradigms, especially when added to languages with higher-order procedures and complex data structures. In such a language, programs can create, manipulate, and alter cyclic data structures and recursive nestings of procedures. A program can also create a kind of "object": a bundle consisting of data and procedures for changing the data in the bundle, where other parts of the program can only inspect or change the data through the procedures.

In this chapter, we extend ISWIM to accomodate state directly. The encoding of state for loops, as above, suggests a way of handling state in a machine that manipulates text instead of memory locations. Just as the state in `n` and `i` from the C program above is emulated by passing arround `n` and `i` Scheme variables, we will model state in general as an entity that is passed around in the evaluator.

The idea of state flows most naturally from a machine point of view, so we approach a model of state in terms of a machine.

10.1 Evaluation with State

To model state, we extend the syntax of ISWIM expressions with a assignment expression:

$$\begin{array}{rcll} M & = & \ldots & \text{same as plain ISWIM} \\ & | & (:= \ X \ M) & \end{array}$$

When a language provides assignment statements for identifiers, the nature of identifiers no longer resembles that of mathematical variables. A mathematical variable represents some fixed value, but assignable identifiers denote a varying association between names and values. We call the association of an identifier with a current value as its **state** and refer to the execution of assignments as **state changes**.

Evaluating the expression $((\lambda x.((\lambda y.x)\ (:=\ \ x\ (+\ x\ \ulcorner 1\urcorner))))\ \ulcorner 12\urcorner)$ must therefore proceed roughly like the following:

$$
\begin{array}{lll}
 & ((\lambda x.((\lambda y.x)\ (:=\ x\ (+\ x\ \ulcorner 1\urcorner))))\ \ulcorner 12\urcorner) & \\
\rightarrow & ((\lambda y.x)\ (:=\ x\ (+\ x\ \ulcorner 1\urcorner))) & \text{with} \quad x = \ulcorner 12\urcorner \\
\rightarrow & ((\lambda y.x)\ (:=\ x\ (+\ \ulcorner 12\urcorner\ \ulcorner 1\urcorner))) & \text{with} \quad x = \ulcorner 12\urcorner \\
\rightarrow & ((\lambda y.x)\ (:=\ x\ \ulcorner 13\urcorner)) & \text{with} \quad x = \ulcorner 12\urcorner \\
\rightarrow & ((\lambda y.x)\ \texttt{void}) & \text{with} \quad x = \ulcorner 13\urcorner \\
\rightarrow & x & \text{with} \quad x = \ulcorner 13\urcorner, y = \texttt{void} \\
\rightarrow & \ulcorner 13\urcorner &
\end{array}
$$

From the above example, we get two clues about how to model state:

- We cannot use simple substitution when applying a function to an argument. For example, if we replaced every x with $\ulcorner 12\urcorner$, we would end up with an ill-formed expression $(:=\ \ulcorner 12\urcorner\ (+\ \ulcorner 12\urcorner\ \ulcorner 1\urcorner))$, and we wouldn't know which $\ulcorner 12\urcorner$s to replace with $\ulcorner 13\urcorner$s later.
- Instead of rewriting an expression to an expression on each evluation step, we rewrote a pair on each step: an expression and a mapping from variables to values.

We might try to model := using an expression–environment pair, but our experience with CEK suggests that we shouldn't try to think of the variable mapping as exactly an environment; to deal with closures, we made a textual "copy" of the environment in each closure. Such copies would defeat the central mapping in the above example that made assignment work, because $(\lambda x.y)$ would have its own separate environment for x.

The solution is to think of memory as a new slot in the machine. Function application selects an unused slot in memory, and replaces all instances of the variable with the slot address. Meanwhile, variable lookup is replaced by slot lookup, and assignment corresponds to changing the value of a slot. The collection of slots is called a **store**.

To refine our above sketch, we rewrite expression-store pairs using a set of slot names σ:

$$
\begin{array}{lllll}
 & \langle & ((\lambda x.((\lambda y.x)\ (:=\ x\ (+\ x\ \ulcorner 1\urcorner))))\ \ulcorner 12\urcorner) & , \quad \{\} & \rangle \\
\rightarrow & \langle & ((\lambda y.\sigma_1)\ (:=\ \sigma_1\ (+\ \sigma_1\ \ulcorner 1\urcorner))) & , \quad \{\langle \sigma_1, \ulcorner 12\urcorner \rangle\} & \rangle \\
\rightarrow & \langle & ((\lambda y.\sigma_1)\ (:=\ \sigma_1\ (+\ \ulcorner 12\urcorner\ \ulcorner 1\urcorner))) & , \quad \{\langle \sigma_1, \ulcorner 12\urcorner \rangle\} & \rangle \\
\rightarrow & \langle & ((\lambda y.\sigma_1)\ (:=\ \sigma_1\ \ulcorner 13\urcorner)) & , \quad \{\langle \sigma_1, \ulcorner 12\urcorner \rangle\} & \rangle \\
\rightarrow & \langle & ((\lambda y.\sigma_1)\ \texttt{void}) & , \quad \{\langle \sigma_1, \ulcorner 13\urcorner \rangle\} & \rangle \\
\rightarrow & \langle & \sigma_1 & , \quad \{\langle \sigma_1, \ulcorner 13\urcorner \rangle, \langle \sigma_2, \texttt{void} \rangle\} & \rangle \\
\rightarrow & \langle & \ulcorner 13\urcorner & , \quad \{\langle \sigma_1, \ulcorner 13\urcorner \rangle, \langle \sigma_2, \texttt{void} \rangle\} & \rangle
\end{array}
$$

With this approach, we must extend ISWIM again to allow store addresses as expressions, and also to support the `void` value generated by an assignment. We do not expect an evaluatable program to contain σ or `void` initially, but reduction steps create store addresses and `void` in expression positions.

$$
\begin{array}{lcll}
M & = & \ldots & \text{same as plain ISWIM} \\
 & | & (:=\ X\ M) & \\
 & | & \sigma & \\
 & | & (:=\ \sigma\ M) & \\
 & | & \texttt{void} & \\
V & = & \ldots & \text{same as plain ISWIM} \\
 & | & \texttt{void} & \\
C & = & \ldots & \text{same as plain ISWIM} \\
 & | & (:=\ X\ C) & \\
E & = & \ldots & \text{same as plain ISWIM} \\
 & | & (:=\ X\ E) & \\
\sigma & = & \text{a store address} & \\
\Sigma & = & \text{a function } \{\langle \sigma, V \rangle, \ldots\} &
\end{array}
$$

$$
\mathcal{FV}(\,(:=\ X\ M)\,) \quad = \quad \{X\} \cup \mathcal{FV}(M)
$$

Note that a store address is *not* a value, which means that when a variable (replaced by a store address) is used as an argument to a function, we must extract the value from the store before applying the function. That's the behavior we want for a call-by-value language, as opposed to a call-by-reference language.

To define evaluation, we define the **CS machine**, which transforms expression-store pairs $\langle M, \Sigma\rangle$:

$$
\begin{array}{lll}
\langle E[((\lambda X.M)\ V)], \Sigma\rangle & \longmapsto_{\mathsf{cs}} & \langle E[M[X \leftarrow \sigma]], \Sigma[\sigma \leftarrow V]\rangle \quad [\mathsf{cs}\beta_\mathsf{s}] \\
 & & \quad \text{where } \sigma \notin \mathrm{dom}(\Sigma) \\
\langle E[\sigma], \Sigma\rangle & \longmapsto_{\mathsf{cs}} & \langle E[V], \Sigma\rangle \quad [\mathsf{cs1}] \\
 & & \quad \text{where } V = \Sigma(\sigma) \\
\langle E[(:=\ \sigma\ V)], \Sigma[\sigma \leftarrow V']\rangle & \longmapsto_{\mathsf{cs}} & \langle E[\mathtt{void}], \Sigma[\sigma \leftarrow V]]\rangle \quad [\mathsf{cs2}] \\
\langle E[(o^n\ V_1 \ldots\ V_n)], \Sigma\rangle & \longmapsto_{\mathsf{cs}} & \langle E[V], \Sigma\rangle \\
 & & \quad \text{where } V = \delta(o^n, V_1, \ldots V_n) \quad [\mathsf{cs}\delta]
\end{array}
$$

$$
eval_{\mathsf{cs}}(M) = \begin{cases} b & \text{if } \langle M, \emptyset\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} \langle b, \Sigma\rangle \\ \mathtt{function} & \text{if } \langle M, \emptyset\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} \langle \lambda X.N, \Sigma\rangle \\ \mathtt{void} & \text{if } \langle M, \emptyset\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} \langle \mathtt{void}, \Sigma\rangle \end{cases}
$$

To make examples easier to read, we introduce two macros:

$$
\begin{array}{lll}
(\mathsf{let}\ X = M\ \mathsf{in}\ N) & \doteq & ((\lambda X.N)\ M) \\
(\mathsf{begin}\ M\ N) & \doteq & (\mathsf{let}\ X = M\ \mathsf{in}\ N) \\
 & & \quad \text{where } X \notin FV(N)
\end{array}
$$

With these abbreviations, we can write the original example as

$$(\mathsf{let}\ x = \ulcorner 12 \urcorner\ \mathsf{in}\ (\mathsf{begin}\ (:=\ x\ \ulcorner 13 \urcorner)\ x))$$

Here's another example:

$$
\begin{array}{l}
\langle\ \mathsf{let}\ i = \ulcorner 0 \urcorner \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \{\}\rangle \\
\quad \mathsf{in\ let}\ f = \mathsf{Y_V}\lambda f.\lambda x. \\
\qquad\qquad\qquad \mathsf{if0}\ x \\
\qquad\qquad\qquad\quad i \\
\qquad\qquad\qquad\quad (\mathsf{begin} \\
\qquad\qquad\qquad\quad\ (:=\ i\ (+\ i\ x)) \\
\qquad\qquad\qquad\quad\ (f\ (-\ x\ \ulcorner 1 \urcorner))) \\
\qquad\quad \mathsf{in} \\
\qquad\qquad (f\ \ulcorner 1 \urcorner) \\
\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}}\ \langle\ \mathsf{let}\ f = \mathsf{Y_V}\lambda f.\lambda x. \qquad\qquad\qquad\qquad , \{\sigma_1 = \ulcorner 0 \urcorner\}\rangle \\
\qquad\qquad\qquad\qquad\quad \mathsf{if0}\ x \\
\qquad\qquad\qquad\qquad\qquad \sigma_1 \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{begin} \\
\qquad\qquad\qquad\qquad\qquad\ (:=\ \sigma_1\ (+\ \sigma_1\ x)) \\
\qquad\qquad\qquad\qquad\qquad\ (f\ (-\ x\ \ulcorner 1 \urcorner))) \\
\qquad\qquad\quad \mathsf{in} \\
\qquad\qquad\qquad (f\ \ulcorner 1 \urcorner) \\
\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}}\ \langle (\sigma_2\ \ulcorner 1 \urcorner)\ , \{\sigma_1 = \ulcorner 0 \urcorner, \sigma_2 = F\}\rangle
\end{array}
$$

where F =the value of $\mathsf{Y_V}\lambda f.\lambda x.$
$$\begin{array}{l}\mathsf{if0}\ x\\ \quad \sigma_1\\ \quad (\mathsf{begin}\\ \quad\ (:=\ \sigma_1\ (+\ \sigma_1\ x))\\ \quad\ (f\ (-\ x\ \ulcorner 1\urcorner)))\end{array}$$

$$\begin{array}{lll}\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} & \langle\ \mathsf{if0}\ \sigma_4 & ,\{\sigma_1=\ulcorner 0\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner\}\rangle\\ & \quad \sigma_1 & \\ & \quad (\mathsf{begin} & \\ & \quad\ (:=\ \sigma_1\ (+\ \sigma_1\ \sigma_4)) & \\ & \quad\ (\sigma_3\ (-\ \sigma_4\ \ulcorner 1\urcorner))) & \end{array}$$

$$\begin{array}{lll}\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} & \langle\ (\mathsf{begin} & ,\{\sigma_1=\ulcorner 0\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner\}\rangle\\ & \quad (:=\ \sigma_1\ (+\ \sigma_1\ \sigma_4)) & \\ & \quad (\sigma_3\ (-\ \sigma_4\ \ulcorner 1\urcorner))) & \end{array}$$

$$\begin{array}{lll}\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} & \langle\ (\mathsf{begin} & ,\{\sigma_1=\ulcorner 0\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner\}\rangle\\ & \quad (:=\ \sigma_1\ (+\ \ulcorner 0\urcorner\ \ulcorner 1\urcorner)) & \\ & \quad (\sigma_3\ (-\ \sigma_4\ \ulcorner 1\urcorner))) & \end{array}$$

$$\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}}\ \langle\ (\sigma_3\ (-\ \sigma_4\ \ulcorner 1\urcorner))\ ,\{\sigma_1=\ulcorner 1\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner\}\rangle$$

$$\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}}\ \langle\ (\sigma_3\ \ulcorner 0\urcorner)\ ,\{\sigma_1=\ulcorner 1\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner\}\rangle$$

$$\begin{array}{lll}\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} & \langle\ \mathsf{if0}\ \sigma_6 & ,\ \{\sigma_1=\ulcorner 1\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner,\\ & \quad \sigma_1 & \quad \sigma_5=F,\sigma_6=\ulcorner 0\urcorner\}\ \rangle\\ & \quad (\mathsf{begin} & \\ & \quad\ (:=\ \sigma_1\ (+\ \sigma_1\ \sigma_6)) & \\ & \quad\ (\sigma_5\ (-\ \sigma_6\ \ulcorner 1\urcorner))) & \end{array}$$

$$\begin{array}{ll}\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} & \langle\ \sigma_1,\ \{\sigma_1=\ulcorner 1\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner,\\ & \quad \sigma_5=F,\sigma_6=\ulcorner 0\urcorner\}\ \rangle\end{array}$$

$$\begin{array}{ll}\longmapsto\!\!\!\!\rightarrow_{\mathsf{cs}} & \langle\ \ulcorner 1\urcorner,\ \{\sigma_1=\ulcorner 1\urcorner,\sigma_2=F,\sigma_3=F,\sigma_4=\ulcorner 1\urcorner,\\ & \quad \sigma_5=F,\sigma_6=\ulcorner 0\urcorner\}\ \rangle\end{array}$$

10.2 Garbage Collection

In the preceding example, the evaluator must allocate two new slots in the store whenever F is called: one for the f argument and one for x. (Actually, more slots; the trace above ignores the extra arguments introduced by $\mathsf{Y_V}$.) But after F makes a recursive call, neither of the slots is ever used again. For example, σ_2 is never used in the second half of the evaluation. At some point, we could have easily predicted that σ_2 will never be used again; inspecting the expression and store half-way through the evaluation, we could see that σ_2 is not reachable from the expression.

In contrast, σ_1 is used all the way to the end of the evaluation. At the point where the expression to evaluate is $(\sigma_3\ 0)$, σ_1 does not appear in the expression. However, σ_3 contains the expression F, and F includes a reference to σ_1. Thus, even at the expression $(\sigma_3\ 0)$, we can see that σ_1 might be used again.

Since we might want to run the example program for an input much larger than 1, an evaluator for our language must somehow forget about useless slots. Otherwise, a program that we expect to run in constant space will keep accumulating slots on each function call.

Forgetting unused slots is the essence of **garbage collection** in languages like Java and Scheme. We can formalize the notion of garbage collection within our language by defining the

live slots $\mathcal{LS}$ of an expression and state as follows:

$$\begin{array}{lcl}
\mathcal{LS}(\sigma) & = & \{\sigma\} \\
\mathcal{LS}(b) & = & \emptyset \\
\mathcal{LS}(X) & = & \emptyset \\
\mathcal{LS}(\texttt{void}) & = & \emptyset \\
\mathcal{LS}(\,(\lambda X.M)\,) & = & \mathcal{LS}(M) \\
\mathcal{LS}(\,(M_1\ M_2)\,) & = & \mathcal{LS}(M_1) \cup \mathcal{LS}(M_2) \\
\mathcal{LS}(\,(o^n\ M_1 \ldots\ M_n)\,) & = & \mathcal{LS}(M_1) \cup \ldots \mathcal{LS}(M_n) \\
\mathcal{LS}(\,(:=\ X\ M)\,) & = & \mathcal{LS}(M) \\
 & & \\
\mathcal{LS}(\langle M, \{\langle \sigma_1, V_1\rangle, \ldots \langle \sigma_n, V_n\rangle\}\rangle) & = & \mathcal{LS}(M) \cup \mathcal{LS}(V_1) \cup \ldots \mathcal{LS}(V_n)
\end{array}$$

Any σ that is in $\mathrm{dom}(\Sigma)$ but not in $\mathcal{LS}(\langle M, \Sigma\rangle)$ is a **garbage slot**. We can add a rule to $\longmapsto_{\mathsf{cs}}$ that removes a set of garbage slots:

$$\begin{array}{lcll}
\langle M, \Sigma[\sigma_1 \leftarrow V_1] \ldots [\sigma_n \leftarrow V_n]\rangle & \longmapsto_{\mathsf{cs}} & \langle M, \Sigma\rangle & [\mathsf{csgc}] \\
\quad \text{if } \sigma_1, \ldots \sigma_n \notin \mathcal{LS}(\langle M, \Sigma\rangle) & & &
\end{array}$$

Adding this rule to $\longmapsto_{\mathsf{cs}}$ makes it non-deterministic in two ways. First, the rule can apply at the same time as other rules, which means that the programmer does not know when a garbage collection will occur. Second, the rule could match with different sets of σs, which means that a programmer does not know which slots are freed when a collection does occur.

More importantly, if we ignore memory constraints, an evaluation of M with as many garbage collection steps as possible will always produce the same result as an evalution of M without any garbage collection steps. In other words, the garbage collector *proves* that certain slots will never contribute to evaluation, so that it is safe to remove them from the store.

If we consider memory constraints, then we need to ensure that the garbage collection rule is used often enough, and that it collects enough garbage. One possibility is to change the machine so that a garbage collection occurs with every reduction, and that the resulting Σ is the smallest possible one after each step; this strategy is similar to requiring that environments are pruned on every closure creation (as in Section 7.3). As it turns out, a unique minimal Σ exists for each step, so the resulting machine would be deterministic. However, performing a garbage collection after every evaluation step is not a practical implementation. We return to the question of a practical implementation in Section 10.4.

▷ **Exercise 10.1.** Revise the evaluation of

$$\begin{array}{l}
\mathsf{let}\ i = \ulcorner 0 \urcorner \\
\ \mathsf{in}\ \mathsf{let}\ f = \mathsf{Y_V} \lambda f.\lambda x.\ \mathsf{if0}\ x\ i\ (\mathsf{begin}\ (:=\ i\ (+\ i\ x))\ (f\ (-\ x\ \ulcorner 1 \urcorner))) \\
\quad \mathsf{in}\ (f\ \ulcorner 1 \urcorner)
\end{array}$$

in Section 10.1 by inserting a [csgc] step after each of the shown steps. Each [csgc] should produce a minimal store.

▷ **Exercise 10.2.** Suppose that we add a simpler garbage-collection rule to $\longmapsto_{\mathsf{cs}}$ that always collects a single garbage slot:

$$\begin{array}{lcl}
\langle M, \Sigma[\sigma \leftarrow V]\rangle & \longmapsto_{\mathsf{cs}} & \langle M, \Sigma\rangle \\
\quad \text{if } \sigma \notin \mathcal{LS}(\langle M, \Sigma\rangle) & &
\end{array}$$

Why is this rule generally not sufficient to ensure proper garbage collection? (Hint: Why did we ignore reference counting as a possible strategy for memory management?) Provide an expression with the following properties:

1. It loops forever using $\longmapsto_{\mathsf{cs}}$ rules other than [csgc].
2. It loops in bounded space when the [csgc] rule is used occasionally (i.e., never twice in consecutive steps).
3. It cannot loop in bounded space using the simpler rule above (even if the rule is applied multiple times in consecutive steps).

Hint: no plain ISWIM expression meets all three conditions. We introduce garbage collection in this chapter because we did not need it before.

10.3 CEKS Machine

The CS machine contains none of our earlier refinements over the basic ISWIM textual machine. For every step, it has to find the redex, sometimes perform substitutions over large expressions, etc. We can get all of our old refinements back by starting with the CEK machine and extending it with state.

Environments for the **CEKS machine** map variables to slot addresses. Also, since the new := expression form contains a subexpression, we must extend the continuation grammar $\overline{\kappa}$. Finally, a store in the CEKS machine maps slot addresses to value closures (instead of plain values).

$\overline{\mathcal{E}}$	=	a function $\{\langle X, \sigma\rangle, \ldots\}$	
$\overline{\kappa}$	=	$\ldots$	same as CEK
	|	$\langle \mathtt{set}, c, \overline{\kappa}\rangle$	
$\overline{\Sigma}$	=	a function $\{\langle \sigma, v\rangle, \ldots\}$	

Most of the reduction rules for CEKS are the same as the CEK rules, except that they propagate a store. Function application and variable look are changed, however. In the follow definition, the rules with an asterisk are new or significantly revised compared to the CEK rules:

$\langle\langle (M\ N), \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle M, \overline{\mathcal{E}}\rangle, \langle \mathtt{arg}, \langle N, \overline{\mathcal{E}}\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle$ [ceks1]

$\langle\langle (o^n\ M\ N \ldots), \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle M, \overline{\mathcal{E}}\rangle, \langle \mathtt{opd}, \langle o^n\rangle, \langle\langle N, \overline{\mathcal{E}}\rangle, \ldots\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle$ [ceks2]

$\langle\langle V, \overline{\mathcal{E}}\rangle, \langle \mathtt{fun}, \langle (\lambda X.M), \overline{\mathcal{E}}'\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle M, \overline{\mathcal{E}}'[X \leftarrow \sigma]\rangle, \overline{\kappa}, \overline{\Sigma}[\sigma \leftarrow \langle V, \overline{\mathcal{E}}\rangle]\rangle$ [ceks3]*
if $V \notin X$ and $\sigma \notin \mathrm{dom}(\overline{\Sigma})$

$\langle\langle V, \overline{\mathcal{E}}\rangle, \langle \mathtt{arg}, \langle N, \overline{\mathcal{E}}'\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle N, \overline{\mathcal{E}}'\rangle, \langle \mathtt{fun}, \langle V, \overline{\mathcal{E}}\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle$ [ceks4]
if $V \notin X$

$\langle\langle b, \overline{\mathcal{E}}\rangle, \langle \mathtt{opd}, \langle\langle b_i, \overline{\mathcal{E}}_i\rangle \ldots \langle b_1, \overline{\mathcal{E}}_1\rangle,\ o^n\rangle, \langle\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle V, \emptyset\rangle, \overline{\kappa}, \overline{\Sigma}\rangle$ [ceks5]
where $\delta(o^n, b_1, \ldots, b_i, b) = V$

$\langle\langle V, \overline{\mathcal{E}}\rangle, \langle \mathtt{opd}, \langle c', \ldots o^n\rangle, \langle\langle N, \overline{\mathcal{E}}'\rangle, c, \ldots\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle N, \overline{\mathcal{E}}'\rangle, \langle \mathtt{opd}, \langle\langle V, \overline{\mathcal{E}}\rangle, c', \ldots o^n\rangle, \langle c, \ldots\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle$ [ceks6]
if $V \notin X$

$\langle\langle X, \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle c, \overline{\kappa}, \overline{\Sigma}\rangle$ [ceks7]*
where $\overline{\Sigma}(\overline{\mathcal{E}}(X)) = c$

$\langle\langle (:=\ X\ M), \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle M, \overline{\mathcal{E}}\rangle, \langle \mathtt{set}, \langle X, \overline{\mathcal{E}}\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle$ [ceks8]*

$\langle\langle V, \overline{\mathcal{E}}'\rangle, \langle \mathtt{set}, \langle X, \overline{\mathcal{E}}\rangle, \overline{\kappa}\rangle, \overline{\Sigma}\rangle \longmapsto_{\mathsf{ceks}} \langle\langle \mathtt{void}, \emptyset\rangle, \overline{\kappa}, \overline{\Sigma}[\sigma \leftarrow \langle V, \overline{\mathcal{E}}'\rangle]\rangle$ [ceks9]*
if $V \notin X$ where $\overline{\mathcal{E}}(X) = \sigma$

$$eval_{\mathsf{ceks}}(M) = \begin{cases} b & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}, \emptyset\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{ceks}} \langle\langle b, \overline{\mathcal{E}}\rangle, \mathtt{mt}, \overline{\Sigma}\rangle \\ \mathtt{function} & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}, \emptyset\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{ceks}} \langle\langle \lambda X.N, \overline{\mathcal{E}}\rangle, \mathtt{mt}, \overline{\Sigma}\rangle \\ \mathtt{void} & \text{if } \langle\langle M, \emptyset\rangle, \mathtt{mt}, \emptyset\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{ceks}} \langle\langle \mathtt{void}, \overline{\mathcal{E}}\rangle, \mathtt{mt}, \overline{\Sigma}\rangle \end{cases}$$

Note that none of the rules for the CEKS machine insert store addresses into an expression position; they are all in the domain of $\overline{\Sigma}$ and the range of $\overline{\mathcal{E}}$. Thus, if we want to add a garbage-collection rule to our CEKS machine, we need to define the set of live slots based on the machine's current environment. In addition, the continuation may contain environments, so the live slot calculation needs to check the machine's continuation, and value closures in $\overline{\Sigma}$ also contain environments.

$$\langle\langle M, \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}[\sigma_1 \leftarrow \langle V_1, \overline{\mathcal{E}}_1\rangle] \ldots [\sigma_n \leftarrow \langle V_n, \overline{\mathcal{E}}_n\rangle]\rangle \longmapsto_{\mathsf{cs}} \langle\langle M, \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}\rangle \quad [\mathsf{ceksgc}]$$
$$\text{if } \sigma_1, \ldots \sigma_n \notin \mathcal{LS}(\overline{\mathcal{E}}) \cup \mathcal{LS}(\overline{\kappa}) \cup \mathcal{LS}(\overline{\Sigma})$$

$$\begin{array}{lcl} \mathcal{LS}(\overline{\mathcal{E}}) & = & \mathrm{rng}(\overline{\mathcal{E}}) \\ \\ \mathcal{LS}(\mathtt{mt}) & = & \emptyset \\ \mathcal{LS}(\langle \mathtt{fun}, \langle V, \overline{\mathcal{E}}\rangle, \overline{\kappa}\rangle) & = & \mathcal{LS}(\overline{\mathcal{E}}) \cup \mathcal{LS}(\overline{\kappa}) \\ \mathcal{LS}(\langle \mathtt{arg}, \langle N, E\rangle, \overline{\kappa}\rangle) & = & \mathcal{LS}(\overline{\mathcal{E}}) \cup \mathcal{LS}(\overline{\kappa}) \\ \mathcal{LS}(\langle \mathtt{opd}, \langle\langle V_i, \overline{\mathcal{E}}_i\rangle, \ldots o^m\rangle, \langle\langle M_j, \overline{\mathcal{E}}_j\rangle, \ldots\rangle, \overline{\kappa}\rangle) & = & \mathcal{LS}(\overline{\mathcal{E}}_i) \cup \ldots \mathcal{LS}(\overline{\mathcal{E}}_j) \cup \ldots \mathcal{LS}(\overline{\kappa}) \\ \mathcal{LS}(\langle \mathtt{set}, \langle X, \overline{\mathcal{E}}\rangle, \overline{\kappa}\rangle) & = & \mathcal{LS}(\overline{\mathcal{E}}) \cup \mathcal{LS}(\overline{\kappa}) \\ \\ \mathcal{LS}(\{\langle \sigma_1, \langle V_1, \overline{\mathcal{E}}_1\rangle\rangle, \ldots \langle \sigma_n, \langle V_n, \overline{\mathcal{E}}_n\rangle\rangle\}) & = & \mathcal{LS}(\overline{\mathcal{E}}_1) \cup \ldots \mathcal{LS}(\overline{\mathcal{E}}_n) \end{array}$$

10.4 Implementing Garbage Collection

The CEKS machine is nearly a practical implementation of a language with state, but our garbage collection rule is still just a *specification* instead of an *implementation*. In other words, it describes a constraint on garbage collection, but not how to find the garbage slots.

The standard algorithm for finding the minimal set of live slots involves "coloring" the slots "black", "gray", and "white". Black corresponds to a live slot whose value references only black and gray slots. Gray cooresponds to a live slot whose value may reference a white slot. White is a potentially garbage slot; at the end of the following process, all white slots are garbage:

- Color all slots white.
- Color all slots referenced by the current environment gray.
- For every environment $\overline{\mathcal{E}}$ referenced by the current continuation, color all slots referenced by $\overline{\mathcal{E}}$ gray.
- Repeat until there are no gray slots in the store:
  - Choose a gray slot, σ.
  - The value for σ in the current store is some $\langle c, \overline{\mathcal{E}}\rangle$; color all slots referenced by $\overline{\mathcal{E}}$ gray.
  - Color σ black.

We can implement this process as a $\longmapsto_{\mathsf{gc}}$ sub-machine to be used within the CEKS machine. The first part of the machine state corresponds to the set of gray slots, and the second part corresponds to the set of black slots:

$$\begin{array}{lll} \langle\{\sigma_0, \sigma, \ldots\}, & \longmapsto_{\mathsf{gc}} & \langle\{\sigma, \ldots\} \cup ((\mathcal{LS}(V_0) \cup \mathcal{LS}(\overline{\mathcal{E}}_0)) \backslash \{\sigma_0, \sigma', \ldots\}), \\ \{\sigma', \ldots\}, & & \{\sigma_0, \sigma', \ldots\}, \\ \overline{\Sigma}[\sigma_0 \leftarrow \langle V_0, \overline{\mathcal{E}}_0\rangle]\rangle & & \overline{\Sigma}[\sigma_0 \leftarrow \langle V_0, \overline{\mathcal{E}}_0\rangle]\rangle \end{array}$$

With the transitive-reflexive closure $\longmapsto\!\!\!\!\rightarrow_{\mathsf{gc}}$, we can redefine the CEKS GC rule as follows:

$$\langle\langle M, \overline{\mathcal{E}}\rangle, \overline{\kappa}, \overline{\Sigma}\rangle \longmapsto_{\mathsf{cs}} \langle\langle M, \overline{\mathcal{E}}\rangle, \overline{\kappa}, \emptyset[\sigma_1 \leftarrow \overline{\Sigma}(\sigma_1)] \ldots [\sigma_n \leftarrow \overline{\Sigma}(\sigma_n)]\rangle \quad \text{[ceksgc]}$$
$$\text{if } \langle \mathcal{LS}(M) \cup \mathcal{LS}(\overline{\mathcal{E}}), \emptyset, \overline{\Sigma}\rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{gc}} \langle \emptyset, \{\sigma_1, \ldots \sigma_n\}, \overline{\Sigma}\rangle$$

This rule determines a unique and minimal store for a garbage collection step in the CEK machine. At this point, the only missing part of the specification is a way to ensure that the [ceksgc] rule is used frequently enough.

▷ **Exercise 10.3.** Implement the CEKS machine. Every other step taken by your CEKS machine should be a [ceksgc] step that collects as many slots as possible.

10.5 History

The apporach to specifying garbage collection with $\longmapsto_{\mathsf{gc}}$ is due to Morrisett, Felleisen, and Harper [6].

Part III

Models of Typed Languages

Chapter 11: Types

The text string

$$(+ \ulcorner 1 \urcorner \ -)$$

is not a well-formed ISWIM expression, according to the grammar for M. Consequently, an evaluator for ISWIM need not handle such a text string as its input. The "expression" does not get struck, does not signal an error, and does not loop forever; because it's not an *expression* at all, such claims would be nonsense.

In contrast, the text string

$$(+ \ulcorner 1 \urcorner \ (\lambda x.\ulcorner 9 \urcorner))$$

is a well-formed ISWIM expression, according to the grammar for M. Nevertheless, it seems clearly "bad". It's immediately obvious that the expression will get stuck if we try to evaluate it, because a function value cannot be provided as an argument to a primitive operation.

In this and the following chapters, we study expression checkers that can detect and reject "bad" expressions before evaluating them. Since we would like to actually implement the checker, distinguishing good and bad programs should be a mechanical process. Also, we need to explain the checking process to programmers. In particular, we would like take the sort of formal notation that we've developed for explaining program executions, and use it to explain program checking. In short, we would like to develop a formal view of program checking.

For a formal specification of "good" programs, we might try to refine the grammar M to exclude text strings like $(+ \ulcorner 1 \urcorner \ (\lambda x.\ulcorner 9 \urcorner))$. Even if such a M can be defined, however, the required changes would make the grammar especially complex. Extra complexity in M implies corresponding complexity in proofs and programs over M, so directly modifying the grammar is usually not the best approach. Instead, we will define a separate expression-checking relation.

The new relation will still distinguish good and bad expressions in a purely syntactic and static manner, in the same way that a grammar distinguishes text strings that are elements of a set from those that are not. In other words, although the job of the relation is to distinguish expressions that evaluate badly from those that evaluate nicely, the checker cannot execute the program to make the distintiction. Doing so would defeat the purpose of checking expressions before evaluating them.

Recall that every expression M either

- evaluates to a value;
- gets stuck during evaluation; or
- loops forever.

Since we cannot solve the halting problem, we cannot implement a decision procedure to put an expression into exactly one of these categories; the power of our checking relation will be necessarily limited. The checker can still be useful, though:

- The checker might reject expressions that provably get stuck or loop forever. In that case, when an expression passes the type checker, it might or might not produce a value.
- The checker might accept only expressions that provably produce a value. In that case, when an expression passes the type checker, it will certainly produce a value, but a different program that produces a value might be rejected by the checker.

⇒ • The checker might accept only expressions that provably either produce a value or loop forever. In that case, when an expression passes the type checker, it will certainly not get stuck. As in the previous case, a program that produces a value might be rejected by the checker.

A **type relation** (or **type system**) implements the last of the above possibilities. A language that consists of an expression grammar, evaluation relation, and type relation is a **typed language**. Each language that we have studied so far, which comprised only an expression grammar and a reduction relation, was an **untyped language**.

The design of a type system is an open-ended task: since any type system for a realistic language must reject some set of programs that do not get stuck, we can always search for an improved type system that rejects fewer good programs. In this chapter, we begin our exploration of types with a small language and a simple type checker. In the following chapters, we explore a series of type systems for ISWIM.

11.1 Numbers and Booleans

To start exploring type systems, we begin with a small language of booleans and numbers:

$$
\begin{array}{rcl}
N & = & \texttt{t} \\
 & | & \texttt{f} \\
 & | & \ulcorner n \urcorner \\
 & | & (+\ N\ N) \\
 & | & (\texttt{iszero}\ N) \\
 & | & (\texttt{if}\ N\ N\ N)
\end{array}
$$

$$
\begin{array}{lcll}
(+\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & \texttt{z} & \ulcorner n+m \urcorner & \\
(\texttt{iszero}\ \ulcorner 0 \urcorner) & \texttt{z} & \texttt{t} & \\
(\texttt{iszero}\ \ulcorner n \urcorner) & \texttt{z} & \texttt{f} & \text{where} \quad n \neq 0 \\
(\texttt{if}\ \texttt{t}\ N_1\ N_2) & \texttt{z} & N_1 & \\
(\texttt{if}\ \texttt{f}\ N_1\ N_2) & \texttt{z} & N_2 &
\end{array}
$$

$$
\begin{array}{rcl}
C & = & [\,] \\
 & | & (+\ C\ N) \\
 & | & (+\ N\ C) \\
 & | & (\texttt{iszero}\ C) \\
 & | & (\texttt{if}\ C\ N\ N) \\
 & | & (\texttt{if}\ N\ C\ N) \\
 & | & (\texttt{if}\ N\ N\ C)
\end{array}
\qquad
\begin{array}{rcl}
E & = & [\,] \\
 & | & (+\ E\ N) \\
 & | & (+\ V\ E) \\
 & | & (\texttt{iszero}\ E) \\
 & | & (\texttt{if}\ E\ N\ N)
\end{array}
\qquad
\begin{array}{rcl}
V & = & \texttt{t} \\
 & | & \texttt{f} \\
 & | & \ulcorner n \urcorner
\end{array}
$$

If we define $\longmapsto_{\texttt{z}}$, $\longmapsto\!\!\!\!\rightarrow_{\texttt{z}}$ and $eval_{\texttt{z}}$ in the usual way, then this language has many expressions that get stuck. For example:

$$(+\ \texttt{t}\ \ulcorner 0 \urcorner)$$

gets stuck because + doesn't operate on booleans, and

$$(\texttt{if}\ (+\ \ulcorner 1 \urcorner\ \ulcorner 5 \urcorner)\ \texttt{t}\ \ulcorner 0 \urcorner)$$

gets stuck because the test expression for `if` does not produce a boolean.

Since the language for N is not Turing-complete, we could actually define an type checker based on evaluation. In preparation for more interesting languages, however, we will adopt a less precise strategy that scales to more interesting languages.

The key idea is that programs get stuck because the wrong kind of value flows into a position where the reduction rules inspect the value. The + operator expects numbers, `iszero` expects numbers, and `if` expects booleans.

We will therefore design a relation that associates expressons with **types**. Like an expression, a type in the formal sense is a purely syntactic entity. In the same way that we define formal evaluation relations based on an intution of how expressions should be evaluated, we also define

a formal type relation based on our intutions about the kinds of values that appear during evaluation.

For our number and boolean language, we first define the set of types T:

$$\begin{array}{rcl} T & = & \texttt{bool} \\ & | & \texttt{num} \end{array}$$

The relation $\vdash \_ : \_$ matches expressions with types. Literal boolean expressions have the type `bool`, and literal numbers have the type `num`:

$$\begin{array}{c} \vdash \texttt{t} : \texttt{bool} \\ \vdash \texttt{f} : \texttt{bool} \\ \vdash \ulcorner n \urcorner : \texttt{num} \end{array}$$

We might also introduce a rule that assigns every addition expression the type `num`. However, we will use $\vdash \_ : \_$ both to describe the kind of value returned by an expression, and to identify expressions that cannot get stuck. In other words, we want to design $\vdash \_ : \_$ so that if $\vdash N : T$ for some N and T, then N cannot get stuck.

Thus, we want an addition expression to have type `num` only if its subexpressions also have type `num`. The `iszero` rule is similar, except that it gets the type `bool`:

$$\frac{\vdash N_1 : \texttt{num} \quad \vdash N_2 : \texttt{num}}{\vdash (+\ N_1\ N_2) : \texttt{num}}$$

$$\frac{\vdash N : \texttt{num}}{\vdash (\texttt{iszero}\ N) : \texttt{bool}}$$

The only remaining expression form is `if`. Clearly, the first sub-expression of an `if` expression should have type `bool`, but the **z** rule for `if` makes no requirements on the other expressions. This analysis suggests a rule of the shape:

$$\frac{\vdash N_1 : \texttt{bool}}{\vdash (\texttt{if}\ N_1\ N_2\ N_3) : \_}$$

But what is the type for the entire `if` expression? If N_1 evaluates to `t`, the result of the `if` expression is the result of N_2. If N_1 instead evaluates to `f`, the result of the `if` expression is the result of N_3. Since we do not want to evaluate N_1 to type-check the expression, we are forced to require that N_2 and N_3 both have the same type, and that type is the one for the `if` expression:

$$\frac{\vdash N_1 : \texttt{bool} \quad \vdash N_2 : T \quad \vdash N_3 : T}{\vdash (\texttt{if}\ N_1\ N_2\ N_3) : T}$$

This rule can match with T as `bool` or as `num`, but all three instances of T must match the same type.

At this point, we have a complete set of type rules for N and **v**. We can see that (`if (if t f t)` $\ulcorner 0 \urcorner$ $\ulcorner 1 \urcorner$) is categorized as a "good" expression, since $\vdash \_ : \_$ assigns it a type:

$$\frac{\dfrac{\vdash \texttt{t} : \texttt{bool} \quad \vdash \texttt{f} : \texttt{bool} \quad \vdash \texttt{t} : \texttt{bool}}{\vdash (\texttt{if t f t}) : \texttt{bool}} \qquad \vdash \ulcorner 0 \urcorner : \texttt{num} \quad \vdash \ulcorner 1 \urcorner : \texttt{num}}{\vdash (\texttt{if (if t f t)}\ \ulcorner 0 \urcorner\ \ulcorner 1 \urcorner) : \texttt{num}}$$

In contrast, (`if` $\ulcorner 7 \urcorner$ $\ulcorner 0 \urcorner$ $\ulcorner 1 \urcorner$) has no type, since $\not\vdash \ulcorner 7 \urcorner : \texttt{bool}$. The following example also has no type:

(`if t` $\ulcorner 1 \urcorner$ `f`)

despite the fact that it reduces via $\longmapsto_{\mathbf{z}}$ to $\ulcorner 1 \urcorner$. Our type system is thus conservative in the expressions it designates as "good". Whether this conservatism is acceptable depends on a programmer's needs, just like any other aspect of the language.

Lemma 11.1 [Unique Evaluation Context for $\longmapsto_{\mathbf{z}}$]: If $N \notin V$, then there exist E and N' such that $N = E[N']$ and N has one of the following forms:

- $(+\ V_1\ V_2)$
- $(\texttt{iszero}\ V)$
- $(\texttt{if}\ V\ N_1\ N_2)$

Lemma 11.2 [Unique Type for N]: If $\vdash N : T$ and $\vdash N : T'$, then $T = T'$.

Lemma 11.3: If $\vdash C[N] : T$, then $\vdash N : T'$ for some T'.

Lemma 11.4: If $\vdash C[N] : T$, $\vdash N : T'$, and $\vdash N' : T'$, then $\vdash C[N'] : T$.

▷ **Exercise 11.1.** Prove that $\vdash (\texttt{if}\ (\texttt{iszero}\ \ulcorner 3 \urcorner)\ (\texttt{iszero}\ (+\ \ulcorner 1 \urcorner\ \ulcorner 3 \urcorner))\ \texttt{t}) : \texttt{bool}$ (i.e., show the proof tree).

▷ **Exercise 11.2.** Prove Lemma 11.1 (Unique Evaluation Context).

▷ **Exercise 11.3.** Prove Lemma 11.2 (Unique Type).

▷ **Exercise 11.4.** Prove Lemma 11.3.

▷ **Exercise 11.5.** Prove Lemma 11.4.

11.2 Soundness

A typed language is **sound** if its type system and evaluation rules are consistent, i.e., if the type system assigns an expression a type, then the expression does not get stuck when evaluated.

Theorem 11.5 [Soundness of $\vdash \_ : \_$ and $\longmapsto\!\!\!\!\rightarrow_{\mathbf{z}}$]: If $\vdash N : T$, then either

- $N \longmapsto\!\!\!\!\rightarrow_{\mathbf{z}} V$, or
- $N \longmapsto\!\!\!\!\rightarrow_{\mathbf{z}} N'$ implies that $N' \longmapsto_{\mathbf{z}} N''$ for some N''.

To prove the soundness theorem, we develop a pair of lemmas. The first is a **preservation** (or **subject reduction**) lemma that says when $\longmapsto_{\mathbf{z}}$ maps a typed expression to some other expression, the expressions have the same type. The second is a **progress** lemma, which says that $\longmapsto_{\mathbf{z}}$ relates every non-V typed expression to some other expression. Together, these lemmas prove the soundness theorem. This technique for proving soundness was developed by Wright and Felleisen.

Lemma 11.6 [Preservation for $\longmapsto_{\mathbf{z}}$]: If $N \longmapsto_{\mathbf{z}} N'$ and $\vdash N : T$, then $\vdash N' : T$.

Proof for Lemma 11.6: Since, $N \longmapsto_{\mathbf{z}} N'$, there must be some E, N_1, and N_1' such that $N = E[N_1]$ where $N_1\ \mathbf{z}\ N_1'$. By Lemma 11.3 there must be some T_1 such that $\vdash N_1 : T_1$. Consider all cases for $N_1\ \mathbf{z}\ N_1'$, which will prove a sub-claim, $\vdash N_1' : T_1$:

- **Case** $(+\ \ulcorner n \urcorner\ \ulcorner m \urcorner)\ \mathbf{z}\ \ulcorner n + m \urcorner$
  Since $\vdash \ulcorner n \urcorner : \texttt{num}$ and $\vdash \ulcorner m \urcorner : \texttt{num}$, $\vdash (+\ \ulcorner n \urcorner\ \ulcorner m \urcorner) : \texttt{num}$ so $T_1 = \texttt{num}$. Similarly, $\vdash \ulcorner n + m \urcorner : \texttt{num}$, so the sub-claim holds.

- **Case** $(\texttt{iszero}\ \ulcorner 0 \urcorner)\,\mathbf{z}\,\texttt{t}$
  Since $\vdash \ulcorner 0 \urcorner : \texttt{num}$, $T_1 = \texttt{bool}$. Since $\vdash \texttt{t} : \texttt{bool}$, the sub-claim holds.
- **Case** $(\texttt{iszero}\ \ulcorner n \urcorner)\,\mathbf{z}\,\texttt{f}$
  Analogous to the previous case.
- **Case** $(\texttt{if}\ \texttt{t}\ N_2\ N_3)\,\mathbf{z}\,N_2$
  By assumption, $\vdash (\texttt{if}\ \texttt{t}\ N_2\ N_3) : T_1$, so whatever T_1 is, it must be possible to show $\vdash N_2 : T_1$. Thus, the sub-claim holds.
- **Case** $(\texttt{if}\ \texttt{f}\ N_2\ N_3)\,\mathbf{z}\,N_3$
  Analogous to the previous case.

Finally, by Lemma 11.4, $\vdash E[N_1'] : T$, so the original claim holds.

Lemma 11.7 [Progress for $\longmapsto_{\mathbf{z}}$]: If $\vdash N : T$, then either $N \in V$ or $N \longmapsto_{\mathbf{z}} N'$ for some N'.

Proof for Lemma 11.7: By induction on the structure of N.

- Base cases:
  - **Case** $N = \texttt{t}$
    $\texttt{t} \in V$.
  - **Case** $N = \texttt{f}$
    $\texttt{t} \in V$.
  - **Case** $N = \ulcorner n \urcorner$
    $\ulcorner n \urcorner \in V$.
- Inductive cases:
  - **Case** $N = (+\ N_1\ N_2)$
    There are three sub-cases:
    - **Case** $N_1 \notin V$
      By Lemma 11.1, $N_1 = E[N_0]$ for some E and $N_0 \notin V$. By assumption, $\vdash (+\ N_1\ N_2) : T$, which requires $\vdash N_0 : T_0$ for some T_0. By induction, $N_0 \longmapsto_{\mathbf{z}} N_0'$; in particular, $N_0\,\mathbf{z}\,N_0'$ due to the partitioning of N_1 into E and N_0. Let $E' = (+\ E\ N_2)$; then $N = E'[N_0]$ and $E'[N_0] \longmapsto_{\mathbf{z}} E'[N_0']$.
    - **Case** $N_2 \notin V$
      Analogous to the previous case.
    - **Case** $N_1 = V_1$**,** $N_2 = V_2$
      Since $\vdash (+\ N_1\ N_2) : T$, $\vdash N_1 : \texttt{num}$ and $\vdash N_2 : \texttt{num}$. By inspection of V, we see that the only members of V that have type $\texttt{num}$ are numbers: $\ulcorner n \urcorner$. Thus, $N = (+\ \ulcorner n_1 \urcorner\ \ulcorner n_2 \urcorner)$. By the definition of $\longmapsto_{\mathbf{z}}$, $N \longmapsto_{\mathbf{z}} \ulcorner n_1 + n_2 \urcorner$.
  - **Case** $N = (\texttt{iszero}\ N_1)$
    Analogous to the previous case.
  - **Case** $N = (\texttt{if}\ N_1\ N_2\ N_3)$
    There are two sub-cases:
    - **Case** $N_1 \notin V$
      This case is analogous to the first sub-case of $N = (+\ N_1\ N_2)$.
    - **Case** $N_1 = V_1$
      Since $\vdash (\texttt{if}\ N_1\ N_2\ N_3) : T$, $\vdash N_1 : \texttt{bool}$. By inspection of V, we see that the only members of V that have type $\texttt{bool}$ are $\texttt{t}$ and $\texttt{f}$. If $N_1 = \texttt{t}$, then $N \longmapsto_{\mathbf{z}} N_2$, and if $N_1 = \texttt{f}$ then $N \longmapsto_{\mathbf{z}} N_3$.

▷ **Exercise 11.6.** Prove Theorem 11.5 (Soundness). Begin by proving a multi-step presevation lemma: If $N \longmapsto\!\!\!\!\rightarrow_{\mathbf{z}} N'$ and $\vdash N : T$, then $\vdash N' : T$.

Chapter 12: Simply Typed ISWIM

We are now ready to consider a typed variant of ISWIM that can reject expressions such as

$$(+ \ulcorner 1 \urcorner \ (\lambda x.\ulcorner 9 \urcorner))$$

Our simple language from the previous chapter had exactly two types: `bool` and `num`. Since ISWIM is parameterized over the basic constrants and primitive operations, a typed ISWIM will also be parameterized over the basic types. But, as the expression above illustrates, a typed ISWIM will always need at least one type: the type for functions.

In this chapter, we present **Simply Typed ISWIM**, which illustrates a simple type system for functions.

12.1 Function Types

For the sake of concrete examples, we will continue to use the same basic constants and primitive operations that we used in previous chapters. Thus, the only basic type we need is `num`.

We might then define a type system for Simply Typed ISWIM with the following elements:

$$\begin{array}{rcl} T & = & \texttt{num} \\ & | & \texttt{function} \end{array}$$

Such a type system could reject

$$(+ \ulcorner 1 \urcorner \ (\lambda x.\ulcorner 9 \urcorner))$$

because + will require arguments of type `num`, and $(\lambda x.\ulcorner 9 \urcorner)$ will have type `function`. But consider the expression

$$(+ \ulcorner 1 \urcorner \ (\,(\mathsf{if0}\ M\ (\lambda x.\ulcorner 8 \urcorner)\ \lambda xy.y\,)\ \ulcorner 7 \urcorner))$$

Both arms of the if0 expression are of type `function`, so the application expression that uses the result of the if0 expression could be well-typed. However, the result of the application to $\ulcorner 7 \urcorner$ will either be a number or the identity function, depending on whether M produces $\ulcorner 0 \urcorner$ or not. Thus, the addition expression will potentially get stuck.

The problem is that the type `function` does not encode enough information about the structure of a function. Some functions expect numbers, and some expect other functions. Some functions produce numbers, and some produce other functions. The type for a function needs to expose this information.

The solution is to support types of the form $(\_ \to \_)$. The left-hand side of the arrow specifies the type of value that a function consumes, and the right-hand side specified the type of value that a function returns. For example, if a function takes a number and produces a number, its type is $(\texttt{num} \to \texttt{num})$. If a function takes a number and produces another function from numbers to numbers, its type is $(\texttt{num} \to (\texttt{num} \to \texttt{num}))$.

The general type grammar for Simply Typed ISWIM is as follows:

$$\begin{array}{rcl} T & = & B \\ & | & (T \to T) \\ B & = & \text{a basic type} \end{array}$$

The set B depends on the basic constants b. For our running example b, we define B to contain only a token for number types:

$$B \ = \ \texttt{num}$$

Before defining type rules, we must address one more aspect of Simply Typed ISWIM expressions. Consider the function

$$\lambda x.x$$

Its type might be ($\texttt{num} \to \texttt{num}$) if we want to use it on numbers, but the function works just as well on any kind of value, including other functions. Without more information about the kind of value that x is supposed to be, the type rules will have to guess. To avoid this problem (for now), we modify the expression grammar of Simply Typed ISWIM so that the formal argument of a function is annotated with an explicit type, indicating the kind of value that the function expects to consume. Thus, the above expression must be written as

$$\lambda x{:}\texttt{num}.x$$

for the identity function on numbers, or

$$\lambda x{:}(\texttt{num} \to \texttt{num}).x$$

for the identify function on functions from numbers to numbers.

$$\begin{array}{rcl} M & = & X \\ & | & (\lambda X{:}T.M) \\ & | & (M\ M) \\ & | & b \\ & | & (o^n\ M\ \ldots\ M) \end{array}$$

12.2 Type Rules for Simply Typed ISWIM

In the same way that we parameterize Simply Typed ISWIM over its basic constants and their types, we assume an external typing function $\mathcal{B}$ for basic constants, and an external function Δ that supplies the expected and result types of primitive operations:

$$\vdash b : B \quad \text{if} \quad \mathcal{B}(b) = B$$

$$\frac{\vdash M_1 : B_1\ \ldots \qquad \vdash M_n : B_n}{\vdash (o^n\ M_1 \ldots\ M_n) : T} \quad \text{where} \quad \Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, T\rangle$$

For our basic constants and primitive operations, $\mathcal{B}$ maps all basic constants to `num`, and Δ requires `num` arguments for all operations.

$$\begin{array}{lcl} \mathcal{B}(b) & = & \texttt{num} \\ \\ \Delta(\texttt{add1}) & = & \langle\langle\texttt{num}\rangle, \texttt{num}\rangle \\ \Delta(\texttt{sub1}) & = & \langle\langle\texttt{num}\rangle, \texttt{num}\rangle \\ \Delta(\texttt{iszero}) & = & \langle\langle\texttt{num}\rangle, (\texttt{num} \to (\texttt{num} \to \texttt{num}))\rangle \\ \\ \Delta(+) & = & \langle\langle\texttt{num}, \texttt{num}\rangle, \texttt{num}\rangle \\ \Delta(-) & = & \langle\langle\texttt{num}, \texttt{num}\rangle, \texttt{num}\rangle \\ \Delta(*) & = & \langle\langle\texttt{num}, \texttt{num}\rangle, \texttt{num}\rangle \\ \Delta(\uparrow) & = & \langle\langle\texttt{num}, \texttt{num}\rangle, \texttt{num}\rangle \end{array}$$

Now that we have rules for basic constants and primitive operations, we need rules for expressions of the form X, $(\lambda X.M)$, and $(M\ M)$.

What type should a variable X have? For evaluation, we never needed to consider the *value* of a variable, because the reduction rules caused (bound) variables to be replaced with

values before the variable itself was evaluated. In the CEK machine, we saw that environments provided an equivalent notion of evaluation without rewriting the body of a function.

For the type checker, we will use an environment to handle variables. This choice emphasizes that type checking is static (i.e., the type checker only inspects an expression, and does not evaluate it). For type checking, instead of mapping variables to values, a environment Γ maps variables to types. We implement the enviornment as a sequence of type associations:

$$\begin{array}{rcll} \Gamma & = & \epsilon & \\ & | & \Gamma, X{:}T & \\ & & & \\ \Gamma(X) & = & T & \text{if} \quad \Gamma = \Gamma', X{:}T \\ \Gamma(X) & = & \Gamma'(X) & \text{if} \quad \Gamma = \Gamma', X'{:}T \text{ and } X \neq X' \end{array}$$

The type relation for Simply Typed ISWIM maps pairs Γ and M to a type T, written as $\Gamma \vdash M : T$. For many rules, like the ones for basic constants and primitive operations, the environment is simply passed along to type subexpressions, but the environment is consulted to assign a type to a variable.

$$\Gamma \vdash X : T \quad \text{if} \quad \Gamma(X) = T$$
$$\Gamma \vdash b : B \quad \text{if} \quad \mathcal{B}(b) = B$$

$$\frac{\Gamma \vdash M_1 : B_1 \ \ldots \quad \Gamma \vdash M_n : B_n}{\Gamma \vdash (o^n \ M_1 \ldots \ M_n) : T} \quad \text{where} \quad \Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, T\rangle$$

The type environment must be extended to assign a type to the body of a λ expression. For example, the expression

$$\lambda x{:}\texttt{num}.(+ \ x \ \ulcorner 3 \urcorner)$$

is a function, so it should have a type of the form $(T \to T)$. The type for the left-hand side of the arrow is provided by the programmer: `num`. To get the type for the right-hand side, we must check the body expression $(+ \ x \ \ulcorner 3 \urcorner)$ in an environment where x is mapped to `num`.

More generally, the rule for checking a function expression is as follows:

$$\frac{\Gamma, X{:}T \vdash M : T'}{\Gamma \vdash (\lambda x{:}T.M) : (T \to T')}$$

Finally, the type of an application expression $(M \ N)$ depends on the kind of value that the function M returns. In particular, since M must produce a function, it must have an arrow type, and the right-hand side of the arrow type determines the result type of the application. Furthermore, the type of N must match the type of argument value expected by M, which is the left-hand side of the arrow type.

$$\frac{\Gamma \vdash M : (T' \to T) \quad \Gamma \vdash N : T'}{\Gamma \vdash (M \ N) : T}$$

The following example illustrates typing with functions, using `n` as an abbreviation for `num`:

$$\dfrac{\dfrac{\dfrac{f{:}(\texttt{n} \to \texttt{n}) \vdash f : (\texttt{n} \to \texttt{n}) \quad f{:}(\texttt{n} \to \texttt{n}) \vdash \ulcorner 0 \urcorner : \texttt{n}}{f{:}(\texttt{n} \to \texttt{n}) \vdash (f \ \ulcorner 0 \urcorner) : \texttt{num}}}{\vdash (\lambda f{:}(\texttt{n} \to \texttt{n}).(f \ \ulcorner 0 \urcorner)) : ((\texttt{n} \to \texttt{n}) \to \texttt{n})} \quad \dfrac{\dfrac{y{:}\texttt{num} \vdash y : \texttt{num}}{y{:}\texttt{num} \vdash (\texttt{add1} \ y) : \texttt{num}}}{\vdash (\lambda y{:}\texttt{n}.(\texttt{add1} \ y)) : (\texttt{n} \to \texttt{n})}}{\vdash ((\lambda f{:}(\texttt{n} \to \texttt{n}).(f \ \ulcorner 0 \urcorner)) \ (\lambda y{:}\texttt{n}.(\texttt{add1} \ y))) : \texttt{n}}$$

Note that "Γ" never appears in the above proof tree, in the same way that "M" and "T" never appear. Instead, just as "M" in the rules is replaced by a specific expression, "Γ" is replaced by a specific type environment.

▷ **Exercise 12.1.** Replace T_1, T_2, and T_3 with specific types in

$$\vdash (\lambda x{:}T_1.\lambda y{:}T_2.(y\ x))\ \ulcorner 10 \urcorner\ (\lambda z{:}\texttt{num}.(\texttt{iszero}\ z)) : T_3$$

and prove it (i.e., show the proof tree).

12.3 Soundness

We can prove that our type rules for Simply Typed ISWIM are sound with respect to $\longmapsto_{\mathbf{v}}$ as long as δ is consistent with $\mathcal{B}$ and Δ. In other words, we require the following assumption for any δ, $\mathcal{B}$, and Δ.

Assumption 1 (soundness of primitives):

$\delta(o^n, b_1, ...b_n) = V$	implies	$\Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, T\rangle$, $\mathcal{B}(b_1) = B_1, \ldots\ \mathcal{B}(b_n) = B_n$, and $\vdash V : T$
and		
$\Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, T\rangle$, and $\mathcal{B}(b_1) = B_1, \ldots\ \mathcal{B}(b_n) = B_n$	implies	$\exists V\ \delta(o^n, b_1, ...b_n) = V$

With this assumption, we can prove the soundness of Simply Typed ISWIM using the same technique as in the previous section: we define and prove Preservation and Progress lemmas.

Theorem 12.1 [Soundness of Simply Typed ISWIM]: If $\vdash M : T$, then either

- $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$, or
- $M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} M'$ implies that $M' \longmapsto_{\mathbf{v}} M''$ for some M''.

Lemma 12.2 [Preservation for Simply Typed ISWIM]: If $M \longmapsto_{\mathbf{v}} N$ and $\vdash M : T$, then $\vdash N : T$.

Lemma 12.3 [Progress for Simply Typed ISWIM]: If $\vdash M : T$, then either $M \in V$ or $M \longmapsto_{\mathbf{v}} N$ for some N.

As before, the above lemmas rely on a set of minor lemmas about unique types and types in context:

Lemma 12.4: If $\vdash M : T$ and $\vdash M : T'$, then $T = T'$.

Lemma 12.5: A proof tree for $\vdash C[M] : T$ contains a proof tree for $\Gamma \vdash M : T'$ for some Γ and T'.

Lemma 12.6: If a proof tree for $\vdash C[M] : T$ contains a proof tree for $\Gamma \vdash M : T'$, then $\Gamma \vdash N : T'$ implies $\vdash C[N] : T$.

In addition, because the type rules work with type environments while the evalution rules use substitution, we need a lemma that relates the types of an expression before and after substitution:

Lemma 12.7: If $\Gamma, X{:}T' \vdash M : T$ and $\vdash V : T'$, then $\Gamma \vdash M[X \leftarrow V] : T$.

▷ **Exercise 12.2.** Prove Lemma 12.2 (Preservation), assuming the other lemmas above.

▷ **Exercise 12.3.** Prove Lemma 12.3 (Progress), assuming the other lemmas above.

12.4 Normalization

In the number and boolean language of the previous chapter, the evaluation of any typed expression terminates. This fact is fairly obvious, since *every* expression in the language either gets stuck or terminates; there is no way to write a loop.

We have seen many looping ISWIM expressions, such as Ω. We might ask, then, whether any Simply Typed ISWIM expression that has a type can loop. The answer, perhaps surprisingly, is "no" (assuming well-behaved primitive operations, as explaiend below). Every typed expression in Simply Typed ISWIM terminates!

A corollary of the above claim is that Ω has no well-typed representation in Simply Typed ISWIM. To see why, recall that Ω is

$$(\lambda x.x\ x)\ (\lambda x.x\ x)$$

To make Ω a Simply Typed ISWIM expression, we need an explicit type T for the argument x:

$$(\lambda x{:}T.x\ x)\ (\lambda x{:}T'.x\ x)$$

If there is such a T, then it must be an arrow type $(T_1 \to T_2)$, since the argument x is applied as a function. Furthermore, since x is applied to itself, the left-hand side of the arrow type, T_1, must be the same as the type for x, i.e., $T_1 = (T_1 \to T_2)$. But now we have a problem, one that is reminiscent of our attempt to define the recursive function mult as an abbreviation: we cannot expand T_1 to a type expression that does not contain T_1.

In short, we have no way to write a type for x. The Y-combinator trick that we used for mult not work for T_1, because we have nothing equivalent to λ in the type language (i.e., we cannot abstract over type expressions).

We have allowed primitive operators to generate arbitrary values, including λ expressions, so a primitive operator might circumvent Ω's problem by generating a closed expression that contains the original operator. We could ensure that primitive operators do not "misbehave" in this way by introducing one more assumption about the primitives:

> **Assumption 2:**
> $\delta(o^n, b_1, \ldots b_n) = (\lambda X.M)$ implies M contains no primitive operations

Even ignoring the complications of this assumption, simple induction does not suffice to prove that all typed expressions terminate. Roughly, the problem is that a β_v reduction generates an expression that is not a sub-structure of the original expression (due to variable substitutions), so the obvious inductive hypothesis does not apply.

Instead, the proof requires an auxilliary relation, $\mathcal{V}$:

$\mathcal{V}(M, B)$	if	$\vdash M : B$
	and	$M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$ for some V
$\mathcal{V}(M, (T_1 \to T_2))$	if	$\vdash M : (T_1 \to T_2)$
	and	$M \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}} V$ for some V
	and	$\mathcal{V}(N, T_1)$ implies $\mathcal{V}(\,(M\ N), T_2)$

The $\mathcal{V}(\_, \_)$ relation captures the notion that an expression terminates, and that it can be put into any context and not affect the termination of the context. We can easily prove that a single evaluation step preserves this "deep termination" property of an expression:

Lemma 12.8: If $M \longmapsto_{\mathbf{v}} N$, then $\mathcal{V}(M, T)$ if and only if $\mathcal{V}(N, T)$.

Proof for Lemma 12.8: By induction on the structure of T.

- Base case:
  - **Case** $T = B$
    If either M or N terminates, then clearly both terminate with the same V. Furthermore, M and N have the same type by Presevation, so the claim holds.
- Inductive case:
  - **Case** $T = (T_1 \to T_2)$ **for some** T_1 **and** T_2
    Assume that $\mathcal{V}(M, T)$, so that $M \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{v}} V$ for some V, and $\mathcal{V}(N', T_1)$ for any N' implies $\mathcal{V}(\,(M\ N'), T_2)$. Clearly $N \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{v}} V$. Since $M \longmapsto_{\mathbf{v}} N$, we have $(M\ N') \longmapsto_{\mathbf{v}} (N\ N')$. Thus, by induction, $\mathcal{V}(\,(N\ N'), T_2)$.
    If we assume $\mathcal{V}(N, T)$, we can similarly conclude $\mathcal{V}(M, T)$.

We are now ready to prove the main lemma. The primitive-operation rule can break the induction we'd like to use, so at first we leave it out. We then prove a similar lemma that supports primitive operations, as long as they satisfy Assumptions 1 and 2.

Lemma 12.9: If $\Gamma = X_1{:}T_1, \ldots X_n{:}T_n$ and the following hold:

- M contains no primitive applications,
- $\Gamma \vdash M : T$, and
- $\mathcal{V}(V_1, T_1)$, $\ldots$ $\mathcal{V}(V_n, T_n)$ for some V_1, $\ldots$, V_n,

then $\mathcal{V}(M[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], T)$.

Proof for Lemma 12.9: By induction on $\Gamma \vdash M : T$.

- Base cases:
  - **Case** $\Gamma \vdash X : T$
    X must be some X_i for $i \in [1, n]$. Thus, $X[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n] = V_i$, and we have assumed $\mathcal{V}(V_i, T)$, so the claim holds.
  - **Case** $\Gamma \vdash b : T$
    $T \in B$, $b[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n] = b$, and $b \in V$ clearly terminates, so the claim holds.
- Inductive cases:
  - **Case** $\Gamma \vdash (\lambda X{:}T_1.N) : T$
    T must be of the form $(T_1 \to T_2)$, where $\Gamma, X{:}T_1 \vdash N : T_2$. Clearly $M' = M[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n]$ terminates, since M is a value. Choose an arbitrary N' such that $\mathcal{V}(N', T_1)$. To prove the claim, we must show that $\mathcal{V}(\,(M'\ N'), T_2)$.
    By the definition of $\mathcal{V}$, $N' \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{v}} V$ for some V; since M' is a value, we can see that $(M'\ N') \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{v}} (M'\ V)$. By β_{v}, this expression reduces further to

    $$N[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n][X \leftarrow V]$$

    That N' reduces to V also implies $\mathcal{V}(V, T_1)$ by a simple multi-step extension of Lemma 12.8. Combined with $\Gamma, X{:}T_1 \vdash N : T_2$ and the induction hypothesis, we can conclude

    $$\mathcal{V}(N[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n][X \leftarrow V], T_2)$$

    Again by a multi-step-extended Lemma 12.8, we can conclude $\mathcal{V}(\,(M'\ N'), T_2)$, as desired.

– **Case** $\Gamma \vdash (M_1\ M_2) : T$
By the type rule for application and by induction,

$$\mathcal{V}(M_1[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], (T_1 \rightarrow T))$$

and

$$\mathcal{V}(M_2[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], T_1)$$

for some T_1. Since M_1 has an arrow type, by the definition of $\mathcal{V}$,

$$\mathcal{V}(\,(M_1[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n]\ M_2[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n]), T)$$

We can lift the substitutions out to get the equivalent

$$\mathcal{V}(\,(M_1\ M_2)[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], T)$$

which is what we wanted.

– **Case** $\Gamma \vdash (o^m\ M_1 \ldots\ M_m) : T$
By assumption, this case cannot occur.

Lemma 12.10: If $\Gamma = X_1{:}T_1, \ldots X_n{:}T_n$ and the following hold:

- $\Gamma \vdash M : T$, and
- $\mathcal{V}(V_1, T_1)$, ... $\mathcal{V}(V_n, T_n)$ for some V_1, ..., V_n,

then $\mathcal{V}(M[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], T)$.

Proof for Lemma 12.10: The proof is analogous to the proof for Lemma 12.9, except that the primitive operation case must be addressed.

- **Case** $\Gamma \vdash (o^m\ M_1 \ldots\ M_m) : T$

  By the type rules and by induction,

$$\begin{aligned}&\mathcal{V}(M_1[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], B_1),\\ &\ldots\\ &\mathcal{V}(M_n[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], B_n)\end{aligned}$$

  Thus, M will eventually reduce to $(o^n\ b_1 \ldots\ b_m)$. According to Assumption 1, δ must provide a value V such that $\vdash V : T$. According to Assumption 2, V contains no primitive operations. Thus, we can apply the weaker Lemma 12.9 to conclude $\mathcal{V}(V, T)$. By an extension of Lemma 12.8 and by lifting out subsitutions, we can conclude that

$$\mathcal{V}((o^m\ M_1 \ldots\ M_m)[X_1 \leftarrow V_1] \ldots [X_n \leftarrow V_n], B_1), T)$$

  as desired.

Chapter 13: Variations on Simply Typed ISWIM

Since all Simply Typed ISWIM programs terminate, clearly we have lost some expressive power in the typed version of ISWIM. In particular, we can no longer write recursive programs. As we'll see in the following sub-sections, we have also lost the ability to encode complex conditionals, or to encode data structures such as pairs.

To recover useful kinds of expressiveness in a typed setting, we can refine the core type rules so that they allow more programs, including the ones that implement conditionals and pairs. We explore that approach in future chapters.

Meanwhile, in this chapter, we explore a simpler approach: extend Simply Typed ISWIM with explicit constructs for conditionals, pairs, etc. With these constructs in the language, we can define type rules that work directly on the constructs. In other words, the explicit constructs provide a place to hang specific type rules that directly reflect our intuition about the constructs.

13.1 Conditionals

In Chapter 11, we defined a language of booleans and numbers with a built-in `if` form for conditionals. In contrast, for plain ISWIM, we used the encoding or true and false that we first developed for the pure λ-calculus of Chapter 3, where true is $(\lambda x.\lambda y.x)$ and false is $(\lambda x.\lambda y.y)$. The `iszero` primitive operation cooperates with this encoding, mapping a number to either $(\lambda x.\lambda y.x)$ or $(\lambda x.\lambda y.y)$, and the `if0` macro combines the function encoding of true and false with applications to achieve a conditional branch.

For Simply Typed ISWIM, we have to pick typed functions to use for encodings of true and false. Indeed, when assigning a Δ mapping for `iszero`, we implicitly picked a typed encoding; the result type for `iszero` was defined as $(\texttt{num} \to (\texttt{num} \to \texttt{num}))$, which is the type of $(\lambda x{:}\texttt{num}.\lambda y{:}\texttt{num}.x)$ and $(\lambda x{:}\texttt{num}.\lambda y{:}\texttt{num}.y)$. Thus, we can write

$$((\texttt{iszero}\ \ulcorner 0 \urcorner)\ \ulcorner 1 \urcorner\ \ulcorner 2 \urcorner)$$

which evaluates to $\ulcorner 1 \urcorner$, and is also well-typed. But, unfortunately, we cannot redefine the if0 macro to work with this encoding:

$$\begin{array}{rcl}(\textsf{if0}\ K\ M\ N) & \doteq & (((\texttt{iszero}\ K)\ (\lambda X{:}\texttt{num}.M)\ (\lambda X{:}\texttt{num}.N))\ \ulcorner 0 \urcorner) \\ & & \text{where } X \notin \mathcal{FV}(M) \cup \mathcal{FV}(N)\end{array}$$

The problem is that the first argument to the result of $(\texttt{iszero}\ K)$ will have a type $(\texttt{num} \to \_)$, while the typed encoding of true and false expects an argument of type num. To solve this problem, we might choose a different encoding for booleans:

$$\begin{array}{rcl}\textsf{true} & \doteq & (\lambda x{:}(\texttt{num} \to \texttt{num}).\lambda y{:}(\texttt{num} \to \texttt{num}).x) \\ \textsf{false} & \doteq & (\lambda x{:}(\texttt{num} \to \texttt{num}).\lambda y{:}(\texttt{num} \to \texttt{num}).y)\end{array}$$

In that case, if Δ gives `iszero` the result type

$$((\texttt{num} \to \texttt{num}) \to ((\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})))$$

then an expression like $(\textsf{if0}\ \ulcorner 0 \urcorner\ \ulcorner 1 \urcorner\ \ulcorner 2 \urcorner)$ will have a type:

$$\dfrac{\dfrac{\dfrac{\dots}{\vdash ((\texttt{iszero}\ \ulcorner 0 \urcorner)\ (\lambda x{:}\texttt{n}.\ulcorner 1 \urcorner)) : ((\texttt{n} \to \texttt{n}) \to (\texttt{n} \to \texttt{n}))} \quad \dfrac{\dots}{\vdash ((\lambda x{:}\texttt{n}.\ulcorner 2 \urcorner)) : (\texttt{n} \to \texttt{n})}}{\vdash ((\texttt{iszero}\ \ulcorner 0 \urcorner)\ (\lambda x{:}\texttt{n}.\ulcorner 1 \urcorner)\ (\lambda x{:}\texttt{n}.\ulcorner 2 \urcorner)) : (\texttt{n} \to \texttt{n})} \quad \vdash \ulcorner 0 \urcorner : \texttt{n}}{\vdash (((\texttt{iszero}\ \ulcorner 0 \urcorner)\ (\lambda x{:}\texttt{n}.\ulcorner 1 \urcorner)\ (\lambda x{:}\texttt{n}.\ulcorner 2 \urcorner))\ \ulcorner 0 \urcorner) : \texttt{n}}$$

Neverthless, the new encoding is still limited. Conditionals using `iszero` can only be well-typed when they produce numbers. We have lost the ability to write conditionals that produce functions, as in

$$(\mathsf{if0}\ \ulcorner 0 \urcorner\ (\lambda x{:}\mathtt{num}.(\mathtt{add1}\ x))\ (\lambda x{:}\mathtt{num}.(\mathtt{sub1}\ x)))$$

The easiest way to support such expressions is to add booleans and a conditional form directly into Simply Typed ISWIM.

$$\begin{array}{lcl} M & = & \ldots \\ & | & \mathtt{f} \\ & | & \mathtt{t} \\ & | & (\mathtt{if}\ M\ M\ M) \\ T & = & \ldots \\ & | & \mathtt{bool} \\ \\ \Delta(\mathtt{iszero}) & = & \langle\langle \mathtt{num} \rangle, \mathtt{bool}\rangle \end{array}$$

$$\frac{\Gamma \vdash M_1 : \mathtt{bool} \quad \Gamma \vdash M_2 : T \quad \Gamma \vdash M_3 : T}{\Gamma \vdash (\mathtt{if}\ M_1\ M_2\ M_3) : T}$$

▷ **Exercise 13.1.** Complete the extension of Simply Typed ISWIM for booleans and `if`, including new type rules for `t` and `f`, extended definitions for contexts and evaluation contexts, and new evaluation rules to go with β_v and δ. Then show that

$$(\mathtt{if}\ (\mathtt{iszero}\ \ulcorner 0 \urcorner)\ (\lambda x{:}\mathtt{num}.(\mathtt{add1}\ x))\ (\lambda x{:}\mathtt{num}.(\mathtt{sub1}\ x)))$$

has a type, and that it reduces to a value.

13.2 Pairs

Recall the encoding of pairs from Chapter 3:

$$\begin{array}{lcl} \mathsf{mkpair} & \doteq & \lambda x.\lambda y.\lambda s.(s\ x\ y) \\ \mathsf{fst} & \doteq & \lambda p.(p\ \lambda x.\lambda y.x) \\ \mathsf{snd} & \doteq & \lambda p.(p\ \lambda x.\lambda y.y) \end{array}$$

The problem with these encodings in a typed setting is similar to the problem with true and false: we have to pick a specific type for x and y in mkpair. Unlike the problem with conditionals, however, no primitive operations are involved that require a fixed type for pairs. We might therefore try to define a *family* of abbreviations, one for each pair of types T_1 and T_2, where T_1 is the type of the first element in the pair, and T_2 is the type for the second element.

$$\begin{array}{lcl} \mathsf{mkpair}_{(T_1 \times T_2)} & \doteq & \lambda x{:}T_1.\lambda y{:}T_2.\lambda s{:}(T_1 \to (T_2 \to \_)).(s\ x\ y) \\ \mathsf{fst}_{(T_1 \times T_2)} & \doteq & \lambda p{:}((T_1 \to (T_2 \to T_1)) \to T_1).(p\ \lambda x{:}T_1.\lambda y{:}T_2.x) \\ \mathsf{snd}_{(T_1 \times T_2)} & \doteq & \lambda p{:}((T_1 \to (T_2 \to T_2)) \to T_2).(p\ \lambda x{:}T_1.\lambda y{:}T_2.y) \end{array}$$

This encoding nearly works. The problem is that $\mathsf{fst}_{(T_1 \times T_2)}$ and $\mathsf{snd}_{(T_1 \times T_2)}$ must return different kinds of values if $T_1 \neq T_2$, but the two functions flow into the same s in $\mathsf{mkpair}_{(T_1 \times T_2)}$. Thus, there is no good type to put in place of $\_$ in the definition of $\mathsf{mkpair}_{(T_1 \times T_2)}$, unless we restrict T_1 and T_2 to be the same.

We can solve the problem more generally by adding pairs and pair types directly to Simply Typed ISWIM. The form $\langle M, N\rangle$ pairs the values of M and N together, and the `fst` and `snd` forms extract a value from a pair. The type for a pair is $(T_1 \times T_2)$.

$$
\begin{array}{rcl@{\quad}rcl@{\quad}rcl@{\quad}rcl}
M & = & \ldots & V & = & \ldots & E & = & \ldots & T & = & \ldots \\
 & | & \langle M, M\rangle & & | & \langle V, V\rangle & & | & \langle E, M\rangle & & | & (T \times T) \\
 & | & (\texttt{fst}\ M) & & & & & | & \langle V, E\rangle & & & \\
 & | & (\texttt{snd}\ M) & & & & & | & (\texttt{fst}\ E) & & & \\
 & & & & & & & | & (\texttt{snd}\ E) & & &
\end{array}
$$

$$
\begin{array}{lll}
(\texttt{fst}\ \langle V_1, V_2\rangle) & \mathbf{p} & V_1 \\
(\texttt{snd}\ \langle V_1, V_2\rangle) & \mathbf{p} & V_2
\end{array}
\qquad \mathbf{v} = \ldots \cup \mathbf{p}
$$

$$
\frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2}{\Gamma \vdash \langle M_1, M_2\rangle : (T_1 \times T_2)}
$$

$$
\frac{\Gamma \vdash M : (T_1 \times T_2)}{\Gamma \vdash (\texttt{fst}\ M) : T_1}
\qquad
\frac{\Gamma \vdash M : (T_1 \times T_2)}{\Gamma \vdash (\texttt{snd}\ M) : T_2}
$$

▷ **Exercise 13.2.** Using both the conditional and pair extensions, show that $(\texttt{fst}\ \langle(\texttt{add1}\ \ulcorner 0\urcorner), \texttt{t}\rangle)$ has a type and a value.

13.3 Variants

Some functions need to accept multiple kinds of values that implement variants of an abstract datatype. For example, a list-processing function must accept both a constant representing the empty list, null, and a cons cell $(\mathsf{cons}\ v\ l)$, where v is the first element in the list and l is the rest of the list. In addition to accepting both kinds of values, the function must somehow be able to distinguish them.

We can implement such variants in untyped ISWIM by using pairs and attaching a different tag to each variant. The true tag can identify the empty list, and the false tag can identify a cons cell:

$$
\begin{array}{lcl}
\mathsf{null} & \doteq & \langle \mathsf{true}, \mathsf{false}\rangle \\
\mathsf{cons} & \doteq & \lambda v.\lambda l.\langle \mathsf{false}, \langle v, l\rangle\rangle \\
\mathsf{isnull} & \doteq & \lambda l.\mathsf{fst}\ l
\end{array}
$$

In a typed language, the above encoding does not work. The null constant has the type $(\texttt{bool} \times \texttt{bool})$, whereas a cons cell has type $(\texttt{bool} \times (T \times T'))$ for some T and T'. These types are incompatible, so we cannot write an isnull function that accepts arguments of both shapes.

Once again, we solve the problem by extending our core language. We will extend it with a notion of **variants** (or **disjoint sums**), represented by the type $(T_1 + T_2)$ for variant types T_1 and T_2. A value of type $(T_1 + T_2)$ encapsulates either a value of type T_1 or a value of type T_2.

Three new forms operate on variants of type $(T_1 + T_2)$: $\mathsf{ToLeft}_{(T_1+T_2)}$, $\mathsf{ToRight}_{(T_1+T_2)}$, and $\mathsf{Match}_{((T_1+T_2)\to T)}$.

- $(\mathsf{ToLeft}_{(T_1+T_2)}\ M)$
  This form computes the value for M, which must be of type T, and encapsulates it as value of type $(T_1 + T_2)$. In other words, $\mathsf{ToLeft}_{(T_1+T_2)}$ **injects** a value of type T into $(T_1 + T_2)$.

- $(\mathsf{ToRight}_{(T_1+T_2)}\ M)$
  This form injects the value of M, of type T_2, into $(T_1 + T_2)$.

- $(\mathsf{Match}_{((T_1+T_2)\to T)}\ M\ M_1\ M_2)$
  This form requires M to produce a value of type $(T_1 + T_2)$. The M_1 and M_2 expressions must each produce a function value; the M_1 function must have type $(T_1 \to T)$, and the M_2 function must have type $(T_2 \to T)$. The $\mathsf{Match}_{((T_1+T_2)\to T)}$ form picks an appropriate function, depending on whether the value for M encapsulates a value of type T_1 or T_2. If it's T_1, then M_1 is invoked with the encapsulated value, otherwise M_2 is invoked.

With the new forms, we can take a step towards encoding lists. Lists of arbitrary length require even more typing machinery than we have developed so far, but we can at least define a datatype that encapsulates either an empty list or a "list" containing a single number:

$$\begin{array}{lcl}
\mathsf{null} & \doteq & (\mathsf{ToLeft}_{(\mathtt{bool}+\mathtt{num})}\ \mathtt{f}) \\
\mathsf{list1} & \doteq & \lambda v{:}\mathtt{num}.(\mathsf{ToRight}_{(\mathtt{bool}+\mathtt{num})}\ v) \\
\mathsf{isnull} & \doteq & \lambda l{:}(\mathtt{bool}+\mathtt{num}).\ (\mathsf{Match}_{((\mathtt{bool}+\mathtt{num})\to\mathtt{bool})}\ l \\
 & & \qquad (\lambda x{:}\mathtt{bool}.\mathtt{t}) \\
 & & \qquad (\lambda x{:}\mathtt{num}.\mathtt{f}))
\end{array}$$

$$\begin{array}{lcl}
M & = & \ldots \\
 & | & (\mathsf{ToLeft}_{(T+T)}\ M) \\
 & | & (\mathsf{ToRight}_{(T+T)}\ M) \\
 & | & (\mathsf{Match}_{((T+T)\to T)}\ M\ M\ M) \\
 & & \\
V & = & \ldots \\
 & | & (\mathsf{ToLeft}_{(T+T)}\ V) \\
 & | & (\mathsf{ToRight}_{(T+T)}\ V)
\end{array}
\qquad
\begin{array}{lcl}
E & = & \ldots \\
 & | & (\mathsf{ToLeft}_{(T+T)}\ E) \\
 & | & (\mathsf{ToRight}_{(T+T)}\ E) \\
 & | & (\mathsf{Match}_{((T+T)\to T)}\ E\ M\ M) \\
 & | & (\mathsf{Match}_{((T+T)\to T)}\ V\ E\ M) \\
 & | & (\mathsf{Match}_{((T+T)\to T)}\ V\ V\ E) \\
 & & \\
T & = & \ldots \\
 & | & (T+T)
\end{array}$$

$$\begin{array}{lcl}
(\mathsf{Match}_{((T_1+T_2)\to T)}\ (\mathsf{ToLeft}_{(T_1+T_2)}\ V)\ V_1\ V_2) & \mathbf{s} & (V_1\ V) \\
(\mathsf{Match}_{((T_1+T_2)\to T)}\ (\mathsf{ToRight}_{(T_1+T_2)}\ V)\ V_1\ V_2) & \mathbf{s} & (V_2\ V)
\end{array}$$

$$\mathbf{v} = \ldots \cup \mathbf{s}$$

$$\frac{\Gamma \vdash M : T_1}{\Gamma \vdash (\mathsf{ToLeft}_{(T_1+T_2)}\ M) : (T_1 + T_2)}$$

$$\frac{\Gamma \vdash M : T_2}{\Gamma \vdash (\mathsf{ToRight}_{(T_1+T_2)}\ M) : (T_1 + T_2)}$$

$$\frac{\Gamma \vdash M : (T_1 + T_2) \quad \Gamma \vdash M_1 : (T_1 \to T) \quad \Gamma \vdash M_2 : (T_2 \to T)}{\Gamma \vdash (\mathsf{Match}_{((T_1+T_2)\to T)}\ M\ M_1\ M_2) : T}$$

▷ **Exercise 13.3.** using the macro definitions above, show that (isnull (list1 ⌜13⌝)) has a type and a value.

13.4 Recursion

Using the conditional extension from Section 13.1, we can define a typed version of the mksum function:

$$\mathsf{mksum} \doteq \lambda f{:}(\mathtt{num}\to\mathtt{num}).\lambda n{:}\mathtt{num}.(\mathtt{if}\ (\mathtt{iszero}\ n)\ \ulcorner 0\urcorner\ (+\ n\ (f\ (\mathtt{sub1}\ n))))$$

But we know that Simply Typed ISWIM doesn't allow recursion, so we should expect that

$$\mathsf{Y}_\mathsf{v}\ \mathsf{mksum}$$

has no type. Why?

Clearly mksum has a type. Applying the type rules we have assigns mksum the type $((\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num}))$. In other words, given a function from numbers to numbers, mksum produces a function from numbers to numbers.

The problem is that $\mathsf{Y_v}$ itself has no type, for the same reason that Ω has no type. We could avoid this problem by extending the language with an explicit $\mathsf{Y_v}$ function and giving it an appropriate type. But since the untyped $\mathsf{Y_v}$ works on many kinds of functions, we would have to define a family of $\mathsf{Y_v}$ functions. One member of the family would create recursive functions of type $(\texttt{num} \to \texttt{num})$, another would create functions of type $(\texttt{num} \to \texttt{bool})$, and so on.

Alternatively, we can add a `fix` form to the language that acts like $\mathsf{Y_v}$, but automatically adapts to the target function type. A proto-recursive function (such as mksum) takes a function of type $(T_1 \to T_2)$ to generate a function of type $(T_1 \to T_2)$. Thus, the type rule for `fix` is

$$\frac{\Gamma \vdash M : ((T_1 \to T_2) \to (T_1 \to T_2))}{\Gamma \vdash (\texttt{fix}\ M) : (T_1 \to T_2)}$$

Since T_1 and T_2 only appear within $(T_1 \to T_2)$, we can simplify the rule by using T in place of $(T_1 \to T_2)$. The complete extension of the language is as follows:

$$\begin{array}{rcl@{\qquad\qquad}rcl} M & = & \ldots & E & = & \ldots \\ & | & (\texttt{fix}\ M) & & | & (\texttt{fix}\ E) \end{array}$$

$$(\texttt{fix}\ (\lambda X{:}T.M)) \quad \mathbf{y} \quad M[X \leftarrow (\texttt{fix}\ (\lambda X{:}T.M))]$$

$$\mathbf{v} = \ldots \cup \mathbf{y}$$

$$\frac{\Gamma \vdash M : (T \to T)}{\Gamma \vdash (\texttt{fix}\ M) : T}$$

The $\mathbf{y}$ reduction rule implements the essense of the fixed-point equiation directly, so `fix` certainly acts like $\mathsf{Y_v}$. And, using the new type rule we have

$$\frac{\dfrac{\ldots}{\vdash \mathsf{mksum} : ((\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num}))}}{\vdash (\texttt{fix}\ \mathsf{mksum}) : (\texttt{num} \to \texttt{num})}$$

Thus, (`fix` mksum) is the recursive multiplication function for a type language.

The soundness theorem for Simply Typed ISWIM holds for the langauge extended with `fix`. This is easy to see; if $(\texttt{fix}\ (\lambda X{:}T.M))$ has type T, then X can be replaced by $(\texttt{fix}\ (\lambda X{:}T.M))$ in M without changing the type of M.

The normalization theorem no longer holds, as demonstrated by the following infinite loop:

$$((\texttt{fix}\ \lambda f{:}(\texttt{num} \to \texttt{num}).\lambda n{:}\texttt{num}.(f\ n))\ \ulcorner 0 \urcorner)$$

▷ **Exercise 13.4.** Prove that $\vdash (\texttt{fix}\ (\lambda x{:}T.x)) : T$ for any T. How is it possible that $(\texttt{fix}\ (\lambda x{:}T.x))$ manufactures a value of type T out of thin air? For example, what number does $(\texttt{fix}\ (\lambda x{:}\texttt{num}.x))$ produce when it is evaluated?

Chapter 14: Polymorphism

With the extensions of the previous chapter, we can write many interesting programs in Simply Typed ISWIM. We cannot necessarily write them as conveniently, however. For example, instead of writing a single identity function

$$\lambda x.x$$

we have to write for one every kind of value that we want to use as an argument to the identity function

$$\lambda x{:}\texttt{num}.x$$
$$\lambda x{:}\texttt{bool}.x$$
$$\lambda x{:}(\texttt{num} \to \texttt{num}).x$$
$$\lambda x{:}((\texttt{bool} \to \texttt{num}) + (\texttt{num} \times \texttt{bool})).x$$

This duplication of code is wasteful—especially when the function is more complex than identity.

In this chapter, we extend our language and type rules so that we can once again write functions, such as identity, that are **polymorphic**: they work simultaneously on many kinds (or "shapes") of data.[1]

A function that does not inspect its argument in any way is clearly a candidate for polymorphism, but some functions inspect their arguments only partially. For example, a swap function

$$\lambda p.\langle \mathsf{snd}\ p, \mathsf{fst}\ p\rangle$$

works only on pair obejcts, but it does not inspect the values in the pair. Thus, this function is polymorphic, because it works on many different kinds of pairs.

The kind of abstraction that we need over many types is similar to the kind of abstraction that λ provides over many specific values. This observation suggests that we extend Simply Typed ISWIM with a mechanism for abstracting an expression over a type, and combining a type-abstracted function with a specific type to get a specific function. The resulting language is **Polymorphic ISWIM**. (Note: In the same way that the functional core of ISWIM is called the λ-calculus, the functional core of Polymorphic ISWIM is sometimes called **System F**.)

14.1 Polymorphic ISWIM

To distinguish type abstractions from normal abstractions, we use Λ instead of λ for type abstractions. In addition, we keep type identifiers and variable identifiers separate, by designating a separate set of identifiers for types. Lowercase Roman letters are used for variable identifiers, and lowercase Greek letters for type identifers.

Thus, the polymorphic identify function is

$$\mathsf{id} \doteq \Lambda\alpha.\lambda x{:}\alpha.x$$

where α is a placeholder for an unknown type (in the same way that x is a placeholder for an unknown value).

Along with application for normal functions to values, we need type application for polymophic functions to types. We distinguish type applications from normal applications by using infix square brackets around a type argument. Thus, to apply id to the number $\ulcorner 5 \urcorner$, we write

$$(\mathsf{id}[\texttt{num}]\ \ulcorner 5 \urcorner)$$

[1] The kinf of polymorhpism that we explore in this chapter is sometimes called **parameteric polymorphism**. In Chapter 18, we will study a different kind of polymorphism.

The num type argument gets substituted for α in $(\lambda x{:}\alpha.x)$ to produce a normal function, which can then be applied to $\ulcorner 5 \urcorner$ in the usual way.

A polymorphic swap function needs two types: one for the first element in the original pair, and one for the second element of the original pair:

$$\mathsf{swap} \doteq \Lambda\alpha.\Lambda\beta.\lambda p{:}(\alpha \times \beta).\langle \mathtt{snd}\ p, \mathtt{fst}\ p\rangle$$

By applying the polymorphic swap to num and bool

$$\mathsf{swap}[\mathtt{num}][\mathtt{bool}]$$

we obtain a function of type $((\mathtt{num} \times \mathtt{bool}) \to (\mathtt{bool} \times \mathtt{num}))$.

What is the type of swap itself or of id, in polymorphic form? The id expression is not quite a function, because it must be applied to type arguments to obtain a function. Nevertheless, the type for id should indicate that it produces a function, and that the type of the function depends on the supplied type arguments. To write such types, we use $\forall$ with a type variable:

$$\vdash \mathsf{id} : (\forall\alpha.(\alpha \to \alpha))$$

Type expressions of this form are called **universal types**.

In the same way that swap contains two Λs, its type contains two $\forall$s:

$$\vdash \mathsf{swap} : (\forall\alpha.(\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))))$$

The swap example shows that type expressions, such as $(\alpha \times \beta)$, can include type variables, such as α and β. Thus, our new expression and type grammars are as follows:

$$\begin{array}{rcl} M & = & \ldots \\ & | & (\Lambda A.M) \\ & | & M[T] \\ V & = & \ldots \\ & | & (\Lambda A.M) \\ T & = & \ldots \\ & | & A \\ & | & (\forall A.T) \\ A & = & \text{a type variable: } \alpha, \beta, \ldots \end{array}$$

The reduction rules are extended to include type applications:

$$(\Lambda A.M)[T] \quad \beta_{\mathsf{t}} \quad M[A \leftarrow T]$$

$$\mathbf{v} = \ldots \cup \beta_{\mathsf{t}}$$

where substitution for type variables is defined in the obvious way, and $\mathcal{FV}(T)$ extracts the free type variables of T in the obvious way.

To assign a type to Polymorphic ISWIM experssions, we need to track type variables in an envionment along with normal variables. Up to now, Γ has mapped normal variables to types. For Polymorphic ISWIM, we extend Γ so that it also contains type variables. The type variables have no bindings (the implicit binding would be "is a type"), but they are needed in the environment to disallow free type variables. For example, the expression

$$\lambda x{:}\alpha.x$$

should not have a type, because α has no meaning, in the same way that the expression

$$x$$

without any context has no type. The revised definition of Γ follows:

$$
\begin{array}{rcl}
\Gamma & = & \epsilon \\
 & | & \Gamma, X{:}T \\
 & | & \Gamma, A
\end{array}
$$

$$
\begin{array}{lcll}
\Gamma(X) & = & T & \text{if } \Gamma = \Gamma', X{:}T \\
\Gamma(X) & = & \Gamma'(X) & \text{if } \Gamma = \Gamma', X'{:}T \text{ and } X \neq X' \\
\Gamma(X) & = & \Gamma'(X) & \text{if } \Gamma = \Gamma', A
\end{array}
$$

$$
\begin{array}{lcll}
\Gamma(A) & & & \text{if } \Gamma = \Gamma', A \\
\Gamma(A) & \text{when} & \Gamma'(A) & \text{if } \Gamma = \Gamma', A' \text{ and } A \neq A' \\
\Gamma(A) & \text{when} & \Gamma'(A) & \text{if } \Gamma = \Gamma', X'{:}T
\end{array}
$$

The type rules for type abstraction and type application use the environment with type variables. Furthermore, in expressions with type expressions (including λ expressions), the type expression must be checked to ensure that it has no free type variables; the $\_ \vdash \_$ relation performs this check.

$$\frac{\Gamma, A \vdash M : T}{\Gamma \vdash (\Lambda A.M) : (\forall A.T)} \qquad \frac{\Gamma \vdash M : (\forall A.T') \quad \Gamma \vdash T}{\Gamma \vdash M[T] : T'[A \leftarrow T]}$$

$$\frac{\Gamma, X{:}T \vdash M : T' \quad \Gamma \vdash T}{\Gamma \vdash (\lambda X{:}T.M) : (T \rightarrow T')}$$

$$\Gamma \vdash A \text{ if } \Gamma(A) \qquad \Gamma \vdash B$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2}{\Gamma \vdash (T_1 \rightarrow T_2)} \qquad \ldots \textit{pair types, etc.}\ldots \qquad \frac{\Gamma, A \vdash T}{\Gamma \vdash (\forall A.T)}$$

Type abstractions introduce yet one more wrinkle into the type-checking process. The type rule for function application sets up an implicit equation between the type of the argument expression, and the argument type of the function expression. But now that we have Λ abstractions that parameterize over an arbitrary type variable, our type grammar includes expressions that should be treated the same, but are not syntactically equivalent. For example, $(\Lambda\alpha.(\alpha \rightarrow \alpha))$ and $(\Lambda\beta.(\beta \rightarrow \beta))$ should be equivalent.

To allow such types to be treated the same, we must modify the application rule to use an explicit equality check, and we must also define a set of rules $\vdash \_ \leftrightarrow \_$ for type equivalence.

$$\frac{\Gamma \vdash M : (T' \rightarrow T) \quad \Gamma \vdash N : T'' \quad \vdash T' \leftrightarrow T''}{\Gamma \vdash (M\ N) : T}$$

$$\vdash T \leftrightarrow T \qquad \frac{\vdash T_1 \leftrightarrow T_1' \quad \vdash T_2 \leftrightarrow T_2'}{\vdash (T_1 \rightarrow T_2) \leftrightarrow (T_1' \rightarrow T_2')} \qquad \ldots$$

$$\frac{\vdash T[A \leftarrow A''] \leftrightarrow T'[A' \leftarrow A'']}{\vdash (\forall A.T) \leftrightarrow (\forall A'.T')} \quad \text{where } A'' \notin \mathcal{FV}(T) \cup \mathcal{FV}(T')$$

▷ **Exercise 14.1.** Show that $\vdash \mathsf{swap} : (\forall\alpha.(\forall\beta.((\alpha \times \beta) \rightarrow (\beta \times \alpha))))$.

▷ **Exercise 14.2.** Although languages with polymorhpism typically also have a built-in notion of pair, Polymorphic ISWIM supports the pure λ-calculus encoding of pairs, after the encoding is augmented with appropriate types, Λ abstractions, and type applications. Show this encoding (i.e., show versions of mkpair, fst, and snd that are well-typed in Polymorphic ISWIM).

Chapter 15: Type Inference

Now that we have polymorphism in our language, we can write interesting programs without duplicating functionality to satisfy the type system. Nevertheless, the type system still demands a certain amount of work from the programmer that was not required in plain ISWIM. For example, a programmer must write Λ and α in

$$\Lambda\alpha.\lambda x{:}\alpha.x$$

instead of simply

$$\lambda x.x$$

to define the polymorphic identity function.

In this chapter, we study **type inference** (or **type reconstruction**), which relives the programmer of writing type annotations, but does not sacrifice the benefits of a typed language. The **Type-Inferred ISWIM** language allows a programmer to write expressions that look like plain ISWIM, but the expressions are more like Simply Typed ISWIM expressions with all of the type abstractions, type applications, and function argument types erased. An inference phase in Type-Inferred ISWIM effectively reconstructs the erased types and checks the result. Thus, a program in Type-Inferred ISWIM is has a type only if the inferencer can both fill in types and apply the rules of Simply Typed ISWIM.

After we have developed an inferencer for Simply Typed ISWIM, we can get most of the power of Polymorphic ISWIM by having the inferencer reconstruct Λ abstractions and type applications.

> The techniques and notation used in this chapter are derived from Benjamin Pierce's *Type Systems for Programming Languages* [7].

15.1 Type-Inferred ISWIM

For Type-Inferred ISWIM expressions, we revert to the original ISWIM grammar without type annotations:

$$\begin{array}{rcl} M & = & X \\ & | & (\lambda X.M) \\ & | & (M\ M) \\ & | & b \\ & | & (o^n\ M\ \ldots\ M) \end{array}$$

To define type rules Type-Inferred ISWIM expressions, we might take the rules from Simply Typed ISWIM and omit the type declarations on function arguments:

$$\frac{\Gamma, X{:}T \vdash M : T'}{\Gamma \vdash (\lambda x.M) : (T \to T')} \qquad \ldots$$

The above is a well-defined rule, but it's not obviously decidable like the Simply Typed ISWIM rules. In Simply Typed ISWIM, to check a λ expression, a type checker could extend the current type environment with the declared argument type, check the body expression in the extended environment, and finally construct a function type using the declared type on the left-hand side and the body-type on the right-hand side.

Without a declared argument type, a type checker matching the above rule will have to "guess" a T to use as the argument type. For example,

$$((\lambda x.x)\ \ulcorner 0 \urcorner)$$

The expression above could have type `num`, where the reconstructed type for the x argument is `num`. From this simple example, it might appear that the type rule for application can supply a "guess" to the rule for function by first checking the argument expression, and then using the resulting type for the function argument. That strategy will not work, however, because functions can be arguments, too:

$$((\lambda f.(f\ \ulcorner 0 \urcorner))\ (\lambda x.x))$$

In this example, we cannot type check $(\lambda x.x)$ without guessing `num` for x. At the same time, we cannot check $(\lambda f.(f\ \ulcorner 0 \urcorner))$ without guessing $(\texttt{num} \rightarrow \texttt{num})$ for f.

The second example shows that reconstruction in general requires a global view of the expression being typed. Although the layout of functions and function applications will generate relationships among the types of expressions, expressions with related types can be arbitrarily far apart in the source expression. A "magic" type checker could guess locally the right type for every function variable, so that an overall proof tree fits together, but we need a more mechanical solution.

The mechanical solution is based on **constraint generation** and **unification**. Constraint generation means that the type rules do not assign a concrete type to exach expression, but instead generate type names to be resolved later. The rules also generate constraints that encode relations among different type names. Unification then solves the constraints, assigning a concrete type to every type name generated by the type rules.

15.1.1 Constraint Generation

To generate constraints, we replace the $\_ \vdash \_ : \_$ rule by a $\_ \vdash \_ : \_ \,|\, \_ ; \_$ rule. The right-hand end of the new relation consists of two new parts. One is a set of constraints $\mathcal{C}$, matching type expressions to other type expressions; each type expressions contains type variables that must be resolved to make the paired type expressions equivalent. The other new part is a set of type variables $\mathcal{X}$ that lists the variables generated so far; this set is needed so that we can generate fresh type variables to avoid conflicts.

$$\begin{array}{rcl} \mathcal{C} & = & \text{a set } \{T{=}T, \ldots\} \\ \mathcal{X} & = & \text{a set } \{A, \ldots\} \end{array}$$

$$\not\frown(\{\mathcal{X}_1, \ldots \mathcal{X}_n\}) \quad \text{if} \quad \mathcal{X}_i \cap \mathcal{X}_j = \emptyset \quad \text{for} \quad i, j \in [1, n], i \neq j$$

The new rule for functions makes type-guessing explicit by generating a new type variable for the argument:

$$\frac{\Gamma, X{:}A \vdash M : T' \,|\, \mathcal{C}; \mathcal{X}}{\Gamma \vdash (\lambda X.M) : (A \rightarrow T') \,|\, \mathcal{C}; \mathcal{X} \cup \{A\}} \quad \text{where } A \notin \mathcal{X}$$

The new rule for function application no longer requires the type of the argument expression to syntactically match the argument type of the function expression. Instead, the rule finds a type and set of constraints for each expression, and then combines the constraints with a new one that enforces the argument-type relationship:

$$\frac{\Gamma \vdash M : T_1 \,|\, \mathcal{C}_1; \mathcal{X}_1 \quad \Gamma \vdash N : T_2 \,|\, \mathcal{C}_2; \mathcal{X}_2}{\Gamma \vdash (M\ N) : A \,|\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1{=}(T_2 \rightarrow A)\}; \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}}$$

$$\text{where} \quad \not\frown(\{\mathcal{X}_1, \mathcal{X}_2, \{A\}\})$$

The condition $\not\wr(\{\mathcal{X}_1, \mathcal{X}_2, \{A\}\})$ ensures that the type variables generated by the subexpression are distinct and different from A, so that the constraint sets $\mathcal{C}_1$ and $\mathcal{C}_2$ can be safely merged with the new constraint.

The rules for basic constraints and variables are essentially unchanged; they do not generate constraints or type variables:

$$\Gamma \vdash X : T \,|\, \emptyset; \emptyset \quad \text{if} \quad \Gamma(X) = T$$
$$\Gamma \vdash b : B \,|\, \emptyset; \emptyset \quad \text{if} \quad \mathcal{B}(b) = B$$

The rule for primitive applications is more complex. Each sub-expression M_i generates a type, but also a set of constraints and type variables. The overall set of constraints and type variables includes the ones generated by the sub-expressions, but it also includes constraints mapping the sub-expressions types to the types expected by the primitive constant.

$$\frac{\Gamma \vdash M_1 : T_1 \,|\, \mathcal{C}_1; \mathcal{X}_1 \ \ldots \quad \Gamma \vdash M_n : T_n \,|\, \mathcal{C}_n; \mathcal{X}_n}{\Gamma \vdash (o^n\ M_1 \ldots\ M_n) : T \,|\, \mathcal{C}_1 \cup \ldots \mathcal{C}_n \cup \{T_1{=}B_1, \ldots T_n{=}B_n\}; \mathcal{X}_1 \cup \ldots \mathcal{X}_n}$$

where $\Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, T\rangle$
and $\not\wr(\{\mathcal{X}_1, \ldots \mathcal{X}_n\})$

We can apply these rules to our earlier example:

$$\frac{\dfrac{x{:}\alpha_1 \vdash x : \alpha_1 \,|\, \emptyset; \emptyset}{\vdash (\lambda x.x) : (\alpha_1 \to \alpha_1) \,|\, \emptyset; \{\alpha_1\}} \qquad \vdash \ulcorner 0 \urcorner : \texttt{num} \,|\, \emptyset; \emptyset}{\vdash ((\lambda x.x)\ \ulcorner 0 \urcorner) : \alpha_2 \,|\, \{(\alpha_1 \to \alpha_1){=}(\texttt{num} \to \alpha_2)\}; \{\alpha_1, \alpha_2\}}$$

this derivation does not quite provide a type for the expression. Instead, it provides the "type" α_2 with the constraint $(\alpha_1 \to \alpha_1){=}(\texttt{num} \to \alpha_2)$. Since the constraints require α_1 to be `num` and α_2 to be α_1, we can easily see that α_2 is `num`.

The derivation for our second example is somewhat more involved:

$$\frac{\dfrac{\dfrac{f{:}\alpha_1 \vdash f : \alpha_1 \,|\, \emptyset; \emptyset \quad f{:}\alpha_1 \vdash \ulcorner 0 \urcorner : \texttt{num} \,|\, \emptyset; \emptyset}{f{:}\alpha_1 \vdash (f\ \ulcorner 0 \urcorner) : \alpha_2 \,|\, \{\alpha_1{=}(\texttt{num} \to \alpha_2)\}; \{\alpha_2\}}}{\vdash (\lambda f.(f\ \ulcorner 0 \urcorner)) : (\alpha_1 \to \alpha_2) \,|\, \{\alpha_1{=}(\texttt{num} \to \alpha_2)\}; \{\alpha_1, \alpha_2\}} \qquad \dfrac{x{:}\alpha_3 \vdash x : \alpha_3 \,|\, \emptyset; \emptyset}{\vdash (\lambda x.x) : (\alpha_3 \to \alpha_3) \,|\, \emptyset; \{\alpha_3\}}}{\vdash ((\lambda f.(f\ \ulcorner 0 \urcorner))\ (\lambda x.x)) : \alpha_4 \,|\, \{\alpha_1{=}(\texttt{num} \to \alpha_2), (\alpha_1 \to \alpha_2){=}((\alpha_3 \to \alpha_3) \to \alpha_4)\}; \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}}$$

By considering the resulting constraints carefully, we can see that α_4 must be `num`, which is what we expected. But, although *we* can see α_4 must be `num`, our goals is to make an *type checker* extract this result. The unification phase of our checker serves that role.

▷ **Exercise 15.1.** Show the constraint-generation tree for

$$(\lambda x.(+\ 0\ ((\lambda y.y)\ x)))$$

15.1.2 Unification

The unification phase of our type checker takes the constraints generated by the rules in the previous subsection and solve them. The unification process is similar to solving for variables in a set of algebra equations. For example, given the algebra equations

$$x + y = 16$$

$$x + 3y = 8$$

we can derive $x = 16 - y$ from the first equation, then substitute $16 - y$ for x in the second equation. Manipulations on the second equation then yield $y = -4$, which we can substitute back into $16 - y$ to get $x = 20$.

For the analogous resolution of type constraints, we need a way to pull out equations to simplify the constraint set, plus a way to keep track of substitutions for the final values of removed variables. The following mapping $\mathcal{U}$ is defined recursively to perform a single simplification step while building up a result mapping.

$$
\begin{array}{lcll}
\mathcal{U}(\{A{=}T\} \cup \mathcal{C}) & = & \mathcal{U}(\mathcal{C}[A \leftarrow T]), A{=}T & A \notin \mathcal{FV}(T) \\
\mathcal{U}(\{T{=}A\} \cup \mathcal{C}) & = & \mathcal{U}(\mathcal{C}[A \leftarrow T]), A{=}T & A \notin \mathcal{FV}(T) \\
\mathcal{U}(\{(T_1 \rightarrow T_2){=}(T_3 \rightarrow T_4)\} \cup \mathcal{C}) & = & \mathcal{U}(\mathcal{C} \cup \{T_1{=}T_3, T_2{=}T_4\}) & \\
\mathcal{U}(\{T{=}T\} \cup \mathcal{C}) & = & \mathcal{U}(\mathcal{C}) & \\
\mathcal{U}(\emptyset) & = & \epsilon &
\end{array}
$$

The result of $\mathcal{U}$ is a sequence of equations $\epsilon, A_1{=}T_1, \ldots A_n{=}T_n$. In this sequence, T_1 is the final type for A_1. Substituting T_1 for A_1 in T_2 provides the final type for A_2, etc. For example, taking the constraints generated by $((\lambda x.x)\ \ulcorner 0 \urcorner)$:

$$
\begin{array}{l}
\mathcal{U}(\{(\alpha_1 \rightarrow \alpha_1){=}(\texttt{num} \rightarrow \alpha_2)\}) \\
\quad = \ \mathcal{U}(\{\alpha_1{=}\texttt{num}, \alpha_1{=}\alpha_2\}) \\
\quad = \ \mathcal{U}(\{\texttt{num}{=}\alpha_2\}), \alpha_1{=}\texttt{num} \\
\quad = \ \mathcal{U}(\emptyset), \alpha_2{=}\texttt{num}, \alpha_1{=}\texttt{num} \\
\quad = \ \epsilon, \alpha_2{=}\texttt{num}, \alpha_1{=}\texttt{num}
\end{array}
$$

from which we can immediately extract that α_2 is `num`. Taking the constraints generated by $((\lambda f.(f\ \ulcorner 0 \urcorner))\ (\lambda x.x))$:

$$
\begin{array}{l}
\mathcal{U}(\{\alpha_1{=}(\texttt{num} \rightarrow \alpha_2), (\alpha_1 \rightarrow \alpha_2){=}((\alpha_3 \rightarrow \alpha_3) \rightarrow \alpha_4)\}) \\
\quad = \ \mathcal{U}(\{((\texttt{num} \rightarrow \alpha_2) \rightarrow \alpha_2){=}((\alpha_3 \rightarrow \alpha_3) \rightarrow \alpha_4)\}), \alpha_1{=}(\texttt{num} \rightarrow \alpha_2) \\
\quad = \ \mathcal{U}(\{(\texttt{num} \rightarrow \alpha_2){=}(\alpha_3 \rightarrow \alpha_3), \alpha_2{=}\alpha_4\}), \alpha_1{=}(\texttt{num} \rightarrow \alpha_2) \\
\quad = \ \mathcal{U}(\{\texttt{num}{=}\alpha_3, \alpha_2{=}\alpha_3, \alpha_2{=}\alpha_4\}), \alpha_1{=}(\texttt{num} \rightarrow \alpha_2) \\
\quad = \ \mathcal{U}(\{\alpha_2{=}\texttt{num}, \alpha_2{=}\alpha_4\}), \alpha_3{=}\texttt{num}, \alpha_1{=}(\texttt{num} \rightarrow \alpha_2) \\
\quad = \ \mathcal{U}(\{\texttt{num}{=}\alpha_4\}), \alpha_2{=}\texttt{num}, \alpha_3{=}\texttt{num}, \alpha_1{=}(\texttt{num} \rightarrow \alpha_2) \\
\quad = \ \mathcal{U}(\emptyset), \alpha_4{=}\texttt{num}, \alpha_2{=}\texttt{num}, \alpha_3{=}\texttt{num}, \alpha_1{=}(\texttt{num} \rightarrow \alpha_2) \\
\quad = \ \epsilon, \alpha_4{=}\texttt{num}, \alpha_2{=}\texttt{num}, \alpha_3{=}\texttt{num}, \alpha_1{=}(\texttt{num} \rightarrow \alpha_2)
\end{array}
$$

In this case, we can see immediately that α_4 is `num`. By straightforward substitution, we also find that α_1 is $(\texttt{num} \rightarrow \texttt{num})$.

In general, the $\mathcal{R}([\_], \_)$ relation extracts the concrete binding of a type variable from a sequence produced by $\mathcal{U}$:

$$
\begin{array}{lcll}
\mathcal{R}([\epsilon, A_1{:}T_1, \ldots A_n{:}T_n], A_i) & = & T_i[A_1 \leftarrow T_1] \ldots [A_{i-1} \leftarrow T_{i-1}] & i \in [1, n] \\
\mathcal{R}([\epsilon, A_1{:}T_1, \ldots A_n{:}T_n], A) & = & A & A \notin \{A_1, \ldots A_n\}
\end{array}
$$

The rules for $\mathcal{U}$ are not deterministic, because we can resolve variables in any order that we want, but a mechanical unifier might choose the left-to-right order that we used above. In some cases, however, no $\mathcal{U}$ rule matches, regardless of the chosen variable order. For example, if we generate the constraints for half of Ω:

$$
\frac{\dfrac{x{:}\alpha_1 \vdash x : \alpha_1 \,|\, \emptyset; \emptyset \qquad x{:}\alpha_1 \vdash x : \alpha_1 \,|\, \emptyset; \emptyset}{x{:}\alpha_1 \vdash (x\ x) : \alpha_2 \,|\, \{\alpha_1{=}(\alpha_1 \rightarrow \alpha_2)\}; \{\alpha_2\}}}{\vdash (\lambda x.(x\ x)) : (\alpha_1 \rightarrow \alpha_2) \,|\, \{\alpha_1{=}(\alpha_1 \rightarrow \alpha_2)\}; \{\alpha_1, \alpha_2\}}
$$

we end up with the constaint set $\{\alpha_1{=}(\alpha_1 \to \alpha_2)\}$, which does not match any of the $\mathcal{U}$ rules. This result is expected: Ω has no type in Simply Typed ISWIM, not even if a mechanical process is guessing the type.

In other cases, the constraints and rules for $\mathcal{U}$ do not completely determine a concrete type for an expression. Consider the expression $(\lambda x.x)$:

$$\frac{x{:}\alpha_1 \vdash x : \alpha_1 \,|\, \emptyset; \emptyset}{\vdash (\lambda x.x) : (\alpha_1 \to \alpha_1) \,|\, \emptyset; \{\alpha_1\}}$$

The result type is $(\alpha_1 \to \alpha_1)$. This result is consistent our experience with polymorphic types: the identity function can have any type it needs, depending on the context. Without a context, we are free to choose any value of α_1 that we'd like.

Nevertheless, Type-Inferred ISWIM *does not* infer a polymorphic type for $(\lambda x.x)$. In the context

$$((\lambda f.(f\ (\lambda y.y))\ (f\ \ulcorner 0 \urcorner))\ (\lambda x.x))$$

the type for f will have to be unified with both $(\texttt{num} \to T_1)$ and $((T_3 \to T_3) \to T_4)$. Since `num` and $(T_2 \to T_3)$ cannot be unified, the expression will have no type.

▷ **Exercise 15.2.** Show how $\mathcal{U}$ solves the constraints generated in Exercise 15.1.

15.2 Inferring Polymorphism

We can get the power of Polymorphic ISWIM by having the inferencer reconstruct Λ and type applications at certain places in an expression. More specifically, we require the programmer to designate points for polymorphic inference using a new `let` form.

The `let` form causes the type inferencer to assign a type immediately and locally to the expression for the `let`-bound variable. If the resulting type leaves certain variables in the overall type unconstrained, then the variables are parameterized with $\forall$, making the `let`-bound value polymorphic.

For example, the expression

$$(\texttt{let}\ f\ \texttt{=}\ (\lambda x.x)\ \texttt{in}\ ((f\ (\lambda y.y))\ (f\ \ulcorner 0 \urcorner)))$$

will have a type, because f's binding will be polymorphic.

The function that converts a type expression to a polymorphic type expression is $\mathcal{P}(\_, \_)$. The second argument to $\mathcal{P}(\_, \_)$ is a set of variables that might be unresolved:

$$\begin{array}{lcll} \mathcal{P}(T, \emptyset) & = & T & \\ \mathcal{P}(T, \{A\} \cup \mathcal{X}) & = & \mathcal{P}(\forall A.T, \mathcal{X}) & \text{where} \quad A \in \mathcal{FV}(T) \\ \mathcal{P}(T, \{A\} \cup \mathcal{X}) & = & \mathcal{P}(T, \mathcal{X}) & \text{where} \quad A \notin \mathcal{FV}(T) \end{array}$$

The rule for `let` uses $\mathcal{U}$, $\mathcal{R}$, and $\mathcal{P}$ to assign a type to the `let`-bound variable.

$$\frac{\Gamma \vdash M : T \,|\, \mathcal{C}_1; \mathcal{X}_1 \qquad \Gamma, X{:}T' \vdash N : T'' \,|\, \mathcal{C}_2; \mathcal{X}_2}{\Gamma \vdash (\texttt{let}\ X\ \texttt{=}\ M\ \texttt{in}\ N) : T'' \,|\, \mathcal{C}_1 \cup \mathcal{C}_2; \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}}$$

$$\text{where } T' = \mathcal{P}(\mathcal{R}([\mathcal{U}(\mathcal{C}_1 \cup \{A{=}T\})], A), \mathcal{X}_1) \quad \text{and} \quad \not\wr(\{\mathcal{X}_1, \mathcal{X}_2, \{A\}\})$$

The `let` rule use uses $\mathcal{P}$ with only the variables $\mathcal{X}_1$ introduced during the checking of M; the type expression T may contain other variables that are introduced by the surrounding context, and those must be resolved later. For the same reason, the `let` rule propagates the constraints

$\mathcal{C}_1$, which may contain constraints that are relevant to the larger program, even though they have been resolved once for M. In addition to the useful constraints, $\mathcal{C}_1$ may include constaints that are useless outside of M, but they are also harmless.

Inserting type abstraction with an implicit Λ is only half of the problem. The checker must also insert implicit type applications. We can add a general rule that converts from polymorphic types to normal types

$$\frac{\Gamma \vdash M : \forall A.T \,|\, \mathcal{C}; \mathcal{X}}{\Gamma \vdash M : T[A \leftarrow A'] \,|\, \mathcal{C}; \mathcal{X} \cup \{A'\}} \quad \text{where } A \notin \mathcal{X}$$

Since we are relying on type inference in general, the above rule does not pick a specific A'; unification will find an appropriate type later.

Unfortunately, adding this rule makes the type checker non-deterministic. When checking a function application and the argument expression has a polymorphic type, when should we reduce the type with the above rule, and when should we leave it alone? A related problem is that an expression may have a polymorphic type, but the polymorphic type that is only visible until after unification. Extending the unifier with polymorphism makes unification "higher-order," and much more difficult.

Disallowing polymorphic functions as arguments solves all of these problems. In other words, a function argument can have a reconstructed type α, but it cannot have reconstructed type $\forall\alpha.T$. To enforce this rule, we need only add a restriction to the function and primitive application rules so that no sub-expression type is of the form $\forall A.T$. This restriction forces polymorphic resolution where it is necessary, and ensures that no type expression of the form $\forall A.T$ shows up in a generated constraint.

Applying the polymorphic rules to the example above starts with the following tree

$$\frac{\dfrac{x{:}\alpha_1 \vdash x : \alpha_1 \,|\, \emptyset;\emptyset}{\vdash (\lambda x.x) : (\alpha_1 \to \alpha_1) \,|\, \emptyset;\{\alpha_1\}} \quad \dfrac{\dots}{f{:}\forall\alpha_1.(\alpha_1 \to \alpha_1) \vdash ((f\ (\lambda y.y))\ (f\ \ulcorner 0 \urcorner)) : \dots \mid \dots;\dots}}{\vdash (\texttt{let}\ f\ \texttt{=}\ (\lambda x.x)\ \texttt{in}\ ((f\ (\lambda y.y))\ (f\ \ulcorner 0 \urcorner))) : \dots \mid \dots;\dots}$$

If we abbreviate $\forall\alpha_1.(\alpha_1 \to \alpha_1)$ with T_0 in environments, the top-right subtree will contain a derivation for $(f\ \ulcorner 0 \urcorner)$ where f's type is instantiated as $(\alpha_2 \to \alpha_2)$:

$$\frac{\dfrac{f{:}T_0 \vdash f : \forall\alpha_1.(\alpha_1 \to \alpha_1) \,|\, \emptyset;\emptyset}{f{:}T_0 \vdash f : (\alpha_2 \to \alpha_2) \,|\, \emptyset;\{\alpha_2\}} \quad f{:}T_0 \vdash \ulcorner 0 \urcorner : \texttt{num} \,|\, \emptyset;\emptyset}{f{:}T_0 \vdash (f\ \ulcorner 0 \urcorner) : \alpha_3 \,|\, \{(\alpha_2 \to \alpha_2){=}(\texttt{num} \to \alpha_3)\};\{\alpha_2,\alpha_3\}}$$

Meanwhile, the derivation for $(f\ (\lambda y.y))$ instatiates a different type for f, $(\alpha_4 \to \alpha_4)$:

$$\frac{\dfrac{f{:}T_0 \vdash f : \forall\alpha_1.(\alpha_1 \to \alpha_1) \,|\, \emptyset;\emptyset}{f{:}T_0 \vdash f : (\alpha_4 \to \alpha_4) \,|\, \emptyset;\{\alpha_4\}} \quad \dfrac{f{:}T_0, y{:}\alpha_5 \vdash y : \alpha_5 \,|\, \emptyset;\emptyset}{f{:}T_0 \vdash (\lambda y.y) : (\alpha_5 \to \alpha_5) \,|\, \emptyset;\{\alpha_5\}}}{f{:}T_0 \vdash (f\ (\lambda y.y)) : \alpha_6 \,|\, \{(\alpha_4 \to \alpha_4){=}((\alpha_5 \to \alpha_5) \to \alpha_6)\};\{\alpha_4,\alpha_5,\alpha_6\}}$$

Due to the separate instantiations for f's type, the final collection of constraints will have a solution showing the type of the original expression as `num`.

Chapter 16: Recursive Types

Throughout our exploration of types so far, we have encountered expressions that could not be given a type because an equation of the form $T_1 = \ldots T_1 \ldots$ could not be solved with a concrete type for T_1. For example, Ω has no type because $T_1 = (T_1 \to T_2)$ has no solution for T_1. The same problem appears with more useful recursive functions that are written in the style of Chapter 3.

As we saw in Chapter 13, we can add `fix` to the language to support recursive functions, but a programmer still cannot write Ω or λ-calculus-style recursive functions directly. For the same reasons that motivate type inference, we might prefer to type plain ISWIM expressions as much as possible, including Ω, without forcing a programmer write down type-specific annotations. Inference for general recursive functions requires a way to solve type constraints of the form $T_1 = \ldots T_1 \ldots$.

More importantly, recursive type equations show up in data structures as well as functions. For example, the definition of a list of numbers is inherently recursive:

> A list of numbers is either
>
> - null, or
> - (cons n l) where n is a number and l is a list of numbers.

We know how to deal with the case split, using disjoint sums, and we can use `f` for null and pairs for cons. But the second part of the sum type must be a pair type that refers back to the entire sum. In other words, the type $\mathsf{List}_{\mathtt{num}}$ must obey the following equation:

$$\mathsf{List}_{\mathtt{num}} = (\mathtt{bool} + (\mathtt{num} \times \mathsf{List}_{\mathtt{num}}))$$

Following Chapter 13, we could solve the problem by building lists directly into the type system. But the more general approach of solving recursive equations is especially valuable to the programmer in the case of data, as opposed to functions. A recursive function is a recursive function, but a recursive data structure can have many different shapes: a list, a tree, etc. Supporting all possible data shapes in the core type rules requires a general form of recursive types. In this chapter, we study recursive types by extending Polymorphic ISWIM (i.e., we ignore inference from now on).

16.1 Fixed-points of Type Abstractions

To abstract expressions over types (e.g., to get polymorphic functions), we drew inspiration from our expression language, where we could already parameterize expressions over values. As a result, we added type abstractions ($\Lambda\alpha.M$) and type applications $M[T]$ to the grammar of expressions, and universal types ($\forall\alpha.T$) to the grammar of type expressions.

To solve a recursive equation like the one for $\mathsf{List}_{\mathtt{num}}$, we can similarly draw on our experience with recursive functions: we can add a `tfix` form for type expressions that takes a type abstraction and produces it's fixed point:

$$\mathsf{List}_{\mathtt{num}} \quad \doteq \quad \mathtt{tfix}\ (\forall\alpha.(\mathtt{bool} + (\mathtt{num} \times \alpha)))$$

Adding `tfix` to our type language will work, but by convention, a `tfix` and its following $\forall$ are collapsed together into a single μ:

$$\mathsf{List}_{\mathtt{num}} \quad \doteq \quad \mu\alpha.(\mathtt{bool} + (\mathtt{num} \times \alpha))$$

A type constructed with μ is a **recursive type**. In the same way that f in $(\texttt{fix}\ \lambda f.(\lambda x.(f\ x))$ stands for the entire recursive function within $(\lambda x.(f\ x))$, the α above stands for the entire recursive type within $(\texttt{bool} + (\texttt{num} \times \alpha))$.

As another example, consider the expression:

$$\lambda x.(+\ (x\ x)\ \ulcorner 5 \urcorner)$$

Due to the call $(x\ x)$, The type of x must be a function type. Since the result of the call is used in an addition expression, the result of the function must be a number. Meanwhile, the argument of the function must have the same type as the function itself. Thus, the type for x above is

$$\mu\alpha.(\alpha \to \texttt{num})$$

and the type for the entire function expression is

$$((\mu\alpha.(\alpha \to \texttt{num})) \to \texttt{num})$$

The type for the function shows how μ can appear embedded within a larger type expression. To support recursive types in general, we extend our grammar for T as follows:

$$\begin{array}{rcl} T & = & \ldots \\ & | & (\mu A.T) \end{array}$$

16.2 Equality Between Recursive Types

One difference between recursive functions and recursive types is that we generally do not need to compare two function values, but type checking requires that we compare type expressions for equality. Of course, the type variable bound by a μ type is arbitrary, and we should equate types that are simple "α-conversions" of each other, just as we did for $\forall$ type expressions:

$$\mu\alpha.(\alpha \to \texttt{num}) \quad \text{is the same as} \quad \mu\beta.(\beta \to \texttt{num})$$

But parallels between `fix` and μ suggest some additional complexities in the comparison process. For example,

$$(\texttt{fix}\ \lambda f.(\lambda x.(f\ x))$$

and

$$\lambda x.((\texttt{fix}\ \lambda f.(\lambda x.(f\ x)))\ x)$$

are equivalent expressions, in the sense that they behave the same way in all contexts. Similarly,

$$\mu\alpha.(\alpha \to \texttt{num})$$

and

$$((\mu\alpha.(\alpha \to \texttt{num})) \to \texttt{num})$$

and

$$(((\mu\alpha.(\alpha \to \texttt{num})) \to \texttt{num}) \to \texttt{num})$$

are all equivalent types, in the sense that an expression having any of these types can be used as an expression with any of the other types. Type rules therefore require an extended $\vdash \_ \leftrightarrow \_$ relation (as introduced in §14.1) that captures the equality of types containing μ.

$$\frac{\vdash T[A \leftarrow A''] \leftrightarrow T'[A' \leftarrow A'']}{\vdash (\mu A.T) \leftrightarrow (\mu A'.T')} \quad \text{where } A'' \notin \mathcal{FV}(T) \cup \mathcal{FV}(T')$$

$$\frac{\vdash T[A \leftarrow (\mu A.T)] \leftrightarrow T'}{\vdash (\mu A.T) \leftrightarrow T'} \qquad \frac{\vdash T' \leftrightarrow T[A \leftarrow (\mu A.T)]}{\vdash T' \leftrightarrow (\mu A.T)}$$

This equivalence relation lets us **unfold** a type $(\mu A.T)$ by replacing each A in T with $(\mu A.T)$. Unfolding a recursive type can put it into a form that matches another type.

In addition to extending the type-equivalence relation, the function application rule must allow a μ type for the first sub-expression, instead of an immediate function type. In other words, instead of requiring the function expression to have type $(T' \to T)$, it need only produce a type that is equivalent to some $(T' \to T)$.

$$\frac{\Gamma \vdash M : T_0 \qquad \vdash T_0 \leftrightarrow (T' \to T) \qquad \Gamma \vdash N : T'' \qquad \vdash T' \leftrightarrow T''}{\Gamma \vdash (M\ N) : T}$$

We now have two equivalances to prove in the application rule, $\vdash T_0 \leftrightarrow (T' \to T)$ and $\vdash T' \leftrightarrow T''$. They can be collapsed into a single rule: $\vdash T_0 \leftrightarrow (T'' \to T)$:

$$\boxed{\frac{\Gamma \vdash M : T_0 \qquad \vdash T_0 \leftrightarrow (T'' \to T) \qquad \Gamma \vdash N : T''}{\Gamma \vdash (M\ N) : T}}$$

Using the new rules, we can assign a type to half of $\Omega_{\texttt{num}}$, where $\texttt{r}$ abbreviates $(\mu\alpha.(\alpha \to \texttt{num}))$:

$$\frac{x{:}\texttt{r} \vdash x : \texttt{r} \qquad \dfrac{\vdash (\texttt{r} \to \texttt{num}) \leftrightarrow (\texttt{r} \to \texttt{num})}{\vdash \texttt{r} \leftrightarrow (\texttt{r} \to \texttt{num})} \qquad x{:}\texttt{r} \vdash x : \texttt{r}}{\vdash (\lambda x{:}\texttt{r}.(x\ x)) : (\texttt{r} \to \texttt{num})}$$

To show the type equivalence in the top middle of the tree, we expand the abbreviation $\texttt{r}$ as $(\mu\alpha.(\alpha \to \texttt{num}))$, unfold it to $((\mu\alpha.(\alpha \to \texttt{num})) \to \texttt{num})$, and re-abbreviate as $(\texttt{r} \to \texttt{num})$.

The rules above define a powerful type system for programming with recursive types. Such recursive types are call **equi-recursive** types, because the type rules relate a recursive type and its unfolded version as equivalent. The main cost of equi-recursive types is that type checking does not have a straightforward algorithm. When equating recursive types, multiple $\vdash \_ \leftrightarrow \_$ rules can apply, and a winning strategy for building a complete tree is not always apparent. Furthermore, combining recursive types with type inference would aggravate the problem considerably.

▷ **Exercise 16.1.** Show that $\Omega_{\texttt{num}}$, expressed as

$$(\lambda x{:}(\mu\alpha.(\alpha \to \texttt{num})).(x\ x))\ (\lambda x{:}(\mu\alpha.(\alpha \to \texttt{num})).(x\ x))$$

has a type. What if every `num` is replaced above by an arbitrary type T?

16.3 Isomorphisms Between Recursive Types

An alternative to the powerful equational system required by equi-recursive types is to have the programmer explicitly specify the places in an expression where recursive types must be folded or unfolded. Such explicit annotations, which generate **iso-recursive types**, guide type checking and inference for recursive types in the same way that a `let` form can guide type inference for ploymophism.

To support iso-recursive programmer annotations, we must extend the grammar of expressions with `fold` and `unfold` forms:

$$\boxed{\begin{array}{rcl} M & = & \ldots \\ & | & (\texttt{fold}\ M) \\ & | & (\texttt{unfold}\ M) \end{array}}$$

During evaluation, a `fold` annotation sticks to a value until it is dismissed by an `unfold`:

$$\begin{array}{rcl} V & = & \ldots \\ & | & (\texttt{fold}\ V) \end{array} \qquad \begin{array}{rcl} E & = & \ldots \\ & | & (\texttt{fold}\ E) \\ & | & (\texttt{unfold}\ E) \end{array}$$

$$(\texttt{unfold}\ (\texttt{fold}\ V))\quad \mathbf{f}\quad V \qquad\qquad \mathbf{v} = \ldots \cup \mathbf{f}$$

The primary role of `fold` and `unfold` is in the type checking rules. We revert the $\vdash \_ \leftrightarrow \_$ relation back to its simple form, from Chapter 14, as well as the function application rule. All we need for type checking are two new rules:

$$\frac{\Gamma \vdash M : T[A \leftarrow (\mu A.T)]}{\Gamma \vdash (\texttt{fold}\ M) : (\mu A.T)} \qquad \frac{\Gamma \vdash M : (\mu A.T)}{\Gamma \vdash (\texttt{unfold}\ M) : T[A \leftarrow (\mu A.T)]}$$

Using iso-recursive types, a well-typed half of $\Omega_{\texttt{num}}$ requires a strategically placed `unfold` annotation:

$$\lambda x{:}(\mu\alpha.(\alpha \to \texttt{num})).((\texttt{unfold}\ x)\ x)$$

This expression can be typed using the new `unfold` rule, where r still abbreviates $(\mu\alpha.(\alpha \to \texttt{num}))$:

$$\frac{\dfrac{x{:}\mathsf{r} \vdash x : \mathsf{r}}{x{:}\mathsf{r} \vdash (\texttt{unfold}\ x) : (\mathsf{r} \to \texttt{num})} \qquad x{:}\mathsf{r} \vdash x : \mathsf{r} \qquad \vdash \mathsf{r} \leftrightarrow \mathsf{r}}{\vdash (\lambda x{:}\mathsf{r}.((\texttt{unfold}\ x)\ x)) : (\mathsf{r} \to \texttt{num})}$$

Writing all of Ω requires a `fold` annotation in addition:

$$(\lambda x{:}(\mu\alpha.(\alpha \to \texttt{num})).((\texttt{unfold}\ x)\ x))\ (\texttt{fold}\ \lambda x{:}(\mu\alpha.(\alpha \to \texttt{num})).((\texttt{unfold}\ x)\ x))$$

▷ **Exercise 16.2.** Show that the representation of Ω above has a type.

16.4 Using Iso-Recursive Types

Iso-recursive types simplify the type checking process, and also work with a straightforward extension to the type-inference algorithm developed in the preceding chapter.

Realistic programming languages generally do not support recursive types to implement recursive functions, because `fix` (or the implicit `fix` in a `let rec` form) works well enough. ML, for example, does not support recursive function types, but it provides iso-recursive types through datatype declarations. Nevertheless, ML has no explicit `fold` and `unfold` keyword; instead, each use of a constructor to create a value implies a `fold`, and each use of a constructor in a pattern match implies `unfold`.

Using recursive and polymorphic types, we can (finally!) define a well-typed, general-purpose implementation of lists. Our implementation is much like an ML implementation, but with explcit Λ and `fold` annotations.

We start with a macro for the type of lists containing elements of type T, and an unfolded version of the same type:

$$\begin{array}{lcl} \mathsf{List}_T & \doteq & \mu\alpha.(\texttt{bool} + (T \times \alpha)) \\ \mathsf{UList}_T & \doteq & (\texttt{bool} + (T \times \mathsf{List}_T)) \end{array}$$

We can use this type macro in defining our expression macros. For example, the cons macro will be a polymorphic function of type $\forall\alpha.(\alpha \to (\mathsf{List}_\alpha \to \mathsf{List}_\alpha))$. Even null must be polymorphic, because the null for different kinds of lists will have a different type.

What should car and cdr do when they are given null as an argument? Our language currently has no mechanism for signalling errors, and soundness prevents the macros from getting stuck on bad inputs. One sneaky solution is to loop forever, exploting the fact that Ω can be constructed to "return" any type of value (see Exercise 13.4).

$$
\begin{array}{lcl}
\mathsf{null} & \doteq & \Lambda\alpha.(\mathtt{fold}\ (\mathsf{ToLeft}_{\mathsf{UList}_\alpha}\ \mathtt{f})) \\
\mathsf{cons} & \doteq & \Lambda\alpha.(\lambda v{:}\alpha.\lambda l{:}\mathsf{List}_\alpha.(\mathtt{fold}\ (\mathsf{ToRight}_{\mathsf{UList}_\alpha}\ \langle v, l\rangle))) \\
\mathsf{car} & \doteq & \Lambda\alpha.(\lambda l{:}\mathsf{List}_\alpha.(\mathsf{Match}_{(\mathsf{UList}_\alpha\to\alpha)}\ (\mathtt{unfold}\ l) \\
 & & \qquad (\lambda n{:}\mathtt{bool}.\Omega[\alpha]) \\
 & & \qquad (\lambda p{:}(\alpha\times\mathsf{List}_\alpha).(\mathtt{fst}\ p)))) \\
\mathsf{cdr} & \doteq & \Lambda\alpha.(\lambda l{:}\mathsf{List}_\alpha.(\mathsf{Match}_{(\mathsf{UList}_\alpha\to\mathsf{List}_\alpha)}\ (\mathtt{unfold}\ l) \\
 & & \qquad (\lambda n{:}\mathtt{bool}.\Omega[\mathsf{List}_\alpha]) \\
 & & \qquad (\lambda p{:}(\alpha\times\mathsf{List}_\alpha).(\mathtt{snd}\ p)))) \\
\mathsf{isnull} & \doteq & \Lambda\alpha.(\lambda l{:}\mathsf{List}_\alpha.(\mathsf{Match}_{(\mathsf{UList}_\alpha\to\mathtt{bool})}\ (\mathtt{unfold}\ l) \\
 & & \qquad (\lambda n{:}\mathtt{bool}.\mathtt{t}) \\
 & & \qquad (\lambda p{:}(\alpha\times\mathsf{List}_\alpha).\mathtt{f}))) \\
\Omega & \doteq & \Lambda\alpha.(\ (\lambda x{:}(\mu\beta.(\beta\to\alpha)).((\mathtt{unfold}\ x)\ x)) \\
 & & \qquad (\mathtt{fold}\ \lambda x{:}(\mu\beta.(\beta\to\alpha)).((\mathtt{unfold}\ x)\ x))\ )
\end{array}
$$

The null and cons macros act like ML type constructors, and as expected, they each contain a `fold` form. Similarly, the car, cdr, and isnull macros act like deconstructing pattern matches, so each uses `unfold` on its argument.

▷ **Exercise 16.3.** Show that

$$(\mathsf{car}[\mathtt{num}]\ (\mathsf{cons}[\mathtt{num}]\ \ulcorner 1\urcorner\ \mathsf{null}[\mathtt{num}]))$$

reduces to $\ulcorner 1\urcorner$, and show that it has type `num`. To simplify the type derivation, assume that we have already shown $\Gamma \vdash \Omega[T] : T$ for all Γ and T.

Chapter 17: Data Abstraction and Existential Types

Type soundness ensures that applications of primitive operations do not "go wrong" in the sense that primitive operations are never applied to bad arguments. The information that a program cannot go wrong is perhaps most useful to a programmer, but language implementors can also put that information to work. For example, zero and false can be represented in the same way, without the possibility of strange behavior when false is added to a number. Similarly, functions can be represented by memory addresses, without the possibility that a number will be mis-used as the address of a non-existent function.

In other words, types can hide some of the implementation details of a language. The power to hide implementation details can be useful to programmers, as well as language implementors, particularly for building modularized software. For example, given our implementation of null and cons from the previous chapter, we'd prefer that client programmers using cons always use car to extract the first element of a list, so that we can change the implementation of cons later if becomes necessary. Such good programming style is not enforced by the type language, however, at least not for the way we wrote the code in the previous chapter. The type List_T is merely an abbreviation, and completely transparent to client programmers. Using the expanded type, a client programmer can manipulate list internals directly, either intentionally or accidentally.

To hide the details of the list type from the programmer, we can force client programmers to implement list-using code in a particular way: client code must have the type $\forall\alpha.T$ for some T, where α serves as the type for lists. Since the client code is polymorphic with respect to list values, the client cannot rely on any implementation details about lists.

This approach to data abstraction can codified through a specific extension of the programming language: **existential types**. Like a universal type $\forall A.T$, an existential type $\exists A.T$ describes a value that works with arguments of an unspecified type A. In the case of a universial type, the unspecified type can be replaced by *any* type. In the case of an existential type, the unspecified type can be replaced by only *one particular* type. That one particular type is hidden from client programmers, and the type system can enforce the hiding. As well will see, the difference between $\forall A.T$ and $\exists A.T$ is essentially two levels of procedure indirection.

17.1 Data Abstraction in Clients

Lists are one example of a datatype that we might like to keep abstract from clients, but our current implementation of lists is perhaps the only one that makes sense. For an example where different implemenations of the abstract type could make sense, suppose that a particular program requires a few numbers that are known at start-up, uses the numbers repeatedly in addition expressions, and then requires a fast odd/even parity test. In this case, the abstract datatype is the type of parity-testable numbers, and an implementation of the datatype will provide functions for construction the initial numbers, adding numbers, and checking the parity of a number.

If the base language provides numbers and bitwise operators, the parity test is trivial. Thus, one implementation of the datatype exploits a fast `isodd` function that is built into the language implementation:

$$
\begin{array}{lcl}
\mathsf{Num}_1 & \doteq & \texttt{num} \\
\mathsf{mknum}_1 & \doteq & \lambda n{:}\texttt{num}.n \\
\mathsf{add}_1 & \doteq & \lambda n{:}\mathsf{Num}_1.\lambda m{:}\mathsf{Num}_1.(+\ n\ m) \\
\mathsf{parity}_1 & \doteq & \lambda n{:}\mathsf{Num}_1.(\texttt{isodd}\ n) \\
\mathsf{nfuncs}_1 & \doteq & \langle \mathsf{mknum}_1, \langle \mathsf{add}_1, \mathsf{parity}_1 \rangle \rangle
\end{array}
$$

The nfuncs_1 tuple bundles the number operations into a tuple that acts like a "library module" to be supplied to client programmers.

If the base language does not provide bitwise operators (or other operators that can be used to test for even and odd), then we can optimize the parity test by computing the parity of the initial numbers, tracking parity during additions, and then extracting the parity value of the result. Thus, a second implementation of the datatype uses ($\texttt{num} \times \texttt{bool}$) to implement parity-maintained numbers:

$$\begin{array}{lcl}
\mathsf{Num}_2 & \doteq & (\texttt{num} \times \texttt{bool}) \\
\mathsf{mknum}_2 & \doteq & \lambda n{:}\texttt{num}.\langle n,\ (\mathsf{isodd}\ n)\rangle \\
\mathsf{add}_2 & \doteq & \lambda n{:}\mathsf{Num}_2.\lambda m{:}\mathsf{Num}_2.\langle(+\ (\texttt{fst}\ n)\ (\texttt{fst}\ m)),\ (\mathsf{xor}\ (\texttt{snd}\ n)\ (\texttt{snd}\ m))\rangle \\
\mathsf{parity}_2 & \doteq & \lambda n{:}\mathsf{Num}_2.(\texttt{snd}\ n) \\
\mathsf{nfuncs}_2 & \doteq & \langle \mathsf{mknum}_2,\ \langle \mathsf{add}_2,\ \mathsf{parity}_2\rangle\rangle
\end{array}$$

where isodd and xor are defined in the obvious way. (The isodd function will require $O(n)$ time for an integer n, while xor is constant time.)

Each module-like tuple nfuncs_i has type $\mathsf{NFuncs}_{\mathsf{Num}_i}$ where

$$\mathsf{NFuncs}_T \ \doteq\ ((\texttt{num} \to T) \times ((T \to (T \to T)) \times (T \to \texttt{bool})))$$

Instead of supplying the tuple nfuncs_1 or nfuncs_2 directly to clients, we can hide the specific implementation type Num_i by requiring clients to have the form

$$\forall\alpha.(\mathsf{NFuncs}_\alpha \to T)$$

In other words, client code must be represented by a polymorphic function that cannot rely on the details of the number type α. The function consumes a tuple representing the number operations. The body of the function corresponds to the client code proper; it uses the operations to produce some value of type T.

For example, here is a client that adds 17 and 18, then extracts the parity of the result:

$$\begin{array}{lcl}
\mathsf{client} & \doteq & \Lambda\alpha.\lambda n{:}\mathsf{NFuncs}_\alpha \quad (\mathsf{let}\ \ m{:}(\texttt{num} \to \alpha) = (\texttt{fst}\ n) \\
& & \qquad\qquad\qquad\qquad\quad\ \ a{:}(\alpha \to (\alpha \to \alpha)) = (\texttt{fst}\ (\texttt{snd}\ n)) \\
& & \qquad\qquad\qquad\qquad\quad\ \ p{:}(\alpha \to \texttt{bool}) = (\texttt{snd}\ (\texttt{snd}\ n)) \\
& & \qquad\quad \mathsf{in}\ (p\ (a\ (m\ \ulcorner 17 \urcorner)\ (m\ \ulcorner 18 \urcorner)))))
\end{array}$$

where the multiple-variable let macro has the obvious expansion. It extracts the functions of the "imported module" tuple n into the local variables m, a, and p.

To execute client, we must apply it to one of the implemenations of numbers, either

$$(\mathsf{client}[\mathsf{Num}_1]\ \mathsf{pfunctions}_1)$$

or

$$(\mathsf{client}[\mathsf{Num}_2]\ \mathsf{pfunctions}_2)$$

The type system ensures that both applications will work, as long as client has the $\forall\alpha\ldots$ type above (for some specific T).

17.2 Libraries that Enforce Abstraction

Our goal is to forcing clients to respect data abstraction, but the enforcement is backward. In practice, it is the library implementor who should enforce generality, and not the linking programmer, so that the library implementor is free to implement a library in different ways.

If a library implementor provided nfuncs_1 or nfuncs_2 directly to other programmers, the other programmers could rely on the type Num_1 or Num_2 directly, conflicting with a future revision of the library that changes the implementation type for numbers.

This problem is easily corrected by supplying to programmers a library as a linking procedure, instead of as a tuple. In other words, the library provided by a programmer will combine nfuncs_1 or nfuncs_2 with the linking application:

$$\begin{array}{lcl} \mathsf{Client}_T & \doteq & \forall\alpha.(\mathsf{NFuncs}_\alpha \to T) \\ \mathsf{nlib}_1 & \doteq & \Lambda\beta.\lambda c{:}\mathsf{Client}_\beta.(c[\mathsf{Num}_1]\ \mathsf{pfunctions}_1) \\ \mathsf{nlib}_2 & \doteq & \Lambda\beta.\lambda c{:}\mathsf{Client}_\beta.(c[\mathsf{Num}_2]\ \mathsf{pfunctions}_2) \end{array}$$

Note that nlib_1 and nlib_2 must be parameterized by β, which is the return type of the client using the library. Given nlib_1 or nlib_2, we can link the client program with

$$(\mathsf{nlib}_1[\texttt{bool}]\ \mathsf{client})$$

or

$$(\mathsf{nlib}_2[\texttt{bool}]\ \mathsf{client})$$

The only difference from before is that, given only $\mathsf{pfunction}_i$ instead of nfuncs_i, the linking programmer cannot abuse the library by linking it to a client that exploits the concrete type of numbers. In other words, the type on the c argument of $\mathsf{pfunction}_i$ forces clients to respect the data abstraction.

17.3 Existential ISWIM

The nlib_1 and nlib_2 functions each consume a polymorphic function as an argument. Although this is not a problem in principle, we saw in Section 15.2 that polymorphic values introduce problems for type inference, and are thus disallowed in some languages. We can support module-like packages without polymorphic values by adjusting our language to turn the abstraction-enforcing function "inside out."

The idea is to capture just the core part of nlib_1 and nlib_2. The type of both expressions is $\forall\beta.(\forall\alpha.(\mathsf{NFuncs}_\alpha \to \beta) \to \beta)$. The β parts really have nothing to do with the library, only the clients; mentioning those polymorphic clients requires polymorphic values, which is the part we want to eliminate. To pull out the NFuncs_α path, we need a way to lift the function arrow out of the $\forall\alpha...$ type. To support such lifting, we will enrich our language with **existential types**.

Using our new extensions, the type of the library expression will be

$$\exists\alpha.\mathsf{NFuncs}_\alpha$$

which means "a tuple of three functions involving α, for some hidden type α". Just as $\forall$ types require a creation form ($\Lambda A.M$) and an application form $M[T]$, $\exists$ types require a creation and use form. The creation form is `pack`:

$$(\texttt{pack}[A{=}T]\ M\ \texttt{as}\ T')$$

where M generates the value whose reprsentation is to be partially hidden. The variable A will represent the hidden type, T is the actual hidden type, and T' is the type of M where some instances of T have been replaced by A. The type of the overall `pack` expression will be $\exists A.T'$.

For example, we can use `pack` to create our two implementations of numbers as follows:

$$\begin{array}{lcl} \mathsf{npack}_1 & \doteq & (\texttt{pack}[\alpha{=}\mathsf{Num}_1]\ \mathsf{nfuncs}_1\ \texttt{as}\ \mathsf{NFuncs}_\alpha) \\ \mathsf{npack}_2 & \doteq & (\texttt{pack}[\alpha{=}\mathsf{Num}_2]\ \mathsf{nfuncs}_2\ \texttt{as}\ \mathsf{NFuncs}_\alpha) \end{array}$$

The type of both expressions is the existiential type we wanted, $\exists\alpha.\mathsf{NFuncs}_\alpha$, which in expanded form is $\exists\alpha.((\texttt{num} \to \alpha) \times ((\alpha \to (\alpha \to \alpha)) \times (\alpha \to \texttt{bool})))$.

The `unpack` form uses an abstraction created by `pack`. While unpacking, it does not expose the implementation type hidden by `pack`, but instead declares a local type name to stand for the hidden type, much like the Λ-bound variable stood for the hidden type in the old encoding:

$$(\texttt{unpack}[A]\ X\ \texttt{from}\ M\ \texttt{in}\ M')$$

where M' produces a packed value, the variable A stands for the hidden type locally, the variable X recieves the unpacked value produced by M, and the body M' uses the unpacked value polymorphically in A.

For example, to use npack_1, we select a variable γ to stand in place of the unknown type in client code:

$$\begin{array}{l}(\texttt{unpack}[\gamma]\ n\ \texttt{from}\ \mathsf{npack}_1\ \texttt{in}\\ \quad(\mathsf{let}\ m{:}(\texttt{num} \to \gamma) = (\texttt{fst}\ n)\\ \qquad\quad a{:}(\gamma \to (\gamma \to \gamma)) = (\texttt{fst}\ (\texttt{snd}\ n))\\ \qquad\quad p{:}(\gamma \to \texttt{bool}) = (\texttt{snd}\ (\texttt{snd}\ n))\\ \qquad\mathsf{in}\ (p\ (a\ (m\ 17)\ (m\ 18)))))\end{array}$$

Since npack_1 has type $\exists\alpha.((\texttt{num} \to \alpha) \times ((\alpha \to (\alpha \to \alpha)) \times (\alpha \to \texttt{bool})))$, the $\texttt{unpack}[\gamma]$ expression causes n to have the type $((\texttt{num} \to \gamma) \times ((\gamma \to (\gamma \to \gamma)) \times (\gamma \to \texttt{bool})))$, which is consistent with the let bindings. The overall program therefore has type `bool`.

Note that the body of the `unpack` expression cannot exploit the concrete representation of numbers hidden by `pack`, because the type is reduced to simply γ. Furthermore, if we try to mix functions from the two packages, as in the following expression, the resulting program has no type, because $\gamma_1 \neq \gamma_2$ (i.e., no two type variables are ever known to be equal):

$$\begin{array}{l}(\texttt{unpack}[\gamma_1]\ n_1\ \texttt{from}\ \mathsf{nlib}_1\ \texttt{in}\\ \quad(\texttt{unpack}[\gamma_2]\ n_2\ \texttt{from}\ \mathsf{nlib}_2\ \texttt{in}\\ \qquad(\mathsf{let}\ m_1{:}(\texttt{num} \to \gamma_1) = (\texttt{fst}\ n_1)\\ \qquad\quad a_1{:}(\gamma_1 \to (\gamma_1 \to \gamma_1)) = (\texttt{fst}\ (\texttt{snd}\ n_1))\\ \qquad\quad p_2{:}(\gamma_2 \to \texttt{bool}) = (\texttt{snd}\ (\texttt{snd}\ n_2))\\ \qquad\mathsf{in}\ (p_2\ (a_1\ (m_1\ 17)\ (m_1\ 18))))))\end{array}$$

Using γ_1 for both `unpack` expressions would not allow the packages to be confused, either. The γ_1s would be different in the same sense that the xs in $(\lambda x.\lambda x.x)$ are different.

Hiding implies that a value of type $\exists\alpha.(\alpha \to \texttt{bool})$ is by itself useless; since the type α is unknown, the function cannot be applied to any value. In contrast, a value of type $\exists\alpha.((\texttt{num} \to \alpha) \times (\alpha \to \texttt{bool}))$ can be useful. The first function in the pair can construct values of type α, and the second function in the pair can consume those values. The programmer using such a value need not know anything about the implementation of intermediate value of type α.

To add existential types to ISWIM to form **Existential ISWIM**, we extend the expression grammar with `pack` and `unpack` forms, where the `pack` form generates values.

$$\begin{array}{rcl@{\qquad}rcl}
M & = & \ldots & T & = & \ldots\\
 & | & (\texttt{pack}[A{=}T]\ M\ \texttt{as}\ T) & & | & (\exists A.T)\\
 & | & (\texttt{unpack}[A]\ X\ \texttt{from}\ M\ \texttt{in}\ M) & & &
\end{array}$$

$$\begin{array}{rcl@{\qquad}rcl}
V & = & \ldots & E & = & \ldots\\
 & | & (\texttt{pack}[A{=}T]\ V\ \texttt{as}\ T) & & | & (\texttt{pack}[A{=}T]\ E\ \texttt{as}\ T)\\
 & & & & | & (\texttt{unpack}[A]\ X\ \texttt{from}\ E\ \texttt{in}\ M)
\end{array}$$

$$(\texttt{unpack}[A]\ X\ \texttt{from}\ (\texttt{pack}[A'{=}T']\ V\ \texttt{as}\ T)\ \texttt{in}\ M) \quad \mathbf{x} \quad M[X \leftarrow V][A \leftarrow T']$$

$$\mathbf{v} = \ldots \cup \mathbf{x}$$

The new type-checking rules assign an existential type to each `pack` expressions, and require an existential type for the target of each `unpack` expressions:

$$\frac{\Gamma \vdash M : T[A \leftarrow T'] \quad \Gamma \vdash T' \quad \Gamma, A \vdash T}{\Gamma \vdash (\texttt{pack}[A = T']\ M\ \texttt{as}\ T) : \exists A.T}$$

$$\frac{\Gamma \vdash M_1 : \exists A'.T' \quad \Gamma, X{:}T'[A' \leftarrow A] \vdash M_2 : T}{\Gamma \vdash (\texttt{unpack}[A]\ X\ \texttt{from}\ M_1\ \texttt{in}\ M_2) : T} \quad \text{where } A \notin \mathcal{FV}(T)$$

The constraint $A \notin \mathcal{FV}(T)$ is needed with the `unpack` rule to ensure that the introduced variable A does not escape its scope. Operationally, this constraint implies that a value of an unknown type for particular `unpack` expression cannot escape the dynamic extent of the of the `unpack` expression (where it would be useless, anyway).

▷ **Exercise 17.1.** Show that

$$\begin{array}{l}(\texttt{unpack}[\beta]\ n\ \texttt{from}\ \mathsf{npack}_1\ \texttt{in}\\ \quad (\mathsf{let}\ m{:}(\texttt{num} \to \beta) = (\texttt{fst}\ n)\\ \qquad a{:}(\beta \to (\beta \to \beta)) = (\texttt{fst}\ (\texttt{snd}\ n))\\ \qquad p{:}(\beta \to \texttt{bool}) = (\texttt{snd}\ (\texttt{snd}\ n))\\ \quad \mathsf{in}\ (p\ (a\ (m\ 17)\ (m\ 18)))))\end{array}$$

has a type and evaluates to a value in Existential ISWIM.

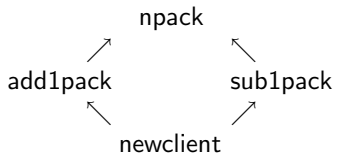
17.4 Modules and Existential Types

Suppose we want to create an extension of npack (either npack_1 or npack_2) that adds an add-1 operation. Our first attempt at this extension might be similar to the earlier client that uses npack, except that we re-pack the unpacked tuple, adding a new function onto the front:

$$\begin{array}{rcl}\mathsf{add1pack} & \doteq & (\texttt{unpack}[\beta]\ n\ \texttt{from}\ \mathsf{npack}_1\ \texttt{in}\\ & & \quad (\mathsf{let}\ m{:}(\texttt{num} \to \beta) = (\texttt{fst}\ n)\\ & & \qquad a{:}(\beta \to (\beta \to \beta)) = (\texttt{fst}\ (\texttt{snd}\ n))\\ & & \quad \mathsf{in}\ (\mathsf{let} f{:}(\beta \to \beta) = \lambda x{:}\beta.(a\ x\ (m\ \ulcorner 1 \urcorner))\\ & & \qquad \mathsf{in}(\texttt{pack}[\alpha{=}\beta]\ \langle f, n\rangle\ \texttt{as}\ ((\alpha \to \alpha) \times \mathsf{NFuncs}_\alpha)))))\end{array}$$

Client code can use add1pack in essentially the same way as npack. Of course, we can imagine other extensions, such as a sub1pack module that adds a subtract-1 operation. The implementation of sub1pack is the same as add1pack, but with $\ulcorner -1 \urcorner$ instead of $\ulcorner 1 \urcorner$.

What happens if a client wants to use both sub1pack and add1pack? Conceptually, the program's achitecture is



This program achitecture is known as **diamond import**, and it demonstrates a problem with our encoding of modules. When newclient unpacks add1pack, it obtains one type of numbers

γ_1, and when newclient unpacks add1pack, it obtains another type of numbers γ_2:

$$\begin{array}{l}(\texttt{unpack}[\gamma_1]\ n_1\ \texttt{from}\ \mathsf{add1pack}_1\ \texttt{in}\\ \quad(\texttt{unpack}[\gamma_2]\ n_2\ \texttt{from}\ \mathsf{sub1pack}_2\ \texttt{in}\\ \quad\quad(\mathsf{let}\ m_1{:}(\texttt{num}\to\gamma_1)\ \texttt{=}\ (\texttt{fst}\ (\texttt{fst}\ n_1))\\ \quad\quad\quad\ a_1{:}(\gamma_1\to\gamma_1)\ \texttt{=}\ (\texttt{fst}\ n_1)\\ \quad\quad\quad\ s_2{:}(\gamma_2\to\gamma_2)\ \texttt{=}\ (\texttt{fst}\ n_2)\\ \quad\quad\ \mathsf{in}\ (s_2\ (a_1\ (m_1\ \ulcorner 0\urcorner))))))\end{array}$$

This expression has no type, because the result of a call to the add-1 function cannot be provided to the sub-1 function; the types are not compatible. Actually, the representations are compatible, and the evaluation of $(s_2\ (a_1\ (m_1\ \ulcorner 0\urcorner)))$ would not get stuck, but the type system is enforcing too many abstraction boundaries.

We can adjust our encoding of modules to solve this problem. The source of the problem is that add1pack and sub1pack each separately unpack the original npack. To avoid this unpacking, we can return to our earlier use of explicit Λ for "clients", though in this case the two clients serve as libraries to other clients. Another part of the problem is that both add1pack and sub1pack re-pack the content of the original npack. In our revised encoding, we will assume instead that clients of add1pack and sub1pack explicitly import npack to get the base funtionality.

Putting the pieces together, our new definition follows:

$$\begin{array}{rcl}\mathsf{add1module} & \doteq & \Lambda\beta.\lambda n{:}\mathsf{NFuncs}_\beta.\\ & & \quad(\mathsf{let}\ m{:}(\texttt{num}\to\beta)\ \texttt{=}\ (\texttt{fst}\ n)\\ & & \quad\quad\ a{:}(\beta\to(\beta\to\beta))\ \texttt{=}\ (\texttt{fst}\ (\texttt{snd}\ n))\\ & & \quad\ \mathsf{in}\ (\mathsf{let}f{:}(\beta\to\beta)\ \texttt{=}\ \lambda x{:}\beta.(a\ x\ (m\ \ulcorner 1\urcorner))\\ & & \quad\quad\ \mathsf{in}\ f)))\end{array}$$

The definition of sub1module is similar. The body of the module is simply f because add1module and sub1module do not introduce any new abstract types. The body of a module that does introduce a new abstract type would have the form $(\texttt{pack}[\alpha{=}T]\ M\ \texttt{as}\ (\beta\to\beta))$ for some hidden type T.

In general, the type of a module will have the shape

$$\forall\beta\ldots\lambda x{:}T_{\beta\ldots}.\exists\alpha\ldots T_{\alpha\ldots,\beta\ldots}$$

where a sequence of βs represents the imported abstract types, the type $T_{\beta\ldots}$ corresponds to module imports whose types mention the βs, a sequence of αs represents exported abstract types, and the type $T_{\alpha\ldots,\beta\ldots}$ corresponds to a set of exports whose types mention the αs and βs.

To use add1module and sub1module, a client must link them together with npack:

$$\begin{array}{l}(\texttt{unpack}[\gamma]\ n\ \texttt{from}\ \mathsf{npack}_1\ \texttt{in}\\ \quad(\mathsf{let}\ a\ \texttt{=}\ (\mathsf{add1module}[\gamma]\ n)\\ \quad\ \mathsf{in}\ \ (\mathsf{let}\ s\ \texttt{=}\ (\mathsf{sub1module}[\gamma]\ n)\\ \quad\quad\ \mathsf{in}\ (\mathsf{let}\ m{:}(\texttt{num}\to\gamma_1)\ \texttt{=}\ (\texttt{fst}\ n)\\ \quad\quad\quad\ \mathsf{in}\ (s_2\ (a_1\ (m_1\ \ulcorner 0\urcorner)))))))\end{array}$$

The main drawback of this approach is readily apparent. As the number of modules that comprise a program grows, and as modules become more interconnected, the client programmer is responsible for an increasingly complex linking step to assemble the program. In practice, this problem can be tempered by staging the linking steps; linking must be delayed only where diamond imports occur, but completed diamonds can be sealed. But when a library module is used through a large program, it tends to generate diamonds that are resolved only in the final linking steps.

Chapter 18: Subtyping

In all of our typed languages so far, the type of a procedure expression fully determines the type of the corresponding argument expression in an application. For example, even with type inference, if M has type $(T_1 \to T_2)$ in $(M\ N)$, then N must have exactly the type T_1, modulo the renaming of variables in T_1 bound by $\forall$, μ, or $\exists$.

A language with **subtyping** provides more flexibility: the type of N might be exactly T_1, or it might be a **subtype** T_1' of T_1. The notion of subtype is defined so that a value whose type is T_1' can replace a value of type T_1 without introducing a stuck state.

So far, none of our ISWIM variants have provided an opportunity for subtyping: intuitively, a value of type `bool` cannot be used in place of a `num` (because an addition might get stuck), a $(\texttt{num} \to \texttt{num})$ cannot be used in place of a $\forall\alpha.(\alpha \to \alpha)$ (because a type application might get stuck), and a value of type $(\texttt{num} \times \texttt{bool})$ cannot be used in place of a $(\texttt{num} \times \texttt{num})$ (because the second part of the pair might be used).

A small addition to our language, however, introduces a natural place for subtyping. We consider the addition of records: tuple values with arbitrary arity, where the fields of the tuple are labelled. Subtyping naturally arises in the form of records that have structly more fields than other others.

18.1 Records and Subtypes

To begin the definition of Subtype ISWIM, we add a record expression form. We also add an expression form for extracting the value of a labelled field from a record:

$$
\begin{array}{rcl@{\qquad}rcl}
M & = & \ldots & E & = & \ldots \\
 & | & \langle L{=}M, \ldots\ L{=}M\rangle & & | & \langle L{=}V, \ldots\ L{=}V, L{=}E, L{=}M, \ldots\ L{=}M\rangle \\
 & | & M.L & & & \\
 & & & & & \\
V & = & \ldots & L & = & \text{a label: } \mathit{size}, \mathit{color}, \ldots \\
 & | & \langle L{=}V, \ldots\ L{=}V\rangle & & & \\
 & & & & & \\
 & & \langle L_0{=}V_0, \ldots\ L_n{=}V_n\rangle.L_i \quad \mathbf{l} \quad V_i & & & \text{for } i \in [0, n] \\
 & & & & & \\
 & & \mathbf{v} = \ldots \cup \mathbf{l} & & &
\end{array}
$$

For example, we can use a labelled tuple to represent a 10-inch fish as

$$\mathsf{fish} \doteq \langle \mathit{size}{=}\ulcorner 10 \urcorner \rangle$$

or a red, 10-inch fish as

$$\mathsf{colorfish} \doteq \langle \mathit{size}{=}\ulcorner 10 \urcorner, \mathit{color}{=}\mathsf{red} \rangle \qquad \mathsf{red} \doteq \ulcorner 3 \urcorner$$

If M produces a fish of either sort, we can obtain the fish's size with $M.\mathit{size}$.

The type of a record must list the field names of the tuple, and also provide a type for each field. A field-selection expression requires a record type to the left of the selection dot where

the field to the right of the dot is included in the record:

$$\begin{array}{rcl} T & = & \ldots \\ & | & \langle L{:}T, \ldots\ L{:}T\rangle \end{array}$$

$$\frac{\Gamma \vdash M_0 : T_0 \quad \ldots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash \langle L_0{=}M_0, \ldots\ L_n{=}M_n\rangle : \langle L_0{:}T_0, \ldots\ L_n{:}T_n\rangle}$$

$$\frac{\Gamma \vdash M : \langle L_0{:}T_0, \ldots\ L_n{:}T_n\rangle}{\Gamma \vdash M.L_i : T_i} \quad \text{where } i \in [0, n]$$

For example, the type of fish is $\langle \mathit{size}{:}\mathtt{num}\rangle$ and the type of colorfish is $\langle \mathit{size}{:}\mathtt{num}, \mathit{color}{:}\mathtt{num}\rangle$.

Using a record type for a function argument, we can define a function that takes a fish and returns its size:

$$\mathsf{fsize} \doteq \lambda f{:}\langle \mathit{size}{:}\mathtt{num}\rangle.\, f.\mathit{size}$$

This function has type $(\langle \mathit{size}{:}\mathtt{num}\rangle \to \mathtt{num})$. Naturally, the expression (fsize fish) has type num, and it produces the value $\ulcorner 10 \urcorner$ when evaluated.

What about (fsize colorfish)? When evaluated, it also produces $\ulcorner 10 \urcorner$. But this expression has no type, because fsize consumes an argument of type $\langle \mathit{size}{:}\mathtt{num}\rangle$, while colorfish has type $\langle \mathit{size}{:}\mathtt{num}, \mathit{color}{:}\mathtt{num}\rangle$. Clearly, we want to make $\langle \mathit{size}{:}\mathtt{num}, \mathit{color}{:}\mathtt{num}\rangle$ a subtype of $\langle \mathit{size}{:}\mathtt{num}\rangle$, because (intuitively) replacing fish with colorfish can never cause a program to become stuck.

In general, any record type that has at least the fields of another record type should be a subtype. To define subtypes formally, we introduce a $\leq$ relation, where $T \leq T'$ means "T is a subtype of T'." Initially, we define the relation as follows:

$$T \ \leq\ T$$

$$\langle L_0{:}T_0, \ldots\ L_n{:}T_n\rangle \ \leq\ \langle L'_0{:}T'_0, \ldots\ L'_m{:}T'_m\rangle \quad \text{if } \{L_0{:}T_0, \ldots\ L_n{:}T_n\} \supseteq \{L'_0{:}T'_0, \ldots\ L'_m{:}T'_m\}$$

The notation $\leq$ suggests the following intuition about the meaning of subtyping: the set of values belonging to a subtype is smaller than the set of values belonging to the original type, because the latter includes the former.

We also adjust the rule for function application to allow subtypes in the argument position:

$$\frac{\Gamma \vdash M : (T' \to T) \quad \Gamma \vdash N : T'' \quad T'' \leq T'}{\Gamma \vdash (M\ N) : T}$$

(To correctly handle polymorphism and recursive types, we could adjust our subtyping rule to absorb the $\vdash \_ \leftrightarrow \_$ relation, but we ignore the extra complexity.)

Using the new rules, we can show a type for (fsize colorfish), abbreviating num with n, *size* with s, and *color* with c:

$$\frac{\dfrac{\dfrac{f{:}\langle s{:}\mathtt{n}\rangle \vdash f : \langle s{:}\mathtt{n}\rangle}{f{:}\langle s{:}\mathtt{n}\rangle \vdash f.s : \mathtt{n}}}{\vdash (\lambda f{:}\langle s{:}\mathtt{n}\rangle.\, f.s) : (\langle s{:}\mathtt{n}\rangle \to \mathtt{n})} \quad \vdash \langle s{=}\ulcorner 10 \urcorner, c{=}\mathsf{red}\rangle : \langle s{:}\mathtt{n}, c{:}\mathtt{n}\rangle \quad \langle s{:}\mathtt{n}, c{:}\mathtt{n}\rangle \leq \langle s{:}\mathtt{n}\rangle}{\vdash ((\lambda f{:}\langle s{:}\mathtt{n}\rangle.\, f.s)\ \langle s{=}\ulcorner 10 \urcorner, c{=}\mathsf{red}\rangle) : \mathtt{n}}$$

▷ **Exercise 18.1.** Does

$$(\mathtt{if}\ M\ \mathsf{fish}\ \mathsf{colorfish}).\mathit{size}$$

produce a value when evaluated, assuming that M produces either t or f? Does it have a type using our rules so far, assuming $\vdash M : \mathtt{bool}$? If not, should it? If not, and if it should, suggest a sound revision to a type rule so that the expression has a type.

18.2 Subtypes and Functions

Our revised type rules not only give (fsize colorfish) a type, but we can apply any function on fishes to color fishes. Indeed, we can write a function that takes an arbitrary function on fishes and applies it to colorfish:

$$\mathsf{apply} \doteq \lambda f{:}(\langle \mathit{size}{:}\mathtt{num}\rangle \to \mathtt{num}).(f\ \mathsf{colorfish})$$

We can easily verify that (apply fsize) has a type. Suppose that we have a function that extracts a fish's color:

$$\mathsf{cfcolor} \doteq \lambda f{:}\langle \mathit{size}{:}\mathtt{num}, \mathit{color}{:}\mathtt{num}\rangle.f.\mathit{color}$$

Can we use cfcolor as an argument to apply? According to the type system, the answer is "no", even though (apply cfcolor) produces a value.

In this case, the type system's restriction is justified. The body of apply would be well-typed if the argument of f is changed to fish instead of colorfish. If we made that change and then applied apply to cfcolor, evaluation would get stuck.

If we really want to apply functions to colorfish, then we should declare a different type on the argument of apply:

$$\mathsf{capply} \doteq \lambda f{:}(\langle \mathit{size}{:}\mathtt{num}, \mathit{color}{:}\mathtt{num}\rangle \to \mathtt{num}).(f\ \mathsf{colorfish})$$

Again, we can easily verify that (capply cfcolor) has a type. But now (capply fsize) has no type, although it still produces a value.

In this case, the type system is too restrictive. Since every function on fishes works on color fishes, capply should accept both kinds of functions. There is no one type we can write for capply's argument, but we can extend our subtyping relation. Again, every function on fishes works on color fishes, which means that the type of a function on fishes should be a subtype of a function on colorfishes:

$$(\langle \mathit{size}{:}\mathtt{num}\rangle \to \mathtt{num}) \ \le\ (\langle \mathit{size}{:}\mathtt{num}, \mathit{color}{:}\mathtt{num}\rangle \to \mathtt{num})$$

At first glance, this relation may look backward, but it is not. Intuitively, a color fish is a special kind of fish, but a function on fishes is a special kind of function on color fishes. Put another way, the set of fishes is larger than the set of color fishes (because the former includes the latter), and the set of functions that work on color fishes is larger than the set of functions that work on fishes (because the former includes the latter).

In general, a subtype relation for the argument part of a function type induces the reverse relation on the function type as a whole. A position in type expression with this property is called a **contra-variant** position.

Before we define the general subtyping rule on function types, consider the result position. Suppose a function returns a color fish; can it be used in place of a function that returns a fish? Yes, just as a color fish can be used in place of a fish. This less surprising kind of position is called **co-variant**: a subtype relation in the result position of a function type induces the same subtype relation in the function type as a whole.

Thus, the general rule for function subtyping is as follows:

$$\frac{T_1' \le T_1 \quad T_2 \le T_2'}{(T_1 \to T_2) \le (T_1' \to T_2')}$$

Using this new rule, we can show a type for (capply fsize):

$$\dfrac{\dfrac{\cdots}{\vdash \mathsf{capply} : ((\langle s{:}\mathtt{n}, c{:}\mathtt{n}\rangle \to \mathtt{n}) \to \mathtt{n})} \quad \dfrac{\cdots}{\vdash \mathsf{fsize} : (\langle s{:}\mathtt{n}\rangle \to \mathtt{n})} \quad \dfrac{\langle s{:}\mathtt{n}, c{:}\mathtt{n}\rangle \le \langle s{:}\mathtt{n}\rangle \quad \mathtt{n} \le \mathtt{n}}{(\langle s{:}\mathtt{n}\rangle \to \mathtt{n}) \le (\langle s{:}\mathtt{n}, c{:}\mathtt{n}\rangle \to \mathtt{n})}}{\vdash (\mathsf{capply}\ \mathsf{fsize}) : \mathtt{n}}$$

18.3 Subtypes and Fields

Suppose that we define a type for a two-fish school:

$$\mathsf{School} \doteq \langle \mathit{one}\!:\!\langle \mathit{size}\!:\!\mathtt{num}\rangle, \mathit{two}\!:\!\langle \mathit{size}\!:\!\mathtt{num}\rangle\rangle$$

Should we be able to use a school composing two color fish as a school of plain fish?

$$\mathsf{cschool} \doteq \langle \mathit{one}{=}\langle \mathit{size}\!:\!1, \mathit{color}\!:\!\mathsf{red}\rangle, \mathit{two}{=}\langle \mathit{size}\!:\!1, \mathit{color}\!:\!\mathsf{blue}\rangle\rangle$$

Yes. A school of color fish clearly works wherever a school of fish works, because the only possible operation on a school is to extract its fish, and the fish are individually replaceable with color fish.

More generally, we can allow subtypes on fields within records co-variantly.

$$\begin{array}{l}\langle L_0\!:\!T_0, \ldots\ L_n\!:\!T_n\rangle \ \le\ \langle L'_0\!:\!T'_0, \ldots\ L'_m\!:\!T'_m\rangle \\ \quad \text{if } \{L_0, \ldots\ L_n\} \supseteq \{L'_0, \ldots\ L'_m\} \\ \quad \text{and } (L_i = L'_j) \Rightarrow (T'_i \le T_j) \text{ for } i \in [0,n], j \in [0,m]\end{array}$$

As it turns out, few realistic programming languages support this kind of subtyping. The reason is that such languages tend to provide an additional operation on records: field update. When field update is allowed along with subtyping on record fields, a field might get replaced with a *super*type value. For example, a color school might be used as a plain school, and then the *one* field of the school could be updated with a fish (not a color fish). Other parts of the program that expect a color school will then find a plain fish in the school, and probably get stuck as a result.

To support update operations, record fields in a typical language are **invariant**, which means that they are neither co-variant nor contra-variant. In particular fields of an object in Java are invariant, as we will see in the next chapter.

18.4 From Records to Objects

Objects in an object-oriented language are similar to records. In addition to field values, however, an object encapsulates a set of **methods**. In a language like ISWIM with procedure values, we can encode methods as procedure-valued fields in a record. A procedure representing a method must consume the "self" object as its argument.

Thus, a fish object with a *size* field and *getsize* method would be encoded as

$$\mathsf{fishobj} \doteq \langle \mathit{size}{=}\ulcorner 10\urcorner, \mathit{getsize}{=}\lambda s{:}\langle \mathit{size}\!:\!\mathtt{num}\rangle.s.\mathit{size}\rangle$$

To call the *getsize* "method" of fishobj, we extract the *getsize* procedure from fishobj and apply it to fishobj:

$$(\mathsf{fishobj}.\mathit{getsize}\ \mathsf{fishobj})$$

This expression has a type and produces the expected result.

A more realistic encoding of objects requires a more precise type for the argument to *getsize*. In particular, methods must be mentioned in the type of s so that the body of the method can call the object's own methods—including the *getsize* method. Such a refinement is possible with recursive types, but we ignore this generalization, because we encounter a problem even with the simple encoding.

To see the problem, consider a color fish with a *getsize* method where the fish's color multiplies its apparent size:

$$\mathsf{colorfishobj} \doteq \langle \mathit{size}{=}\ulcorner 10\urcorner, \mathit{color}{=}\ulcorner 2\urcorner, \mathit{getsize}{=}\lambda s{:}\langle \mathit{size}\!:\!\mathtt{num}, \mathit{color}\!:\!\mathtt{num}\rangle.(* \ s.\mathit{size}\ s.\mathit{color})\rangle$$

Again, the expression for calling *getsize* of colorfishobj has a type and value:

(colorfishobj.*getsize* colorfishobj)

But is the type of colorfishobj a subtype of the type for fishobj?

Even if we allow co-variant record fields, it is not, because the type of colorfishobj's *getsize* method is not a subtype of the type for fishobj's *getsize* method (due to the contra-variance of function arguments). Indeed, we can see that an expression such as

(colorfishobj.*getsize* fishobj)

should not have a type, because it will get stuck; fishobj will not have the *color* field that *getsize* wants.

The final example not only shows that the encoding should fail, but it shows why: a programmer can construct bad "method calls" where the wrong self object is supplied to the method. Following our earlier strategy for addressing type problems, we can solve this one by treating methods differently than fields, and by not allowing method-as-field accesses except those that fit the right pattern. We explore this path in the next chapter.

Chapter 19: Objects and Classes

In an untyped setting, we can easily model classes and objects by using λ to construct methods and pairs or records to construct objects. However, as we saw in the previous chapter, such an encoding of objects does not work well with the type system of Simply Typed ISWIM or its variants. In the same way that adding pairs to Simply Typed ISWIM was the simplest way of obtaining well-typed pairs, adding classes and objects to the language—or starting with an entirely new language based on classes—is the simplest and usually most useful way of obtaining well-typed objects.

In this chapter, we abandon ISWIM and model directly an object-oriented language. In particular, we model a subset of the Java™ language. Java itself is far too large to model formally here. We choose a subset that contains the programming elements we are most interested in, which in this case include classes, objects, fields, methods, inheritance, overriding, and imperative field update. We ignore interfaces, abstract methods, exceptions, and arrays.

19.1 MiniJava Syntax

MiniJava is a simplification of Java for studying classes and mutable objects. A MiniJava program is a sequence of class declarations followed by an expression to evaluate (instead of relying on a `main` method). Each class contains a sequence of field declarations (with a type and constant initial value for each field) followed by a sequence of method declarations.

P	=	$\hat{c} \ldots\ \hat{c}\ M$	a program
$\hat{c}$	=	`class` c `extends` c { $\hat{f} \ldots\ \hat{f}\ \hat{m} \ldots\ \hat{m}$ }	class declaration
$\hat{f}$	=	$T\ f = V$	field declaration
$\hat{m}$	=	$T\ m(T\ X, \ldots T\ X)\{\ M\ \}$	method declaration
c	=	a class name or `Object`	
f	=	a field name	
m	=	a method name	
X	=	a variable name or `this`	

The only expression forms are field access, field update, method calls, super calls, and casts.

M	=	X	
	|	`null`	
	|	`new` c	
	|	M:$c.f$	field access
	|	M:$c.f := M$	field assignment
	|	$M.m(M, \ldots M)$	method call
	|	`super` X:$c.m(M, \ldots M)$	super call
	|	$(c)M$	cast
	|	b	
	|	$(o^n\ M \ldots\ M)$	
T	=	c	
	|	B	
b	=	a basic constant	
o^n	=	an n-ary primitive operation	
B	=	a basic type	

In the case of field accesses and updates, the field name is annotated with its source class, so that we can ignore the problem of field hiding in derived classes. Similarly, for super calls, the target object as a variable (usually `this`) and the superclass are named explicitly. Instead of modeling object constructors, every field in an object is initialized to a given constant, which must be `null` for class-typed fields. As usual, we assume an unspecified set of basic constants b, primitive operations o^n, and basic types B. The set of types T includes basic types and class names.

The following example program defines a *fish* class and a derived *colorfish* class, and then instantiates both classes:

```
class fish extends Object {
    num size = ⌜1⌝
    num getWeight(){ this:fish.size }
    num grow(num a){ this:fish.size := (+ a this:fish.size) }
    num eat(fish f){ this.grow(f.getWeight()) }
}
class colorfish extends fish {
    num color = ⌜7⌝
    num getWeight(){ (* super this:fish.getWeight() this:colorfish.color) }
}
new fish.eat(new colorfish)
```

The *fish* class contains the field *size* and the methods *getWeight*, *grow*, and *eat*. The *getWeight* method returns the value of the *size* field in `this`. The *grow* method increments the *size* field of `this`, and the *eat* method calls *grow* with the weight of the given fish. The *colorfish* class adds a *color* field, and overrides the *getWeight* method. The overriding declaration chains to *fish*'s *getWeight* and multiplies the result by the fish's color.

A well-formed MiniJava program must obey certain high-level constraints: every class name must be declared only once, the built-in class `Object` must not be declared, the fields of each class must be unique, the methods of each class must be unique, and every class that is used must be declared. These properties are most easily captured by relations defined with "is in", which relates arbitrary symbol sequences, and with "$\cdots$", which stands for an arbitrary sequence of symbols with balanced braces and parentheses.

ClassesOnce(P) **iff**
($\texttt{class}\ c\ \cdots\ \texttt{class}\ c'$ is in P) implies $c \neq c'$

ObjectBuiltin(P) **iff**
`class Object` is not in P

FieldsOncePerClass(P) **iff**
($\texttt{class}\ c\ \texttt{extends}\ c'\ \{\cdots\ T_1\ f_1\ \cdots\ T_1\ f_2\ \cdots\}$ is in P) implies $f_1 \neq f_2$

MethodsOncePerClass(P) **iff**
($\texttt{class}\ c\ \texttt{extends}\ c'\ \{\cdots\ T_1\ m_1\ \cdots\ T_1\ m_2\ \cdots\}$ is in P) implies $m_1 \neq m_2$

ClassesDefined(P) **iff**
(c is in P) implies $c = \texttt{Object}$ or ($\texttt{class}\ c$ is in P)

By inspecting the *fish* example, we can easily see that all of these relations hold for the example program.

Besides these basic checks, the declared class must form an inheritance tree (as opposed to a cyclic inheritance graph!), overriding method declarations must be consistent with superclass method definitions, and so on.

We can simplify the checking of such properties by defining a few more derived relations

with "is in". For example, $\prec_P$ relates a class and its declared superclass in P.

$c \prec_P c'$ **iff**
  `class` c `extends` c' is in P
$\langle c.f, T, V\rangle \Subset^{\mathsf{f}}_P c$ **iff**
  `class` c `extends` c' $\{\cdots\ T\ f = V\ \cdots\}$ is in P
$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c$ **iff**
  `class` c `extends` c' $\{\cdots\ T_0\ m(T_1\ X_1, \ldots T_n\ X_n)\{\ M\ \}\ \cdots\}$ is in P

In the earlier example, *fish* $\prec_P$ `Object`, *colorfish* $\prec_P$ *fish*, $\langle$*colorfish.color*, `num`, $\ulcorner 7 \urcorner\rangle \Subset^{\mathsf{f}}_P$ *colorfish*, and $\langle$*grow*, (`num` $\to$ `num`), (*a*), (+ *a* `this`:*fish.size*)$\rangle \Subset^{\mathsf{m}}_P$ *fish*.

The transitive closure of the $\prec_P$ relation is the complete subclass relation $\leq_P$, which must be antisymmetric so that inheritance is well-defined (i.e., the subclass relationship among classes must form a tree):

$\leq_P$ = transitive–reflexive closure of $\prec_P$
WellFoundedClasses(P) **iff** $\leq_P$ is antisymmetirc

Using the $\leq_P$ relation, we can define the final field-inclusion relation, where a class inherits the fields of its superclass, and also check that method overriding preserves the type of a method.

$\langle c.f, T, V\rangle \in^{\mathsf{f}}_P c'$ **iff**
  $\langle c.f, T, V\rangle \Subset^{\mathsf{f}}_P c$ and $c' \leq_P c$
OverridesConsistent(P) **iff**
  ($\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c_1$)
  and ($\langle m, (T'_1 \ldots\ T'_n \to T'_0), (X'_1, \ldots X'_n), M'\rangle \Subset^{\mathsf{m}}_P c_2$)
  implies ($c_1 \not\leq_P c_2$ or $(T_1 \ldots\ T_n \to T_0) = (T'_1 \ldots\ T'_n \to T'_0)$)

Thus, $\langle$*fish.size*, `num`, $\ulcorner 1 \urcorner\rangle \in^{\mathsf{f}}_P$ *fish* and $\langle$*fish.size*, `num`, $\ulcorner 1 \urcorner\rangle \in^{\mathsf{f}}_P$ *colorfish*.

For the purposes of type checking, we could define a method-inclusion relation like the one for fields:

$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c'$ **iff**
  $\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c$ and $c' \leq_P c$

However, in the case that a method is overriden, this definition assigns multiple method implementations to a given class for the method name. A more useful relation would pick the most overriding implementation of the method, i.e., the one from the *minimum* class (according to the $\leq_P$ relation) that defines the method.

$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c'$ **iff**
  $\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c$
  and $c = \min\{c \mid c' \leq_P c$ and $\langle m, (T'_1 \ldots\ T'_n \to T'_0), (X'_1, \ldots X'_n), M'\rangle \Subset^{\mathsf{m}}_P c\}$

For *getWeight* in *fish*, we can see that

$$\langle \textit{getWeight}, (\to \texttt{num}), (), \texttt{this}\text{:}\textit{fish.size}\rangle \in^{\mathsf{m}}_P \textit{fish}$$

but for *getWeight* in *colorfish*, we have

$$\langle \textit{getWeight}, (\to \texttt{num}), (), (* \texttt{ super this}\text{:}\textit{fish.getWeight}() \texttt{ this}\text{:}\textit{colorfish.color})\rangle \in^{\mathsf{m}}_P \textit{colorfish}$$

19.2 MiniJava Evaluation

Since MiniJava provides mutable objects, the evaluation of MiniJava programs requires a store, as in Chapter 10. But unlike mutable variables in ISWIM, the mutable elements in MiniJava are objects, which can be used as values. Consequently, we not only add σ to the set of MiniJava expressions, but also to the set of values. Basic constants, `null`, and objects (represented by store addresses) are the only kind of values. To add σ as an expression, it turns out to be convenient to treat σs as variables. We define also the set of evaluation contexts.

$$
\begin{array}{rcl}
X & = & \text{a variable name, } \texttt{this}\text{, or } \sigma \\
\\
V & = & b \\
& | & \texttt{null} \\
& | & \sigma \\
\\
E & = & [\,] \\
& | & E\texttt{:}c.f \\
& | & E\texttt{:}c.f := M \\
& | & V\texttt{:}c.f := E \\
& | & E.m(M, \ldots M) \\
& | & V.m(V, \ldots V, E, M, \ldots M) \\
& | & \texttt{super } V\texttt{:}c.m(V, \ldots V, E, M, \ldots M) \\
& | & (c)E \\
& | & (o^n\ V \ldots V\ E\ M \ldots\ M)
\end{array}
$$

An object in the store $\langle T, \mathcal{FS}\rangle$ consists of a class tag T, which indicates the class instantiated by the object, and a mapping $\mathcal{FS}$ from class-prefixed field names to values. The class tag is needed for method dispatch and for casts.

The tag is only meaningful in the context of the program P; the evaluation rules combine the tag with static information derived from P. In addition, the program P determines the fields to create for a `new` T expression. Thus, the reduction rules have the form $P \vdash M \longmapsto_{\mathsf{mj}} M'$ to expose the dependence on the (static) program P.

$$
\begin{array}{l}
P \vdash \langle E[\texttt{new } c], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\sigma], \Sigma[\sigma \leftarrow \langle c, \mathcal{FS}\rangle]\rangle \\
\quad \text{where } \sigma \notin \mathrm{dom}(\Sigma) \\
\quad \text{and } \mathcal{FS} = \{\langle c'.f, V\rangle \mid \langle c'.f, T, V\rangle \in^{\mathsf{f}}_P c\} \\
P \vdash \langle E[\sigma\texttt{:}c.f], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma\rangle \\
\quad \text{where } \Sigma(\sigma) = \langle c', \mathcal{FS}\rangle \text{ and } \mathcal{FS}(c.f) = V \\
P \vdash \langle E[\sigma\texttt{:}c.f := V], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma[\sigma \leftarrow \langle c', \mathcal{FS}[c.f \leftarrow V]\rangle]\rangle \\
\quad \text{where } \Sigma(\sigma) = \langle c', \mathcal{FS}\rangle \\
P \vdash \langle E[\sigma.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\texttt{this} \leftarrow \sigma]], \Sigma\rangle \\
\quad \text{where } \Sigma(\sigma) = \langle c, \mathcal{FS}\rangle \text{ and } \langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c \\
P \vdash \langle E[\texttt{super } \sigma\texttt{:}c.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\texttt{this} \leftarrow \sigma]], \Sigma\rangle \\
\quad \text{where } \langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c \\
P \vdash \langle E[(c)\sigma], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\sigma], \Sigma\rangle \\
\quad \text{where } \Sigma(\sigma) = \langle c', \mathcal{FS}\rangle \text{ and } c' \leq_P c \\
P \vdash \langle E[(c)\texttt{null}], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\texttt{null}], \Sigma\rangle \\
P \vdash \langle E[(o^n\ V_1 \ldots\ V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma\rangle \\
\quad \text{where } \delta(o^n, V_1, \ldots V_n) = V
\end{array}
$$

Given the above reduction rules, the evalution of an expression can get stuck in many ways. For example, a field access can fail because the field does not exist in the object, a method call

can fail because the method is not present, a cast can fail because the object is not an instance of the specified type, or the target of an object operation can be `null`.

In the next section, we develop type rules that eliminate many of the stuck states, but the type rules will not be able to avoid all of them. For example, the type system will not reject programs that (eventually) call a method of `null`. Also, if the cast operation is to be useful in the same way that Java's cast is useful, the type system cannot prevent cast failures.

We can nevertheless define a sound language by adding rules that signal errors in certain cases, but rely on type checking to avoid other kinds of errors. In other words, soundness does not mean that errors are impossible, but that *certain kinds* of errors are impossible. For an object-oriented language like Java, the kinds of errors that soundness should preclude are "message not understood" errors (i.e., no such method or field) for non-`null` objects.

The error rules that we add to MiniJava are analogous to run-time exceptions in the real Java language. They are not "message not understood" errors, but instead handle `null` and cast failures.

$$P \vdash \langle E[\texttt{null}.c.f], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$$
$$P \vdash \langle E[\texttt{null}.c.f := V], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$$
$$P \vdash \langle E[\texttt{null}.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$$
$$P \vdash \langle E[(c)\sigma], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: bad cast}, \Sigma\rangle$$
$$\text{where } \Sigma(\sigma) = \langle c', \mathcal{FS}\rangle \text{ and } c' \not\leq_P c$$

▷ **Exercise 19.1.** Suppose that MiniJava is sound (using the type rules developed in the next section), but we replace the rule

$$P \vdash \langle E[(c)\texttt{null}], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\texttt{null}], \Sigma\rangle$$

with

$$P \vdash \langle E[(c)\texttt{null}], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: bad cast}, \Sigma\rangle$$

Is the revised MiniJava still sound? Which of the above rules most closely matches true Java evaluation?[1]

▷ **Exercise 19.2.** Show the evaluation of the *fish* example program's expression in Section 19.1.

19.3 MiniJava Type Checking

To check a MiniJava program P, we first verify that P satisfies all of the ClassesOnce, etc. relations. We then check the class declarations in P with $P \vdash_{\mathsf{d}} \_$, and finally check P's top-level expression with $P, \_ \vdash_{\mathsf{e}} \_ : \_$:

$$\frac{P \vdash_{\mathsf{d}} \hat{c}_1 \ldots \quad P \vdash_{\mathsf{d}} \hat{c}_n \quad P, \vdash_{\mathsf{e}} M : T}{\vdash_{\mathsf{p}} P}$$

where $P = \hat{c}_1 \ldots \hat{c}_n\ M$, ClassesOnce($P$), ObjectBuiltin($P$), FieldsOncePerClass($P$), MethodsOncePerClass($P$), ClassesDefined($P$), WellFoundedClasses($P$), and OverridesConsistent(P)

[1] Don't guess. Consult the Java language specification.

Checking a declaration with $P \vdash_{\mathsf{d}} \_$ requires checking the declared class's fields with $P \vdash_{\mathsf{f}} \_$ and methods with $P, \_ \vdash_{\mathsf{m}} \_$. The extra information to the left of $\vdash$ in $P, \_ \vdash_{\mathsf{m}} \_$ supplies a type for `this`.

$$\frac{P \vdash_{\mathsf{f}} \hat{f}_1 \ldots \quad P \vdash_{\mathsf{f}} \hat{f}_n \quad P, c \vdash_{\mathsf{m}} \hat{m}_1 \ldots \quad P, c \vdash_{\mathsf{m}} \hat{m}_m}{P \vdash_{\mathsf{d}} \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \hat{f}_1 \ldots\ \hat{f}_n\ \hat{m}_1 \ldots\ \hat{m}_m\ \}}$$

Checking a field with $P \vdash_{\mathsf{f}} \_\_$ consists of checking the initial value expression (in the empty environment) to ensure that it has the declared type:

$$\frac{P, \vdash_{\mathsf{e}} V : T}{P \vdash_{\mathsf{f}} T\ f = V}$$

Checking a method with $P, \_ \vdash_{\mathsf{m}} \_$ consists of checking the body expression of the method. The environment of the body expression provides a type for `this` and for each formal argument. For reasons explained further below, we use $P, \_ \vdash_{\mathsf{s}} \_ : \_$ to check the body expression's type instead of $P, \_ \vdash_{\mathsf{e}} \_ : \_$.

$$\frac{P, \texttt{this}{:}c, X_1{:}T_1, \ldots X_n{:}T_n \vdash_{\mathsf{s}} M : T_0}{P, c \vdash_{\mathsf{m}} T_0\ m(T_1\ X_1, \ldots T_n\ X_n)\{\ M\ \}}$$

The $P, \_ \vdash_{\mathsf{e}} \_ : \_$ rules checks expressions for a program P in a given environment. An expression `new` c always has type c, and `null` can be given any class type. The rule for variables uses the environment, as usual. The rules for fields and methods consult the program via information-gathering relations, such as $\in^{\mathsf{f}}_P$. A cast to c has type c as long as its expression has some class type.

$$P, \Gamma \vdash_{\mathsf{e}} \texttt{new}\ c : c \qquad P, \Gamma \vdash_{\mathsf{e}} \texttt{null} : c \qquad P, \Gamma \vdash_{\mathsf{e}} X : T \text{ where } \Gamma(X) = T$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} M : c'}{P, \Gamma \vdash_{\mathsf{e}} M{:}c.f : T} \quad \text{where } \langle c.f, T, V\rangle \in^{\mathsf{f}}_P c' \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} M : c' \quad P, \Gamma \vdash_{\mathsf{s}} M' : T}{P, \Gamma \vdash_{\mathsf{e}} M{:}c.f := M' : T} \quad \text{where } \langle c.f, T, V\rangle \in^{\mathsf{f}}_P c'$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} M : c \quad P, \Gamma \vdash_{\mathsf{s}} M_1 : T_1 \ldots \quad P, \Gamma \vdash_{\mathsf{s}} M_n : T_n}{P, \Gamma \vdash_{\mathsf{e}} M.m(M_1, \ldots M_n) : T_0} \quad \text{where } \langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M_0\rangle \in^{\mathsf{m}}_P c$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} M_1 : T_1 \ldots \quad P, \Gamma \vdash_{\mathsf{s}} M_n : T_n}{P, \Gamma \vdash_{\mathsf{e}} \texttt{super}\ X{:}c.m(M_1, \ldots M_n) : T_0} \quad \text{where } \Gamma(X) = c',\ c' \prec_P c \text{ and } \langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} M : c'}{P, \Gamma \vdash_{\mathsf{e}} (c)M : c} \qquad P, \Gamma \vdash_{\mathsf{e}} b : B \text{ where } \mathcal{B}(b) = B$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} M_1 : B_1 \ \ldots \quad P, \Gamma \vdash_{\mathsf{e}} M_n : B_n}{P, \Gamma \vdash_{\mathsf{e}} (o^n\ M_1 \ldots\ M_n) : B} \quad \text{where } \Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, B\rangle$$

Many of the $P, \_ \vdash_{\mathsf{e}} \_ : \_$ rules use the subsumption rule $P, \_ \vdash_{\mathsf{s}} \_ : \_$, which allows the actual type of an expression to be a subclass of a target class type:

$$\frac{P, \Gamma \vdash_{\mathsf{e}} M : c'}{P, \Gamma \vdash_{\mathsf{s}} M : c} \quad \text{where } c' \leq_P c \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} M : B}{P, \Gamma \vdash_{\mathsf{s}} M : B}$$

Although the subtyping $P, \_ \vdash_{\mathsf{s}} \_ : \_$ rule could be defined as another $P, \_ \vdash_{\mathsf{e}} \_ : \_$ rule, distinguishing the two rules exposes the places that an implementation of the type-checking rules would need to consider subsumption (e.g., in the arguments of methods), and where the most precise type should be used (e.g., in the type of the target of a method call).

▷ **Exercise 19.3.** Show that following program's expressions are well-typed (i.e., show the $P, \_ \vdash_{\mathsf{e}} \_ : \_$ subtrees of the type derivation for the program):

```
class pt1D extends Object {
  num x = ⌜0⌝
  num move(num m){ this:pt1D.x := m }
}
class pt2D extends pt1D {
  num y = ⌜0⌝
  num move(num m){ this:pt2D.y := super this:pt1D.move(m) }
}
new pt2D.move(⌜3⌝)
```

▷ **Exercise 19.4.** Show that the *fish* example program in Section 19.1 is well-typed.

19.4 MiniJava Soundness

Evaluation of a well-typed MiniJava program cannot get stuck.

Theorem 19.1 [Soundness for MiniJava]: If $\vdash_{\mathsf{p}} P$ where $P = \hat{c}_1 \ldots \hat{c}_n \; M$, then either

- $P \vdash \langle M, \emptyset \rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{mj}} \langle V, \Sigma \rangle$ for some V and Σ; or
- $P \vdash \langle M, \emptyset \rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{mj}} \langle M, \Sigma \rangle$ implies $P \vdash \langle M, \Sigma \rangle \longmapsto_{\mathsf{mj}} \langle M', \Sigma' \rangle$; or
- $P \vdash \langle M, \emptyset \rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma \rangle$ for some Σ; or
- $P \vdash \langle M, \emptyset \rangle \longmapsto\!\!\!\!\rightarrow_{\mathsf{mj}} \langle \texttt{error: bad cast}, \Sigma \rangle$ for some Σ.

We can prove soundness in essentially the same way as for Simply Typed ISWIM, but with one important twist: intermediate steps in a MiniJava program contain store addresses as well as normal expressions.

Our type rules so far do not provide a type for a σ expression, because σ never appears in Γ. For any point in evaluation, the appropriate type for a σ depends on its mapping in the store; if σ maps to an object $\langle c, \mathcal{FS} \rangle$, then the appropriate type is c.

To type an intermediate program, therefore, we must construct an environment that maps each of the current σs to a class type, such that the environment is consistent with the store. In addition, the objects in the store must have field values that are consistent with the declared type of the field. To track these requirements, we define environment–store consistency as

follows:

$$
\begin{array}{lll}
\Gamma \leftrightharpoons_P \Sigma \quad \textbf{iff} & & \\
\quad \Sigma(\sigma) = \langle c, \mathcal{FS} \rangle \text{ implies } & \Gamma(\sigma) = c & \leftrightharpoons_1 \\
& \text{and } \mathrm{dom}(\mathcal{FS}) = \{c.f \mid \langle c.f, T, V \rangle \in^{\mathsf{f}}_P c\} & \leftrightharpoons_2 \\
& \text{and } \mathcal{FS}(c.f) = \sigma' \text{ and } \langle c.f, T, V \rangle \in^{\mathsf{f}}_P c & \leftrightharpoons_3 \\
& \quad \text{implies } \Sigma(\sigma') = \langle c', \mathcal{FS}' \rangle \text{ and } c' \leq_P T & \\
& \text{and } \langle c.f, B, V' \rangle \in^{\mathsf{f}}_P c & \leftrightharpoons_4 \\
& \quad \text{implies } \mathcal{FS}(c.f) = b \text{ and } \mathcal{B}(b) = B & \\
\quad \text{and } \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Sigma) & & \leftrightharpoons_5
\end{array}
$$

The first and last constraints, $\leftrightharpoons_1$ and $\leftrightharpoons_5$, ensure that Σ and Γ map the same addresses. The first four constraints verify each object in the store: the Γ mapping of the object should be the object's class ($\leftrightharpoons_1$); all of the fields should be intact ($\leftrightharpoons_2$); object-valued fields contain objects in the store of an appropriate type ($\leftrightharpoons_3$); and basic-typed fields contain constants of the appropriate type ($\leftrightharpoons_4$).

As usual, we need a few simple lemmas to prove soundness:

Lemma 19.2: If $\vdash_{\mathsf{p}} P$ and $P, \Gamma \vdash_{\mathsf{e}} E[M] : T$ then $P, \Gamma \vdash_{\mathsf{e}} M : T'$ for some T'.

Lemma 19.3: If $\vdash_{\mathsf{p}} P$, $P, \Gamma \vdash_{\mathsf{e}} E[M] : T$, $P, \Gamma \vdash_{\mathsf{e}} M : T'$, and $P, \Gamma \vdash_{\mathsf{e}} M' : T'$, then $P, \Gamma \vdash_{\mathsf{e}} E[M'] : T$.

Lemma 19.4: If $P, \Gamma \vdash_{\mathsf{e}} M : T$ and $X \notin \mathrm{dom}(\Gamma)$, then $P, \Gamma, X{:}T' \vdash_{\mathsf{e}} M : T$.

The proof of the above lemmas follows by induction on the structure of E. Note that no expression form in E extends the environment, so the same Γ is used for checking all subexpressions of a particular expression.

Since MiniJava supports subsumption, our soundness proof requires a stronger lemma than Lemma 19.3:

Lemma 19.5: If $\vdash_{\mathsf{p}} P$, $P, \Gamma \vdash_{\mathsf{e}} E[M] : T$, $P, \Gamma \vdash_{\mathsf{e}} M : c$, $P, \Gamma \vdash_{\mathsf{e}} M' : c'$, and $c' \leq_P c$, then $P, \Gamma \vdash_{\mathsf{s}} E[M'] : T$.

Proof for Theorem 19.5: By induction on the structure of E.

- Base case:
  - **Case** $E = [\,]$
    Since $c' \leq_P c$, $P, \Gamma \vdash_{\mathsf{s}} M' : c$.
- Inductive cases; for each sub-context E' in the cases below, $P, \Gamma \vdash_{\mathsf{e}} E'[M] : T'$ for some T' by Lemma 19.2, and by induction $P, \Gamma \vdash_{\mathsf{s}} E'[M'] : T'$. If T' is a basic type, then $P, \Gamma \vdash_{\mathsf{e}} E'[M'] : T'$, and the claim will hold by Lemma 19.3. Otherwise, $T' = c_1$ and $P, \Gamma \vdash_{\mathsf{e}} E'[M'] : c_2$ for some c_1 and c_2 such that $c_2 \leq_P c_1$.
  - **Case** $E = E'{:}c''.f$
    If $\langle c''.f, T, V \rangle \in^{\mathsf{f}}_P c_1$, then $\langle c''.f, T, V \rangle \in^{\mathsf{f}}_P c_2$ by the definition of $\in^{\mathsf{f}}_P$. Thus, $P, \Gamma \vdash_{\mathsf{e}} E[M'] : T$.
  - **Case** $E = E'{:}c''.f := M_0$
    Analogous to the previous case.
  - **Case** $E = V{:}c''.f := E'$
    Since $P, \Gamma \vdash_{\mathsf{e}} V{:}c''.f := E'[M] : T$, $P, \Gamma \vdash_{\mathsf{s}} E'[M] : T$, which means that $c_1 \leq_P T$. By transitivity, $c_2 \leq_P T$, so $P, \Gamma \vdash_{\mathsf{s}} E'[M'] : T$. Thus, $P, \Gamma \vdash_{\mathsf{e}} V{:}c''.f := E'[M'] : T$.

- **Case** $E'.m(M, \ldots M)$
  Similar to the field access case, using the definition of $\in^{\mathsf{m}}_P$, plus the OverridesConsistent(P) portion of $\vdash_{\mathsf{p}} P$.
- **Case** $V.m(V, \ldots V, E', M, \ldots M)$
  Similar to the field assignment case where E' is on the right-hand side; the use of $P, \_ \vdash_{\mathsf{s}} \_ : \_$ for method arguments means that the overall type is the same for M or M'.
- **Case** `super` V**:**$c.m(V, \ldots V, E', M, \ldots M)$
  Analogous to the previous case.
- **Case** $(c)E'$
  Since $E[M']$ has a class type, the type of the cast expression is the same with M or M'.
- **Case** $(o^n\ V \ldots V\ E\ M \ldots\ M)$
  The type of $E[M']$ cannot be a class type.

The substitution lemma similarly supports subtyping:

Lemma 19.6: If $\vdash_{\mathsf{p}} P$, $P, \Gamma, X_1{:}T_1, \ldots X_n{:}T_n \vdash_{\mathsf{e}} M : T$, $P, \Gamma \vdash_{\mathsf{e}} V_1 : T_1, \ldots$ and $P, \Gamma \vdash_{\mathsf{e}} V_n : T_n$, then $P, \Gamma \vdash_{\mathsf{e}} M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1] : T$.

The proof of Lemma 19.6 is similar to the proof of Lemma 19.5, plus an induction on n.

With the helper lemmas established, we are ready to consider the main preservation and progress lemmas.

Lemma 19.7 [Preservation for MiniJava]: If $\vdash_{\mathsf{p}} P$, $P, \Gamma \vdash_{\mathsf{e}} M : T$, $\Gamma \leftrightharpoons_P \Sigma$, and $P \vdash \langle M, \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle M', \Sigma'\rangle$, then either

- there exists a Γ' such that $P, \Gamma' \vdash_{\mathsf{s}} M' : T$ and $\Gamma' \leftrightharpoons_P \Sigma'$;
- M' is `error: dereferenced null`; or
- M' is `error: dereferenced null`.

Proof for Lemma 19.7: By examination of the reduction $P \vdash \langle M, \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle M', \Sigma'\rangle$. For each case, if execution has not halted with an error configuration, we construct the new environment Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment. In some cases, we show $P, \Gamma' \vdash_{\mathsf{e}} M' : T$, which clearly implies $P, \Gamma' \vdash_{\mathsf{s}} M' : T$.

- **Case** $P \vdash \langle E[\texttt{new}\ c], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\sigma], \Sigma[\sigma \leftarrow \langle c, \mathcal{FS}\rangle]\rangle$
  Set $\Gamma' = \Gamma, \sigma{:}c$.
  1. We are given $P, \Gamma \vdash_{\mathsf{e}} E[\texttt{new}\ c] : T$, and $\sigma \notin \mathrm{dom}(\Sigma)$. By $\leftrightharpoons_5$, $\sigma \notin \mathrm{dom}(\Gamma)$. Thus, $P, \Gamma' \vdash_{\mathsf{e}} E[\texttt{new}\ c] : T$ by Lemma 19.4. Since $P, \Gamma' \vdash_{\mathsf{e}} \texttt{new}\ c : c$ and $P, \Gamma' \vdash_{\mathsf{e}} \sigma : c$, Lemma 19.3 implies $P, \Gamma' \vdash_{\mathsf{e}} E[\sigma] : T$.
  2. Let $\Sigma(\sigma) = \langle c, \mathcal{FS}\rangle$; σ is the only new element in $\mathrm{dom}(\Sigma')$ and $\mathrm{dom}(\Gamma)$.
     $\leftrightharpoons_1$**:** $\Gamma'(\sigma) = c$.
     $\leftrightharpoons_2$**:** $\mathrm{dom}(\mathcal{FS})$ is correct by construction.
     $\leftrightharpoons_3$**:** Since no σ (the only kind of value with a class type) is in the envionrment for fields in $\vdash_{\mathsf{p}} P$, no initial field value is an object address.
     $\leftrightharpoons_4$**:** $\vdash_{\mathsf{p}} P$ ensures that any initial value V in B has the declared type of the field.
     $\leftrightharpoons_5$**:** The only change to Γ and Σ is σ.
- **Case** $P \vdash \langle E[\sigma{:}c.f], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma\rangle$
  Set $\Gamma' = \Gamma$.

1. By Lemma 19.2, $P, \Gamma \vdash_{\mathsf{e}} \sigma{:}c.f : T'$ for some T'. If V is `null`, then $\leftrightharpoons_4$ requires that T' is a class type c, and `null` can be given any class type in any environment, so $P, \Gamma' \vdash_{\mathsf{e}} E[V] : T$ by Lemma 19.3. If V is some b, then $\leftrightharpoons_3$ and $\leftrightharpoons_4$ require that b has type T', so $P, \Gamma' \vdash_{\mathsf{e}} E[V] : T$ by Lemma 19.3. Finally, if V is σ', then by $\leftrightharpoons_3$ $\Sigma(\sigma') = \langle c', \mathcal{FS}' \rangle$ where $c' \leq_P T'$. By $\leftrightharpoons_1$ and Lemma 19.5, $P, \Gamma' \vdash_{\mathsf{s}} E[V] : T$.
2. Σ and Γ are unchanged.

- **Case** $P \vdash \langle E[\sigma{:}c.f := V], \Sigma \rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma[\sigma \leftarrow \Sigma c' \mathcal{FS}[c.f \leftarrow V]] \rangle$

  Set $\Gamma' = \Gamma$.

  1. The proof is by a straightforward extension of the proof for the preceeding case.
  2. The only change to the store is a field update; thus only $\leftrightharpoons_3$ and $\leftrightharpoons_4$ are affected, and only if V is not `null`.

     $\leftrightharpoons_3$ and $\leftrightharpoons_4$: By Lemma 19.2, $P, \Gamma \vdash_{\mathsf{e}} \sigma{:}c.f := V : T'$ for some T', which in turn implies $P, \Gamma \vdash_{\mathsf{s}} V : T'$. Moreover, T' must be the same as the type for $c.f$ if the type is a basic type, in which case $\leftrightharpoons_4$ applies and holds, or a subclass of the class type for $c.f$. In the latter case, $\leftrightharpoons_1$ indicates that the type tag of V will preserve $\leftrightharpoons_3$. Thus, by Lemma 19.5, $P, \Gamma' \vdash_{\mathsf{s}} E[V] : T$.

- **Case** $P \vdash \langle E[\sigma.m(V_1, \ldots V_n)], \Sigma \rangle \longmapsto_{\mathsf{mj}} \langle E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\mathtt{this} \leftarrow \sigma]], \Sigma \rangle$

  Set $\Gamma' = \Gamma$.

  1. By Lemma 19.2, $P, \sigma.m(V_1, \ldots V_n) \vdash_{\mathsf{e}} T'$: for some T', which in turn implies $P, \Gamma \vdash_{\mathsf{e}} \sigma : c$ for some c, and $P, \Gamma \vdash_{\mathsf{e}} V_i : T_i$ for i in $[1, n]$. Finally,

     $$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M \rangle \in^{\mathsf{m}}_P c$$

     $\vdash_{\mathsf{p}} P$ proves (through $P, \_ \vdash_{\mathsf{m}} \_$) that

     $$P, \mathtt{this}{:}c', X_1{:}T_1, \ldots X_1{:}T_n \vdash_{\mathsf{e}} M : T_0$$

     where c' is the defining class of m. From the definition of $\in^{\mathsf{m}}_P$, $c \leq_P c'$. Thus, Lemma 19.6 gives

     $$P, \Gamma \vdash_{\mathsf{s}} M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\mathtt{this} \leftarrow \sigma] : T'$$

     By Lemma 19.5,

     $$P, \Gamma' \vdash_{\mathsf{s}} E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\mathtt{this} \leftarrow \sigma]] : T$$

  2. Σ and Γ are unchanged.

- **Case** $P \vdash \langle E[\mathtt{super}\ \sigma{:}c.m(V_1, \ldots V_n)], \Sigma \rangle \longmapsto_{\mathsf{mj}} \langle E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\mathtt{this} \leftarrow \sigma]], \Sigma \rangle$

  The proof is essentially the same as in the preceeding case.

- **Case** $P \vdash \langle E[(c)\sigma], \Sigma \rangle \longmapsto_{\mathsf{mj}} \langle E[\sigma], \Sigma \rangle$

  Set $\Gamma' = \Gamma$.

  1. Since the reduction applies, $\Sigma(\sigma) = \langle c', \mathcal{FS} \rangle$ where $c' \leq_P c$. By $\leftrightharpoons_1$, $P, \Gamma \vdash_{\mathsf{e}} \sigma : c'$, Thus, by Lemma 19.5, $P, \Gamma \vdash_{\mathsf{s}} E[\sigma] : T$.
  2. Σ and Γ are unchanged.

- **Case** $P \vdash \langle E[(c)\texttt{null}], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\texttt{null}], \Sigma\rangle$
  Set $\Gamma' = \Gamma$.
  1. The cast expression has type c, and `null` can be given type c, so $P, \Gamma \vdash_{\mathsf{e}} E[\texttt{null}] : T$ by Lemma 19.3.
  2. Σ and Γ are unchanged.
- **Case** $P \vdash \langle E[(o^n\ V_1 \ldots\ V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma\rangle$
  Set $\Gamma' = \Gamma$.
  1. By Assumption 1 (see Chapter 12), the basic constant V has the same type as the operation expression. By Lemma 19.3, $P, \Gamma \vdash_{\mathsf{e}} E[V] : T$.
  2. Σ and Γ are unchanged.
- **Case** $P \vdash \langle E[\texttt{null:}c.f], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$
  Since the result is an error state, the claim holds.
- **Case** $P \vdash \langle E[\texttt{null:}c.f := V], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$
  Since the result is an error state, the claim holds.
- **Case** $P \vdash \langle E[\texttt{null}.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$
  Since the result is an error state, the claim holds.
- **Case** $P \vdash \langle E[(c)\sigma], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: bad cast}, \Sigma\rangle$
  Since the result is an error state, the claim holds.

Lemma 19.8 [Progress for MiniJava]: If $\vdash_{\mathsf{p}} P$, $P, \Gamma \vdash_{\mathsf{e}} M : T$, and $\Gamma \leftrightharpoons_P \Sigma$, then either $M \in V$ or there exists a $\langle M', \Sigma'\rangle$ such that $P \vdash \langle M, \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle M', \Sigma'\rangle$.

Proof for Lemma 19.8: By induction on the structure of $P, \Gamma \vdash_{\mathsf{e}} M : T$.

- Base cases:
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} \texttt{new}\ c : c$
    The object-creating reduction always applies.
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} \texttt{null} : c$
    The expression `null` is a value.
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} X : T$
    The variable X must be an object address σ, which is a value.
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} b : B$
    The expression b is a value.
- Inductive cases; since (by inspection) every type rule requires subexpressions to have a type, the inductive hypothesis applies to all subexpressions. If any subexpression is not a value, then it is a candidate for a nested evaluation context (compare E to M) and the claim holds. Thus, for the remainder of the proof, assume that all subexpressions are values.
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} V\texttt{:}c.f : T$
    If V is `null`, then the error-signalling reduction applies. Otherwise, V must have a class type that is a subtype of c; in particular, it must be an object address σ. By $\leftrightharpoons_5$, $\leftrightharpoons_1$, and $\leftrightharpoons_2$, σ contains the selected field.
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} V_0\texttt{:}c.f := V : T$
    Analogous to the previous case.
  - **Case** $P, \Gamma \vdash_{\mathsf{e}} V.m(V_1, \ldots V_n) : T$
    If V is `null`, then the error-signalling reduction applies. Otherwise, V must have a class type containing a method m with n arguments, which implies that V is an object address. This information, along with $\leftrightharpoons_5$ and $\leftrightharpoons_1$, implies that method call can proceed.

- **Case** $P, \Gamma \vdash_{\mathtt{e}} \mathtt{super}\ V\mathtt{:}c.m(V_1, \ldots V_n) : T$
  Analogous to the previous case; V is syntactically restricted so that it cannot be `null`.
- **Case** $P, \Gamma \vdash_{\mathtt{e}} (c)V : T$
  Either V is `null`, in which case the cast always succeeds, or V must be an object address (to have a class type), and the cast will either succeed or produce an error.
- **Case** $P, \Gamma \vdash_{\mathtt{e}} (o^n\ V_1 \ldots\ V_n) : B$
  By Assumption 2 (see Chapter 12), δ provides a result for the primitive operation.

▷ **Exercise 19.5.** Suppose that we extend MiniJava with a Java-like `instanceof` operator:

$$(M\ \mathtt{instanceof}\ c)$$

For simplicity, assume that b includes suitable return values for `instanceof`, b_t and b_f, and that B contains a corresponding element B_b such that $\mathcal{B}(b_t) = B_b$ and $\mathcal{B}(b_f) = B_b$. The type rule for the new form is

$$\frac{P, \Gamma \vdash_{\mathtt{e}} M : c'}{P, \Gamma \vdash_{\mathtt{e}} (M\ \mathtt{instanceof}\ c) : B_b}$$

Define new reduction rules to handle `instanceof`, and describe the proof modifications needed to demonstrate soundness for the extended language.

19.5 MiniJava Summary

Syntax

P	=	$\hat{c} \ldots\ \hat{c}\ M$	a program
$\hat{c}$	=	`class` c `extends` c { $\hat{f} \ldots\ \hat{f}\ \hat{m} \ldots\ \hat{m}$ }	class declaration
$\hat{f}$	=	$T\ f = V$	field declaration
$\hat{m}$	=	$T\ m(T\ X, \ldots T\ X)\{\ M\ \}$	method declaration
M	=	X	
	|	`null`	
	|	`new` c	
	|	$M{:}c.f$	field access
	|	$M{:}c.f := M$	field assignment
	|	$M.m(M, \ldots M)$	method call
	|	`super` $X{:}c.m(M, \ldots M)$	super call
	|	$(c)M$	cast
	|	b	
	|	$(o^n\ M \ldots\ M)$	
T	=	c	
	|	B	
V	=	b	
	|	`null`	
	|	σ	
E	=	$[\,]$	
	|	$E{:}c.f$	
	|	$E{:}c.f := M$	
	|	$V{:}c.f := E$	
	|	$E.m(M, \ldots M)$	
	|	$V.m(V, \ldots V, E, M, \ldots M)$	
	|	`super` $V{:}c.m(V, \ldots V, E, M, \ldots M)$	
	|	$(c)E$	
	|	$(o^n\ V \ldots V\ E\ M \ldots\ M)$	
c	=	a class name or `Object`	
f	=	a field name	
m	=	a method name	
X	=	a variable name, `this`, or a σ	
σ	=	a store address	
b	=	a basic constant	
o^n	=	an n-ary primitive operation	
B	=	a basic type	
$\mathcal{FS}$	=	a function $\{\langle c.f, V\rangle, \ldots\}$	
Σ	=	a function $\{\langle \sigma, \mathcal{FS}\rangle, \ldots\}$	

Information-Gathering Relations

ClassesOnce(P) **iff**
($\texttt{class}\ c\ \cdots\ \texttt{class}\ c'$ is in P) implies $c \neq c'$

ObjectBuiltin(P) **iff**
$\texttt{class Object}$ is not in P

FieldsOncePerClass(P) **iff**
($\texttt{class}\ c\ \texttt{extends}\ c'\ \{\cdots\ T_1\ f_1\ \cdots\ T_1\ f_2\ \cdots\}$ is in P) implies $f_1 \neq f_2$

MethodsOncePerClass(P) **iff**
($\texttt{class}\ c\ \texttt{extends}\ c'\ \{\cdots\ T_1\ m_1\ \cdots\ T_1\ m_2\ \cdots\}$ is in P) implies $m_1 \neq m_2$

ClassesDefined(P) **iff**
(c is in P) implies $c = \texttt{Object}$ or ($\texttt{class}\ c$ is in P)

WellFoundedClasses(P) **iff**
$\leq_P$ is antisymmetirc

OverridesConsistent(P) **iff**
$(\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c_1)$
and $(\langle m, (T'_1 \ldots\ T'_n \to T'_0), (X'_1, \ldots X'_n), M'\rangle \Subset^{\mathsf{m}}_P c_2)$
implies ($c_1 \not\leq_P c_2$ or $(T_1 \ldots\ T_n \to T_0) = (T'_1 \ldots\ T'_n \to T'_0)$)

$c \prec_P c'$ **iff**
$\texttt{class}\ c\ \texttt{extends}\ c'$ is in P

$\langle c.f, T, V\rangle \Subset^{\mathsf{f}}_P c$ **iff**
$\texttt{class}\ c\ \texttt{extends}\ c'\ \{\cdots\ T\ f = V\ \cdots\}$ is in P

$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c$ **iff**
$\texttt{class}\ c\ \texttt{extends}\ c'\ \{\cdots\ T_0\ m(T_1\ X_1, \ldots T_n\ X_n)\{\ M\ \}\ \cdots\}$ is in P

$\leq_P$ = transitive, reflexive closure of $\prec_P$

$\langle c.f, T, V\rangle \in^{\mathsf{f}}_P c'$ **iff**
$\langle c.f, T, V\rangle \Subset^{\mathsf{f}}_P c$ and $c' \leq_P c$

$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c'$ **iff**
$\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \Subset^{\mathsf{m}}_P c$
and $c = \min\{c \mid c' \leq_P c$ and $\langle m, (T'_1 \ldots\ T'_n \to T'_0), (X'_1, \ldots X'_n), M'\rangle \Subset^{\mathsf{m}}_P c\}$

Reductions

$P \vdash \langle E[\texttt{new } c], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\sigma], \Sigma[\sigma \leftarrow \langle c, \mathcal{FS}\rangle]\rangle$
where $\sigma \notin \mathrm{dom}(\Sigma)$
and $\mathcal{FS} = \{\langle c'.f, V\rangle \mid \langle c'.f, T, V\rangle \in^{\mathsf{f}}_P c\}$

$P \vdash \langle E[\sigma{:}c.f], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma\rangle$
where $\Sigma(\sigma) = \langle c', \mathcal{FS}\rangle$ and $\mathcal{FS}(c.f) = V$

$P \vdash \langle E[\sigma{:}c.f := V], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma[\sigma \leftarrow \langle c', \mathcal{FS}[c.f \leftarrow V]\rangle]\rangle$
where $\Sigma(\sigma) = \langle c', \mathcal{FS}\rangle$

$P \vdash \langle E[\sigma.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\texttt{this} \leftarrow \sigma]], \Sigma\rangle$
where $\Sigma(\sigma) = \langle c, \mathcal{FS}\rangle$ and $\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c$

$P \vdash \langle E[\texttt{super } \sigma{:}c.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[M[X_n \leftarrow V_n] \ldots [X_1 \leftarrow V_1][\texttt{this} \leftarrow \sigma]], \Sigma\rangle$
where $\langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c$

$P \vdash \langle E[(c)\sigma], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\sigma], \Sigma\rangle$
where $\Sigma(\sigma) = \langle c', \mathcal{FS}\rangle$ and $c' \leq_P c$

$P \vdash \langle E[(c)\texttt{null}], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[\texttt{null}], \Sigma\rangle$

$P \vdash \langle E[(o^n\ V_1 \ldots\ V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle E[V], \Sigma\rangle$
where $\delta(o^n, V_1, \ldots V_n) = V$

$P \vdash \langle E[\texttt{null}{:}c.f], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$

$P \vdash \langle E[\texttt{null}{:}c.f := V], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$

$P \vdash \langle E[\texttt{null}.m(V_1, \ldots V_n)], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: dereferenced null}, \Sigma\rangle$

$P \vdash \langle E[(c)\sigma], \Sigma\rangle \longmapsto_{\mathsf{mj}} \langle \texttt{error: bad cast}, \Sigma\rangle$
where $\Sigma(\sigma) = \langle c', \mathcal{FS}\rangle$ and $c' \not\leq_P c$

Type Rules

$$\frac{P \vdash_{\mathsf{d}} \hat{c}_1 \ldots \quad P \vdash_{\mathsf{d}} \hat{c}_n \quad P, \vdash_{\mathsf{e}} M : T}{\vdash_{\mathsf{p}} P}$$

where $P = \hat{c}_1 \ldots\ \hat{c}_n\ M$, ClassesOnce($P$),
ObjectBuiltin(P), FieldsOncePerClass(P),
MethodsOncePerClass(P), ClassesDefined(P),
WellFoundedClasses(P),
and OverridesConsistent(P)

$$\frac{P \vdash_{\mathsf{f}} \hat{f}_1 \ldots \quad P \vdash_{\mathsf{f}} \hat{f}_n \quad P, c \vdash_{\mathsf{m}} \hat{m}_1 \ldots \quad P, c \vdash_{\mathsf{m}} \hat{m}_m}{P \vdash_{\mathsf{d}} \texttt{class } c \texttt{ extends } c' \ \{\ \hat{f}_1 \ldots\ \hat{f}_n\ \hat{m}_1 \ldots\ \hat{m}_m\ \}}$$

$$\frac{P, \vdash_{\mathsf{e}} V : T}{P \vdash_{\mathsf{f}} T\ f = V} \qquad \frac{P, \texttt{this}{:}c, X_1{:}T_1, \ldots X_n{:}T_n \vdash_{\mathsf{s}} M : T_0}{P, c \vdash_{\mathsf{m}} T_0\ m(T_1\ X_1, \ldots T_n\ X_n)\{\ M\ \}}$$

$$P,\Gamma \vdash_{\mathsf{e}} \texttt{new}\ c : c \qquad P,\Gamma \vdash_{\mathsf{e}} \texttt{null} : c \qquad P,\Gamma \vdash_{\mathsf{e}} X : T \text{ where } \Gamma(X) = T$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} M : c'}{P,\Gamma \vdash_{\mathsf{e}} M{:}c.f : T} \text{ where } \langle c.f, T, V\rangle \in^{\mathsf{f}}_P c' \qquad \frac{P,\Gamma \vdash_{\mathsf{e}} M : c' \quad P,\Gamma \vdash_{\mathsf{s}} M' : T}{P,\Gamma \vdash_{\mathsf{e}} M{:}c.f := M' : T} \text{ where } \langle c.f, T, V\rangle \in^{\mathsf{f}}_P c'$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} M : c \quad P,\Gamma \vdash_{\mathsf{s}} M_1 : T_1 \ldots \quad P,\Gamma \vdash_{\mathsf{s}} M_n : T_n}{P,\Gamma \vdash_{\mathsf{e}} M.m(M_1, \ldots M_n) : T_0}$$
$$\text{where } \langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M_0\rangle \in^{\mathsf{m}}_P c$$

$$\frac{P,\Gamma \vdash_{\mathsf{s}} M_1 : T_1 \ldots \quad P,\Gamma \vdash_{\mathsf{s}} M_n : T_n}{P,\Gamma \vdash_{\mathsf{e}} \texttt{super}\ X{:}c.m(M_1, \ldots M_n) : T_0}$$
$$\text{where } \Gamma(X) = c',\ c' \prec_P c$$
$$\text{and } \langle m, (T_1 \ldots\ T_n \to T_0), (X_1, \ldots X_n), M\rangle \in^{\mathsf{m}}_P c$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} M : c'}{P,\Gamma \vdash_{\mathsf{e}} (c)M : c} \qquad P,\Gamma \vdash_{\mathsf{e}} b : B \text{ where } \mathcal{B}(b) = B$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} M_1 : B_1\ \ldots \quad P,\Gamma \vdash_{\mathsf{e}} M_n : B_n}{P,\Gamma \vdash_{\mathsf{e}} (o^n\ M_1 \ldots\ M_n) : B} \text{ where } \Delta(o^n) = \langle\langle B_1, \ldots B_n\rangle, B\rangle$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} M : c'}{P,\Gamma \vdash_{\mathsf{s}} M : c} \text{ where } c' \leq_P c \qquad \frac{P,\Gamma \vdash_{\mathsf{e}} M : B}{P,\Gamma \vdash_{\mathsf{s}} M : B}$$

Part IV

Appendices

Appendix 20: Structural Induction

The relation **r** defined in the previous lecture is a function, but how could we prove it? Here's a semi-formal argument: The relation **r** is defined by a pair of constraints. The first constraint applies to a • expression that starts with `f`, and the result is determined uniquely by the right-hand part of the expression. The second constraint applies to a • expression that starts with `t`, and the result is determined uniquely as `t`. Since an expression can start with either `f` or `t`, then only one constraint will apply, and the result is unique.

The above argument is fairly convincing because it relies on a directly observable property of the expression: whether it is a • expression that starts with `f` or `t`. Other claims about relations over B may not be so simple to prove. For example, it is not immediately obvious that $eval_{\mathbf{r}}$ is a function. The fundamental reason is that B is recursively defined; there is no way to enumerate all of the interesting cases directly. To prove more general claims, we must use **induction**.

Mathematical induction arguments are based on the natural numbers. A claim is proved for 0, then for $n+1$ assuming the claim for n, and then the claim follows for all n. For most of the proofs we will consider, no natural number n is readily provided for elements of the set (such as B), but one could be derived for an expression based on the number of steps in an argument that the expression belongs to the set (as in §1.1).

Instead of finding appropriate numbers for set elements to enable mathematical induction, however, we will use **structural induction**, which operates directly on the grammar defining set elements. The underlying principle is the same.

20.1 Detecting the Need for Structural Induction

Structural induction applies when proving a claim about a recursively-defined set. The set is well-founded, of course, only if there are atomic elements in the definition; those elements constitute the **base cases** for induction proofs. The self-referential elements of the definition constitute the **induction cases**.

For example, suppose we have the following definition:

$$\begin{array}{rcl} P & = & \alpha \\ & | & (\beta \otimes P) \\ & | & (P \odot P) \end{array}$$

Example members of P include α, $(\beta \otimes \alpha)$, $(\alpha \odot \alpha)$, and $((\beta \otimes (\beta \otimes \alpha)) \odot (\beta \otimes \alpha))$. Here is a claim about elements of P:

> **Theorem 20.1**: For any P, P contains an equal number of βs and $\otimes$s.

The truth of this claim is obvious. But since Theorem 20.1 makes a claim about all possible instances of P, and since the set of Ps is recursively defined, *formally* it must be proved by structural induction.

> **Guideline:** To prove a claim about all elements of a recursively defined set, use structural induction.

The key property of a correctly constructed induction proof is that it is guaranteed to cover an entire set, such as the set of Ps. Here is an example, a proof of the above claim:

> **Proof for Theorem 20.1**: By induction on the structure of P.
>
> - Base cases:

  - **Case** α
    α contains 0 βs and 0 $\otimes$s, so the claim holds.

- Inductive cases:
  - **Case** $(\beta \otimes P)$
    By induction, since P is a substructure of $(\beta \otimes P)$, P contains an equal number—say, n—of βs and $\otimes$s. Therefore, $(\beta \otimes P)$ contains $n + 1$ βs and $\otimes$s, and the claim holds.
  - **Case** $(P_1 \odot P_2)$
    By induction, P_1 contains an equal number—say, n_1—of βs and $\otimes$s. Similarly, P_2 contains n_2 βs and $\otimes$s. Therefore, $(P_1 \odot P_2)$ contains $n_1 + n_2$ βs and $\otimes$s, and the claim holds.

The above proof has a relatively standard shape. The introduction, "by induction on the structure of P" encapsulates a boilerplate argument, which runs as follows:

> The claim must hold for any P. So assume that an arbitrary P is provided. If P has the shape of a base case (i.e., no substructures that are Ps) then we show how we can deduce the claim immediately. Otherwise, we rely on induction and assume the claim for substructures within P, and then deduce the claim. The claim then holds for all P by the principle of structural induction.

The proof for Theorem 20.1 contains a single base case because the definition of P contains a single case that is not self-referential. The proof contains two inductive cases because the definition of P contains two self-referential cases.

> **Guideline:** For a structural induction proof, use the standard format. The section for base cases should contain one case for each base case in the set's definition. The section for induction cases should contain one case for each remaining case in the set's definition.

The standard shape for a structural induction proof serves as a guide for both the proof writer and the proof reader. The proof writer develops arguments for individual cases one by one. The proof reader can then quickly check that the proof has the right shape, and concentrate on each case to check its correctness separately.

Occasionally, within the section for base cases or induction cases in a proof, the case split uses some criteria other than the structure used for induction. This is acceptable as long as the case split merely collapses deeply nested cases, as compared to a standard-format proof. A non-standard case split must cover all of the base or inductive cases in an obvious way.

All proofs over the structure of P, including Theorem 20.1, have the same outline:

> **Proof for Theorem** : By induction on the structure of P.
>
> - Base cases:
>   - **Case** α
>   ...
> - Inductive cases:
>   - **Case** $(\beta \otimes P)$
> $\cdots$ By induction, [claim about P] ...
>   - **Case** $(P_1 \odot P_2)$
> $\cdots$ By induction, [claim about P_1] and, by induction, [claim about P_2] $\cdots$

Only the details hidden by "$\cdots$" depend on the claim being proved. The following claim is also about all Ps, so it will have the same outline as above.

Theorem 20.2: For any P, P contains at least one α.

▷ **Exercise 20.1.** Prove Theorem 20.2.

20.2 Definitions with Ellipses

Beware of definitions that contain ellipses (or asterisks), because they contain hidden recursions. For example,

$$\begin{array}{rcl} W & = & \alpha \\ & | & (\beta W W \ldots W) \end{array}$$

allows an arbitrary number of Ws in the second case. It might be more precisely defined as

$$\begin{array}{rcl} W & = & \alpha \\ & | & (\beta Y) \\ Y & = & W \\ & | & YW \end{array}$$

Expanded this way, we can see that proofs over instances of W technically require mutual induction proofs over Y. In practice, however, the induction proof for Y is usually so obvious that we skip it, and instead work directly on the original definition.

Theorem 20.3: Every W contains α.

Proof for Theorem 20.3: By induction on the structure of W.

- Base case:
  - **Case** α
    The claim obviously holds.
- Inductive case:
  - **Case** $(\beta W_0 W_1 \ldots W_n)$
    Each W_i contains α, and W contains at least one W_i, so the claim holds.

The following theorem can also be proved reasonably without resorting to mutual induction.

Theorem 20.4: For any W, each β in W is preceded by an open parenthesis.

▷ **Exercise 20.2.** Prove Theorem 20.4.

20.3 Induction on Proof Trees

The following defines the set $\triangle P$, read "P is pointy":

$$\begin{array}{ll} \triangle\alpha & \text{[always]} \\ \triangle(P_1 \odot P_2) & \text{if } \triangle P_1 \text{ and } \triangle P_2 \end{array}$$

As usual, the set of $\triangle P$s is defined as the smallest set that satisfies the above rules. The first rule defines a base case, while the second one is self-referential.

Another common notation for defining a set like $\triangle P$ is derived from the notion of proof trees:

$$\triangle\alpha \qquad\qquad \frac{\triangle P_1 \quad \triangle P_2}{\triangle(P_1 \odot P_2)}$$

When $\triangle P$ appears above a line in a definition, we understand it to mean that there must be a pointy proof tree in its place with $\triangle P$ at the bottom. This convention is used in the self-referential second rule.

Both notations simultaneously define two sets: the set of pointy indications $\triangle P$, and the set of pointy proof trees. Nevertheless, both sets are simply patterns of symbols, defined recursively. Examples of pointy proof trees include the following:

$$\triangle\alpha \qquad \dfrac{\triangle\alpha \quad \dfrac{\triangle\alpha \quad \triangle\alpha}{\triangle(\alpha \odot \alpha)}}{\triangle(\alpha \odot (\alpha \odot \alpha))}$$

We can now write claims about $\triangle P$ and its pointy proof trees:

> **Theorem 20.5**: If $\triangle P$, then the pointy proof tree for $\triangle P$ contains an odd number of $\triangle$s.

The proof for this claim works just like the proof of a claim on P, by structural induction:

> **Proof for Theorem 20.5**: By induction on the structure of the pointy proof of $\triangle P$.
>
> - Base cases:
>   - **Case** $\triangle\alpha$
>   The complete tree contains $\triangle\alpha$, a line, and a check mark, which is one $\triangle$ total, so the claim holds.
> - Inductive cases:
>   - **Case** $\triangle(P_1 \odot P_2)$
>   The complete tree contains $\triangle(P_1 \odot P_2)$, plus trees for $\triangle P_1$ and $\triangle P_2$. By induction, since the tree for $\triangle P_1$ is a substructure of the tree for $\triangle(P_1 \odot P_2)$, the $\triangle P_1$ tree contains an odd number of $\triangle$s. Similarly, the $\triangle P_2$ tree contains an odd number of $\triangle$s. Two odd numbers sum to an even number, so the trees for $\triangle P_1$ and $\triangle P_2$ combined contain an even number of $\triangle$s. But the complete tree for $\triangle(P_1 \odot P_2)$ adds one more, giving an odd number of $\triangle$s total, and the claim holds.

20.4 Multiple Structures

Many useful claims about recursively-defined sets involve a connection between two different sets. For example, consider the following claim:

> **Theorem 20.6**: For all $\triangle P$, P contains no βs.

Should this claim be proved by structural induction on P, or on $\triangle P$? We can find the answer by looking closely at what the theorem lets us assume. The theorem starts "for all $\triangle P$", which means the proof must consider all possible cases of $\triangle P$. The proof should therefore proceed by induction on the structure of $\triangle P$. Then, in each case of $\triangle P$, the goal is to show something about a specific P, already determined by the choice of $\triangle P$.

> **Guideline:** A leading "for all" in a claim indicates a candidate for structural induction. An item in the claim that appears on the right-hand side of an implication is *not* a candidate.

In contrast to Theorem 20.6, Theorem 20.5 was stated "if $\triangle P$, then ... ". This "if ... then" pattern is actually serving as a form of "for all". When in doubt, try to restate the theorem in terms of "for all".

Proof for Theorem 20.6: By induction on the structure of the proof of $\triangle P$.

- Base cases:
  - **Case** $\triangle\alpha$
    In this case, P is α, and the claim holds.
- Inductive cases:
  - **Case** $\triangle(P_1 \odot P_2)$
    By induction, P_1 and P_2 each contain no βs, so $(P_1 \odot P_2)$ contains no βs.

Here is a related, but different claim:

Theorem 20.7: For all P, either 1) P contains a β, or 2) $\triangle P$.

▷ **Exercise 20.3.** Prove Theorem 20.7. The theorem must be proved over a different structure than Theorem 20.6.

20.5 More Definitions and More Proofs

We can define more sets and make claims about their relationships. The proofs of those claims will follow in the same way as the ones we have seen above, differing only in the structure used for induction, and the local reasoning in each case.

Here is a new set:

$$
\begin{array}{lcll}
((\beta \otimes \alpha) \odot \alpha) & \diamond & (\beta \otimes \alpha) & \\
(\alpha \odot (\beta \otimes \alpha)) & \diamond & \alpha & \\
(\alpha \odot \alpha) & \diamond & \alpha & \\
(P_1 \odot P_2) & \diamond & (P_1' \odot P_2) & \text{if } P_1 \diamond P_1' \\
(P_1 \odot P_2) & \diamond & (P_1 \odot P_2') & \text{if } P_2 \diamond P_2'
\end{array}
$$

Like our original definition for $\triangle P$, $P \diamond P$ is defined by the least set satisfying the above rules. Examples for this set include $((\beta \otimes \alpha) \odot \alpha) \diamond (\beta \otimes \alpha)$ and $(((\beta \otimes \alpha) \odot \alpha) \beta (\alpha \odot \alpha)) \diamond ((\beta \otimes \alpha) \beta (\alpha \odot \alpha))$. The atomic elements of the definition in this case are the first three rules, and the self-reference occurs in the last two rules.

Theorem 20.8: For all $P \diamond P'$, P contains more $\odot$s than P'.

Proof for Theorem 20.8: By induction on the structure of $P \diamond P'$.

- Base cases:
  - **Case** $((\beta \otimes \alpha) \odot \alpha) \diamond (\beta \otimes \alpha)$
    In this case, P is $((\beta \otimes \alpha) \odot \alpha)$, which has one $\odot$, and P' is $(\beta \otimes \alpha)$, which has zero $\odot$s, so the claim holds.
  - **Case** $(\alpha \odot (\beta \otimes \alpha)) \diamond \alpha$
    In this case, P is $(\alpha \odot (\beta \otimes \alpha))$, which has one $\odot$, and P' is α, which has zero $\odot$s, so the claim holds.
  - **Case** $(\alpha \odot \alpha) \diamond \alpha$
    In this case, P is $(\alpha \odot \alpha)$, which has one $\odot$, and P' is α, which has zero $\odot$s, so the claim holds.
- Inductive cases:

- **Case** $(P_1 \odot P_2) \diamond (P_1' \odot P_2)$
  In this case, P is $(P_1 \odot P_2)$, which means it has as many $\odot$s as P_1 and P_2 combined. P' is $(P_1' \odot P_2)$, which means it has as many $\odot$s as P_1' and P_2 combined. By induction, P_1 has more $\odot$s than P_1', so P has more $\odot$s than P', and the claim holds.
- **Case** $(P_1 \odot P_2) \diamond (P_1 \odot P_2')$
  Analogous to the previous case.

Here is one last definition:

$$\begin{array}{rcl} V & = & \alpha \\ & | & (\beta \otimes V) \end{array}$$

Theorem 20.9: Every V is in P.

Theorem 20.10: If $\triangle P$ and P is not a V, then $P \diamond P'$ for some P'

Theorem 20.11: If $\triangle P$ and $P \diamond P'$, then $\triangle P'$.

▷ **Exercise 20.4.** Prove Theorem 20.9.

▷ **Exercise 20.5.** Prove Theorem 20.10.

▷ **Exercise 20.6.** Prove Theorem 20.11. The proof can proceed in two different ways, since the implicit "for all" applies to both $\triangle P$ and $P \diamond P'$.

Bibliography

[1] A. Appel. *Compiling with Continuations*. Cambridge Press, 1992.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[3] D. R. Chase. Safety considerations for storage allocation optimizations. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 1–10, 1988.

[4] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[5] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[6] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, 1995.

[7] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[8] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Index