PROGRAMMING LANGUAGES AND LAMBDA CALCULI

Matthias Felleisen

Department of Computer Science
Rice University

DRAFT: December 16, 1998

Contents

Preface: Why Calculi?

In a series of papers in the mid-1960's, 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.

These are the basic thoughts that should go into the preface. It is not a final version.

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 highschool, 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. Church had proposed the λ-calculus as a calculus of functions[1]. Given Landin's insight on the central role of procedures as a facility common to all languages, the λ-calculus was a natural starting point. 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

[1]With the goal of understanding all of mathematics based on this calculus

calculi in the mid-1970's, 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.

The 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. With an eye towards ML and Scheme, but for general higher-order languages.

The functional core of Landin's ISWIM [*ref:* LanISWIM] is an extension of Church's pure λ-calculus [*ref:* Church, *ref:* Bar] with primitive data and their associated primitive functions. Church had proposed the λ-calculus as a calculus of functions[2]. The calculus offers a simple, regular syntax for writing donw functions, and a simple equational system that specifies the behavior of programs. Since user-defined procedures in programming languages are the intuitive counterpart of functions, the system was a natural choice for Landin who wanted to define a programming language solely based on data and user-defined procedures. Unlike the pure calculus, ISWIM contains basic and functional constants to mimick primitive data; to avoid specialization, our variant of ISWIM contains a generic sublanguage for primitive data manipulation.

[2]With the goal of understanding all of mathematics based on this calculus

General Notation:

$\mathbb{N}$	the natural numbers	“`Nat`

General Notation for Reduction Systems:

$\longrightarrow_r$	one-step relation	“`onered r`
$\longrightarrow_r^0$	reflexive closure of one-step relation	“`oneredr r`
$\longrightarrow\!\!\!\!\rightarrow_r$	reduction (relation)	“`reduce r`
$=_r$	equality relation	$=$ `_r`
$\underset{1}{\longrightarrow\!\!\!\!\rightarrow}$	parallel reduction	“`para`
$\longmapsto_r$	standard reduction step r	“`standardr`
$\longmapsto_r^*$	transitive closure of ...	“`standardr`
$eval$	evauation function	“`eval`
$\simeq$	operational equality	“`opeq`
$\not\simeq$	operational inequality	“`nopeq`

SA: Syntax and Semantic Relations:

$\ulcorner n \urcorner$	numerals	“`goedel n`
1^+	successor SA primitive	“`add1`
1^-	predecesor SA primitive	“`sub1`
null?	null test	“`nullp`
car	first component of pair	“`car`
cons	the pair constructor	“`cons`
cdr	first component of pair	“`cdr`
a	notion of reduction for SA	
$eval_{sa}$	SA's evaluation function	“`evalsa`
$\mathbb{Z}$	the integers	“`Z`

Functional ISWIM:

(handle ... with ...)	exception handler	`“handle`
(handle ... with ...)	tagged exception handler	`“thandle`
zero?	zero test	`“zero?`
Y_v	call-by-value fixpoint operator	`“Yv`
if0	if-zero expression	`“zero?`
Consts	set of constants	
Values	set of values	
FConsts	set of functional constants	`“fconst`
BConsts	set of basic constants	`“bconst`
Vars	set of variables	`“Var`
Vals	set of values	`“Val`
$M[x \leftarrow N]$	substitution	`M“subst x N`
$\boldsymbol{\lambda}$	generic λ-calculus	`“lam`
$\boldsymbol{\lambda_v}$	λ_v-calculus	`“vvv`
[ ]	empty context	`“hole`
VAR	variables	`“var`
FV	free variables	`“fv`
BV	bound variables	`“bv`
AV	assignable variables	`“av`

Standard Reduction and Machines:

$\longmapsto$	general machine transition step	
`closure`	machine closure tag	
SR-sequence	standard reduction sequence	
EvalConts	evaluation contexts	“`evconts`
$\langle \cdot, \cdot \rangle$	machine state	“`state`
$\longmapsto_{cc}$	CC-transition	“`ccstep`
$\longmapsto_{cc}^*$	transitive closure of $\longmapsto_{cc}$	“`ccsteps`
$\longmapsto_{scc}$	SCC-transition relation	“`sccsteps`
$\longmapsto_{scc}^*$	transitive closure of $\longmapsto_{scc}$	“`sccsteps`
mt	continuation code for representing the top-level	“`mt`
$\langle \mathsf{arg}, \cdot, K \rangle$	...evaluation of function position	“`argK·`
$\langle \mathsf{narg}, \langle \cdot \rangle, \langle \cdot \rangle, K \rangle$	...evaluation of position in primitive application	“`narg`
$\langle \mathsf{fun}, \cdot, K \rangle$	...evaluation of argument position	“`funK·`
$\longmapsto_{ck}$	CK-transition relation	“`ckstep`
$\longmapsto_{ck}^*$	transitive closure of $\longmapsto_{scc}$	“`ckstep`
KCodes	CK continuation codes	“`kcodes`
$\mathcal{KC}$	mapping from *KCodes* to *EvalConts*	“`KC`
$\mathcal{CK}$	mapping from *EvalConts* to *KCodes*	“`CK`
Envs	set of environments	
Closures	set of closures	
$\langle M, E \rangle$	closure of expression M, environment E	“`closeME`
$\mathcal{U}$	mapping from *Closures* to Λ^0	“`unload`
EKCodes	CEK continuation codes	“`ekcodes`
$\mathcal{KK}$	mapping from *EKCodes* to *KCodes*	“`kcodes`
apply	interpreter application function	
unload	machine unload function	
$\longmapsto_{cc}$	CCH-transition	“`cchstep`
$\longmapsto_{cch}^*$	transitive closure of $\longmapsto_{cc}$	“`ccsteps`

Other Stuff:

error	error function	
(catch)	catch function	
(throw)	throw function	
(catch)	catch function	
$\boldsymbol{\lambda_v}$-**C**	λ_v-C-calculus	“`vc`
$\boldsymbol{\lambda_v}$-**S**	λ_v-S-calculus	“`vs`
δ_{error}	delta-error reduction	“`deltaerr`
$\boldsymbol{\lambda_v}$-**e**	error theory	
$\boldsymbol{\lambda_v}$-**a**	abort theory	
$\boldsymbol{\lambda_v - \mathbf{CS}}$		
$\mathcal{C}$	ignore/cc	“`C`
$\mathcal{A}$	abort	“`A`
$:=$	assignment	
$eval_v w$	eval for something or other	
gc	garbage collection axiom	

Chapter 1

Arithmetic

Before people learn how to program they already know how to manipulate arithmetic expresions. In particular, they can determine the result of arithmetic expressions through a simplification process:

$$13 + 6 * 5 = 13 + 30 = 43;$$

and, with the same calculus, they can also prove simple equalities between arithmetic expressions:

$$30 + 13 = 43 = 13 + 30.$$

Most programming languages also contain a sub-language that resembles this "programming language" of arithmetic. Given the familiarity of this sample language, it is natural to study it first in isolation. Such a treatment provides a better understanding of this all-too-familiar language and system of calculation. The detailed development also illustrates the basics of our approach to the definition and analysis of programming languages, and will serve as a template for the chapters to come.

1.1 Syntax and Calculus

SA (*Simple Arithmetic*) is the programming language of arithmetic expressions over the integers ($\mathbb{Z}$). To avoid the vaguaries of mathematicl syntax, SA's expressions are fully parenthesized expressions and resemble those of Scheme and Lisp. The set of expressions contains the subset of numerals, which represent integers. The distinction between the syntactic object $\ulcorner 1 \urcorner$ that represents the integer 1 may look severe but is important to understand the syntactic nature of our approach. Other than numerals the expression language also contains applications of unary and binary operation symbols to appropriate number of expressions.

Definition 1.1.1 (*SA*) SA denotes the set of expressions determined by the following context-free grammar over the alphabet $1^+, 1^-, +, -$ and the constant symbols $\ulcorner n \urcorner$ (for

$n \in \mathbb{N}$):

$$\begin{array}{rrcl} SA: & M & ::= & c \mid (u\ M) \mid (b\ M\ M) \\ & u & ::= & 1^+ \mid 1^- \\ & b & ::= & + \mid - \\ & c & ::= & \ulcorner n \urcorner, \quad \text{for } n \in \mathbb{Z}. \end{array}$$

K, L, N, in addition to M, are meta-variables that range over SA. ∎

The basic action of arithmetic calculation is the replacement of one expression with a syntactically different, but equivalent expression. Mathematically, such an action *relates* two expressions to each other. The nature of arithmetical calculations demands that the relation be an equivalence relation that is compatible with the syntactic structure of expressions. We call such a relation a *congruence* relation.

To formulate the congruence relation for SA, we proceed in stages and rely on the mathematical functions for addition and subtraction. The core of the SA relation fromulates the meaning of the successor, predecessor, addition, and subtraction symbol as that of the respective functions.

Definition 1.1.2 (a) The relation $\mathbf{a} \subseteq \mathrm{SA} \times \mathrm{SA}$ is the set

$$\begin{array}{ll} \{ & ((1^+ \ulcorner m \urcorner), \ulcorner m+1 \urcorner), \\ & ((1^- \ulcorner m \urcorner), \ulcorner m-1 \urcorner), \\ & ((+ \ulcorner m \urcorner \ulcorner n \urcorner), \ulcorner m+n \urcorner), \\ & ((- \ulcorner m \urcorner \ulcorner n \urcorner), \ulcorner m-n \urcorner) \quad \mid \quad m,n \in \mathbb{Z} \ \} \end{array}$$

We will use infix notation to indicate $\mathbf{a}$ relationships:

$$\begin{array}{lcl} (1^+ \ulcorner m \urcorner) & \mathbf{a} & \ulcorner m+1 \urcorner \\ (1^- \ulcorner m \urcorner) & \mathbf{a} & \ulcorner m-1 \urcorner \\ (+ \ulcorner m \urcorner \ulcorner n \urcorner) & \mathbf{a} & \ulcorner m+n \urcorner \\ (- \ulcorner m \urcorner \ulcorner n \urcorner) & \mathbf{a} & \ulcorner m-n \urcorner \end{array}$$

where $m, n \in \mathbb{Z}$. ∎

The specification of $\mathbf{a}$ relies on the distinction between the numeral $\ulcorner n \urcorner$ and the integer n and the distinction between the SA function symbol $+$ and integer addition. The latter is used to create new numerals like $\ulcorner m+n \urcorner$ when needed.

Put operationally, the relation $\mathbf{a}$ specifies how the application of 1^+, 1^-, $+$, and $-$, computes a result by relating applications to numerals. We therefore say "$\mathbf{a}$ *reduces*" a *redex* (*red*ucible *ex*pression) to its *contractum*. For example, $(1^+ \ulcorner 1 \urcorner)$ reduces to $\ulcorner 2 \urcorner$ and $(+ \ulcorner 2 \urcorner \ulcorner 3 \urcorner)$ reduces to $\ulcorner 5 \urcorner$, as expected. A relation on sets of expressions like $\mathbf{a}$ is called a *notion of reduction*.

A simple notion of reduction like **a** is not yet the equality relation that formalizes our intuitive notion of arithmetical calculations. It does not even relate expressions like $(-\ \ulcorner 13\urcorner\ (+\ \ulcorner 2\urcorner\ \ulcorner 3\urcorner))$ and $(-\ \ulcorner 13\urcorner\ \ulcorner 5\urcorner)$ even though the nested expressions are related by **a**. To capture this concept, we need to extend the relation **a** to a relation that can reduce sub-expressions. Technically speaking, we need to find the smallest superset of **a** that is compatible with the syntactic structrue of SA expressions. This extension is called the *compatible closure* of **a**.

In general, the compatible closure of a relation $\mathbf{R}$ on expressions extends $\mathbf{R}$ and is compatible with all possible combinations of expressions into more complex expressions as defined by the grammar of the language. For SA, only unary and binary applications build complex expressions from simple expressions. Thus, the compatible closure need only relate expressions inside of applications.

Definition 1.1.3 (*SA compatible closure of* $\mathbf{R}$*:* $\longrightarrow_R$) Let $\mathbf{R}$ be a relation over SA. Then, the compatible closure of $\mathbf{R}$, notation: $\longrightarrow_R$, is a superset of $\mathbf{R}$ such that $M \longrightarrow_R M'$ implies

$$\begin{array}{rcl}
(1^+\ M) & \longrightarrow_R & (1^+\ M'), \\
(1^-\ M) & \longrightarrow_R & (1^-\ M')), \\
(+\ M\ N) & \longrightarrow_R & (+\ M'\ N), \\
(-\ M\ N) & \longrightarrow_R & (-\ M'\ N)
\end{array}$$

for all N in SA. We use $\longrightarrow_a$ for the compatible closure of **a**. ∎

$$\dfrac{\dfrac{\dfrac{(-\ \ulcorner 12\urcorner\ \ulcorner 5\urcorner) =_a \ulcorner 7\urcorner : \mathbf{a}}{(+\ \ulcorner 5\urcorner\ \bullet) =_a (+\ \ulcorner 5\urcorner\ \ulcorner 7\urcorner): \text{comp}} \qquad (+\ \ulcorner 5\urcorner\ \ulcorner 7\urcorner) =_a \ulcorner 12\urcorner: \mathbf{a}}{(+\ \ulcorner 5\urcorner\ (-\ \ulcorner 12\urcorner\ \ulcorner 5\urcorner)) =_a \ulcorner 12\urcorner: \text{trans}} \qquad \dfrac{(+\ \ulcorner 7\urcorner\ \ulcorner 5\urcorner) =_a \ulcorner 12\urcorner : \mathbf{a}}{\ulcorner 12\urcorner =_a (+\ \ulcorner 7\urcorner\ \ulcorner 5\urcorner): \text{sym}}}{(+\ \ulcorner 5\urcorner\ (-\ \ulcorner 12\urcorner\ \ulcorner 5\urcorner)) =_a (+\ \ulcorner 7\urcorner\ \ulcorner 5\urcorner): \text{trans}}$$

Figure 1.1: Derivation of $(+\ \ulcorner 5\urcorner\ (-\ \ulcorner 12\urcorner\ \ulcorner 5\urcorner)) =_a (+\ \ulcorner 7\urcorner\ \ulcorner 5\urcorner)$

Given the notion of compatibility, we can define the congruence relation on arithmetic expressions as an equivalence relation that extends a compatible relation. Omitting symmerty yields the transitive-reflexive closure of the one-step reduction relation $(\longrightarrow_a)$, which is traditionally referred to as *the* reduction relation generated by **a**.

Definition 1.1.4 (*SA equality, reduction relation over* $\mathbf{R}$*:* $=_R$) Let $\mathbf{R}$ be a relation over SA. Then, SA-equality, notation: $=_R$, is an extension of $\longrightarrow_R$ such that $=_R$ is

reflexive: $M =_R M$;

symmetric: $M =_R N$, if $N =_R M$; and

transitive: $M =_R N$, if $M =_R L$ and $L =_R N$

for all M, L and N in SA. If the relation is not symmetric, it is the *reduction relation* over $\mathbf{R}$, notation: $\longrightarrow\!\!\!\!\!\rightarrow_R$. We refer to the congruence relation generated by $\mathbf{a}$ with $=_a$.
∎

To illustrate how the relation $=_a$ captures our intuitive arithmetic manipulations, we use an example. Consider the equation

$$5 + (12 - 5) = 7 + 5.$$

In SA's syntax, it is

$$(+ \ulcorner 5 \urcorner\ (- \ulcorner 12 \urcorner\ \ulcorner 5 \urcorner)) =_a (+ \ulcorner 7 \urcorner\ \ulcorner 5 \urcorner)$$

Its validity can be verified by the following argument:

$$\begin{array}{rcl}
(+ \ulcorner 5 \urcorner\ (- \ulcorner 12 \urcorner\ \ulcorner 5 \urcorner)) & =_a & (+ \ulcorner 5 \urcorner\ \ulcorner 7 \urcorner) \\
& & \quad \text{because } (- \ulcorner 12 \urcorner\ \ulcorner 5 \urcorner)\ \mathbf{a}\ \ulcorner 7 \urcorner \\
& & \quad \text{and because } =_a \text{ is compatibe;} \\
& & \\
(+ \ulcorner 5 \urcorner\ (- \ulcorner 12 \urcorner\ \ulcorner 5 \urcorner)) & =_a & \ulcorner 12 \urcorner \\
& & \quad \text{because } (+ \ulcorner 5 \urcorner\ \ulcorner 7 \urcorner)\ \mathbf{a}\ \ulcorner 12 \urcorner; \\
& & \\
\ulcorner 12 \urcorner & =_a & (+ \ulcorner 7 \urcorner\ \ulcorner 5 \urcorner) \\
& & \quad \text{because } (+ \ulcorner 7 \urcorner\ \ulcorner 5 \urcorner)\ \mathbf{a}\ \ulcorner 12 \urcorner \\
& & \quad \text{and because } =_a \text{ is symmetric;} \\
(+ \ulcorner 5 \urcorner\ (- \ulcorner 12 \urcorner\ \ulcorner 5 \urcorner)) & =_a & (+ \ulcorner 7 \urcorner\ \ulcorner 5 \urcorner) \\
& & \quad \text{because } =_a \text{ is transitive.}
\end{array}$$

Such an argument can be arranged into a tree: see Figure 1.1. Every argument is arranged as a quotient that should be read from the bottom up and whose line should be read as "because". In other words, the "numerator" is the reason why the "denominator" or conclusion holds. The reason for each step is stated at the end of each conclusion. It is often convenient to prove claims on inductive structure of such argument trees.

Working with the relation $=_a$ in this way is cumbersome and nothing like the arithmetic calculations we learned to perform in grade school. In practice, we skip transitions, are unaware of the reasons for individual steps, and do not need or perceive the tree shape of the overall argument. Once we consider extensions of SA that include more complicated programming facilities, we will need to rely on such formal definitions because the manipulations of program expressions are far more complicated than the manipulation of arithmetic expressions.

1.2 Semantics of SA

Evaluating an arithmetic expression means determining a numeral that is equal to the expression. We take this idea as the definition of an *evaluator* as a function from SA expressions to numerals.

Definition 1.2.1 ($eval_{sa}$) Let $N = \{\ulcorner n \urcorner \mid n \in \mathbb{Z}\}$ be the set of numerals. Then, the evaluation function maps SA expressions to numerals:

$$eval_{sa} : \left\{ \begin{array}{lcl} SA & \longrightarrow & N \\ M & \mapsto & \ulcorner m \urcorner \text{ if } M =_a \ulcorner m \urcorner \end{array} \right.$$

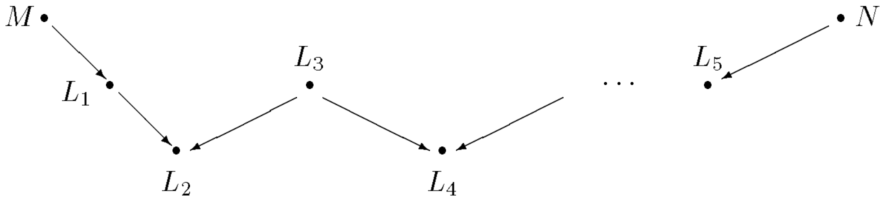
We also write $eval_{sa}(M) = \ulcorner m \urcorner$ to indicate that M has the result $\ulcorner m \urcorner$. ∎

Clearly, $eval_{sa}$ is a relation between SA expressions and numerals, but is it a function? For SA, it is trivial to see that $eval_{sa}$ produces at most one result per program. We only discuss the proof idea in detail to prepare the following chapters.

Fact 1.2.2 (Consistency) *For all programs M in SA, there is at most one numeral $\ulcorner m \urcorner$ such that $eval_{sa}(M) = \ulcorner m \urcorner$.*

To prove the theorem let us assume that a program M evaluates to the numerals $\ulcorner m \urcorner$ and $\ulcorner n \urcorner$. If $eval_{sa}$ is a function then the two numerals must be identical: $\ulcorner m \urcorner = \ulcorner n \urcorner$. By the definition of $eval_{sa}$, the assumption implies that $M =_a \ulcorner m \urcorner$ and $M =_a \ulcorner n \urcorner$. Hence, by the definition of $=_a$, $\ulcorner m \urcorner =_a \ulcorner n \urcorner$. To get from here to the conclusion that $\ulcorner m \urcorner = \ulcorner n \urcorner$, we must study the nature of calculations, that is, the general shape of proofs $M =_a N$. It is precisely for this reason that we need to have a precise and formal definition of the relation $=_a$, and in particular, the definitions of the intermediate relations that we called reductions.

Since **a**-equality is an extension of the one-step reduction for **a**, a calculation to prove $M =_a N$ is generally a series of one-step reductions based on **a** in both directions:



The question is whether these steps can be rearranged such that all reduction steps go from M to some L and from N to the same L. Formally, if $M =_a N$ then there should be an expression L such that $M \twoheadrightarrow_a L$ and $N \twoheadrightarrow_a L$.

If the claim about the rearrangements of equality proofs holds, the proof of consistency is finished. Recall that we have

$$\ulcorner m \urcorner =_a \ulcorner n \urcorner.$$

By the claim, there must be an expression L such that

$$\ulcorner m \urcorner \twoheadrightarrow_a L \qquad \text{and} \qquad \ulcorner n \urcorner \twoheadrightarrow_a L.$$

But numerals are clearly not reducible, i.e., there is no L such that $\ulcorner m \urcorner \longrightarrow_a L$. Therefore both numerals are identical to L and hence identical to each other:

$$\ulcorner m \urcorner = \ulcorner n \urcorner.$$

By the preceding argument we have reduced the proof of $eval_{sa}$'s consistency to a claim about the shape of arguments that prove $M =_a N$. 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 the λ-calculus. If an equality relation on terms generated from some basic notion of reduction satisfies this property, it is nowadays called "Church-Rosser."

Fact 1.2.3 (Church-Rosser) *If $M =_a N$ then there exists an expression L such that $M \twoheadrightarrow_a L$ and $N \twoheadrightarrow_a L$.*

Since the definition of **a**-equality is inductive, we can prove this fact by induction of the structure of the derivation of $M =_a N$. By case analysis, one of the following four conditions holds:

$M \longrightarrow_a N$: In this case the conclusion is immediate.

$M = N$: Again, the fact is trivially true.

$N =_a M$ holds, and therefore $M =_a N$: Now we know from the inductive hypothesis that there is an expression L such that $N \twoheadrightarrow_a L$ and $M \twoheadrightarrow_a L$. But this is precisely what we are looking for, namely, a term L to which the left-hand and the right-hand side of the equation reduces.

$M =_a L, L =_a N$ hold for some $L \in \text{SA}$, and therefore $M =_a N$: By the inductive hypothesis, there exists an expression L_1 such that $M \longrightarrow\!\!\!\!\rightarrow_a L_1$ and $L \longrightarrow\!\!\!\!\rightarrow_a L_1$ and an expression L_2 such that $N \longrightarrow\!\!\!\!\rightarrow_a L_2$ and $L \longrightarrow\!\!\!\!\rightarrow_a L_2$. In pictures we have:

$$\begin{tikzcd}
M \arrow[dr] & = & L \arrow[dl] \arrow[dr] & = & N \arrow[dl] \\
& L_1 & & L_2 &
\end{tikzcd}$$

Now suppose that for such an L that reduces to L_1 and L_2 there exists L_3 such that $L_1 \longrightarrow\!\!\!\!\rightarrow_a L_3$ and $L_2 \longrightarrow\!\!\!\!\rightarrow_a L_3$. Then, it is easy to finish the proof: simply take the term L_3 as the common reduct of M and N.

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:

$$\begin{tikzcd}
& L \arrow[dl] \arrow[dr] & \\
M \arrow[dr] & & N \arrow[dl] \\
& K &
\end{tikzcd}$$

Fact 1.2.4 (Diamond Property) *If $L \longrightarrow\!\!\!\!\rightarrow_a M$ and $L \longrightarrow\!\!\!\!\rightarrow_a N$ then there exists an expression K such that $M \longrightarrow\!\!\!\!\rightarrow_a K$ and $N \longrightarrow\!\!\!\!\rightarrow_a K$.*

Indeed, we can prove that the one-step reduction relation $\longrightarrow_a$ satisfies a diamond property:

> *If $L \longrightarrow_a M$ and $L \longrightarrow_a N$ with $M \neq N$ then there exists an expression K such that $M \longrightarrow_a K$ and $N \longrightarrow_a K$.*

Since the compatible closure of **a** is defined by induction on the structure of expressions, we use structural induction on L to prove the fact. We proceed by case analysis on the last step in the argument why $L \longrightarrow_a M$:

L **a** M: This case is only possible if $L = (1^+ \ulcorner l \urcorner)$, $L = (1^- \ulcorner l \urcorner)$, $L = (+ \ulcorner l \urcorner \ulcorner m \urcorner)$, or $L = (- \ulcorner l \urcorner \ulcorner m \urcorner)$. In each case, L can only be reduced in one way and hence $M = N$. Thus, the claim vacuously holds.

$L = (1^+ \ L_1), M = (1^+ \ M_1)$, **and** $L_1 \longrightarrow_a M_1$**:** Clearly, $N = (1^+ \ N_1)$. Since L_1 is a proper subterm of L, the inductive hypothesis applies, which means that there exists an expression K_1 such that $M_1 \longrightarrow_a K_1$ and $N_1 \longrightarrow_a K_1$. By the definition of the one-step reduction relation, $M \longrightarrow_a (1^+ \ K_1)$ and $N \longrightarrow_a (1^+ \ K_1)$. Hence, $K = (1^+ \ K_1)$.

$L = (1^- \ L_1), M = (1^- \ M_1)$, **and** $L_1 \longrightarrow_a M_1$**:** The argument in this case proceeds as for the preceding case.

$L = (+ \ L_1 \ L_2), M = (+ \ M_1 \ M_2), L_1 \longrightarrow_a M_1$**, and** $L_2 = M_2$**:** Now the argument depends on on which sub-expression of L must be reduced to reach N:

> $N = (+ \ N_1 \ N_2), L_1 \longrightarrow_a N_1$, **and** $L_2 = N_2$**:** Since L_1 is a proper subterm of L, the induction hypothesis applies: there exists an expression K_1 such that $M_1 \longrightarrow_a K_1$ and $N_1 \longrightarrow_a K_1$. We can therefore set $K = (+ \ K_1 \ L_2)$.
>
> $N = (+ \ N_1 \ N_2), L_2 \longrightarrow_a N_2$**, and** $L_1 = N_1$**:** Putting together the assumptions of the two nested cases, we can see that $(+ \ M_1 \ L_2) \longrightarrow_a (+ \ M_1 \ N_2)$ and $(+ \ L_1 \ N_2) \longrightarrow_a (+ \ M_1 \ N_2)$. Hence setting $K = (+ \ M_1 \ N_2)$ proves the claim.

other cases: The proofs for all other cases proceed as in the preceding case.

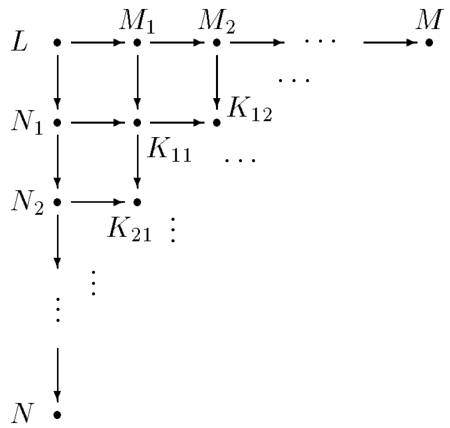
Now that we know that the one-step reduction satisfies a diamond property, it is easy to show that it is transitive-reflexive closure does. Assume that $L \longrightarrow\!\!\!\!\rightarrow_a M$ and $L \longrightarrow\!\!\!\!\rightarrow_a N$. By the inductive definition of the reduction relation $\longrightarrow\!\!\!\!\rightarrow_a$,

$$L \longrightarrow_a{}^m M \text{ and } L \longrightarrow_a{}^n N$$

for some $m, n \in \mathbb{N}$. Pictorially, we have

$$\begin{tikzcd}
L \bullet \arrow[r] \arrow[d] & \overset{M_1}{\bullet} \arrow[r] & \overset{M_2}{\bullet} \arrow[r] & \cdots \arrow[r] & \overset{M}{\bullet} \\
N_1 \bullet \arrow[d] & & & & \\
N_2 \bullet \arrow[d] & & & & \\
\vdots \arrow[d] & & & & \\
N \bullet & & & &
\end{tikzcd}$$

Using the diamond property for the one-step reduction, we can now fill in expressions $K_{1,1}$, $K_{2,1}$, $K_{1,2}$, etc. until the entire square is filled out:



Formally, this idea can also be cast as an induction but the diagramatic form of the argument is more intuitive.

Exercise 1.2.1 Given the diamond property for the one-step reduction relation, prove the diamond property for its transitive closure. ∎

The preceding arguments also show that $M =_a \ulcorner m \urcorner$ if and only if $M \longrightarrow\!\!\!\!\!\rightarrow_a \ulcorner m \urcorner$. 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 SA expressions. In the next chapter we will introduce a programming language for which such apparent backward steps truly shorten the calculation of the result of a program.

Exercise 1.2.2 After determining that a program has a unique value, the question arises whether a program always has a value. The answer is again positive, but only for SA. Prove this fact.

It follows this fact that there is an algorithm for deciding the equality of SA programs. Simply evaluate both programs and compare the results. Realistic programming languages include arbitrary non-terminating expressions and thus preclude a programmer from deciding the equivalence of expressions. ∎

Chapter 2

Functional ISWIM

Maninpulating primitive data is often a repetitive process. It is often necessary to apply the same sequence of primitive operations to different sets of data. To avoid this repetition and to abstract the details of such sequences of operations, programming languages offer the programmer a facility for defining procedures. Adding procedures to our simple language of arithmetic from the preceding chapter roughly corresponds to the ability to define functions on the integers. But once such functions are available we may also want to have functions like the numerical differentation operator that map functions to functions. In other words, a programming language should offer not only procedures that accept and return simple data but also procedures that manipulate procedures.

Church's λ-calculus was an early "programming language" that satisfied the demand for higher-order procedures. The purpose of the λ-calculus was to provide a formal system that isolated the world of functions from everything else in mathematics. Thus it contained nothing but functions and provided an "arithmetic of functions." Since procedures in programming languages correspond to functions, the system was a natural choice for Landin when he wanted to extend the language of arithmetic to a language with programmer-definable procedures.

In this chapter we introduce and analyze ISWIM. Its calculus and its semantics are far more complicated than that of SA. Following Landin, we leave the data sublanguage of ISWIM unspecified and only state minimal assumptions that are necessary for ISWIM's calculus to be consistent. **if** expressions

2.1 ISWIM Syntax

The syntax of the λ-calculus provides a simple, regular method for writing down functions such that they can play the role of functions as well as the role of inputs and outputs of functions. The specification of such functions 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

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

in the λ-calculus syntax we simply 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, named x, that it inputs.

To write down an application of a function f to an argument a, the calculus uses ordinary mathematical syntax modulo the placement of parentheses, *e.g.*,

$$(f\ a).$$

To mimic the regularity of λ-calculus's syntax, ISWIM adds three elements to the syntax of arithmetic expressions:

variables: to calculate with a datum, it is convenient to assign a name to it and to use the name to refer to the datum;

procedures: to create an algorithm for computing outputs from inputs, ISWIM provides λ or procedural abstractions: a procedure determines the name with which it wants to refer to its inputs and, through an expression that contains this name, or variable, it determines the output;

applications: supplying an argument datum to a procedure, happens when a procedure is applied to a datum.

Following Landin's proposal the formal definition of ISWIM's syntax does not make any assumptions about primitive data constants and procedures. Instead, it assumes that there are n-ary primitive procedures and basic constants whose interpretation will be specified eventually.

Definition 2.1.1 (*ISWIM Syntax* (Λ)) ISWIM expressions are expressions over an alphabet that consists of (,), ., and λ as well as variables, and constant and function symbols from the (unspecifed) sets *Vars*, *BConsts*, and $FConsts^n$ (for $n \in \mathbb{N}$). The sets *Vars*, *BConsts*, and *FConsts* are mutually disjoint.

The set of ISWIM expressions, called Λ, contains the following expressions:

$$M ::= x \mid (\lambda x.M) \mid (M\ M) \mid b \mid (o^n\ M \ldots M)$$

where

x	$\in$	*Vars*	an infinite supply of variable names
b	$\in$	*BConsts*	basic data constants
o^n	$\in$	$FConsts^n$, for $n \geq 1$	n-ary primitive functions

An expression $(o\ M_1\ \ldots\ M_n)$ is only legal if $o \in \mathit{FConsts}^n$.

The meta-variables x and many other lower-case letters will serve as variables ranging over *Vars* but also as actual members of *Vars*. In addition to the meta-variable M, K, L, N, and indexed and primed variants range over the set of expressions. In addition to o for function symbols and b for basic constants, we also use the meta-variable c to range over both sets of constants. ∎

We use conventional terminology for abstract syntax trees to refer to the pieces of complex expressions. If some expression N occurs in an expression M, it is a sub-expression. The variables of an expression M are all variables that occur in the expression; $\mathit{VAR}(M)$ denotes the set of variables in expression M. Immediate sub-expressions of a complex expression are named in accordance with their intended use. In a procedure $\lambda x.M$, x is the (*formal*) *procedure parameter* and M is the *procedure body*. The *function position* or the *function* (*expression*) in some application $(M\ N)$ refers to M; similarly, *argument position* and *argument* (*expression* refer to N. Finally, an application of the shape $(o\ M\ \ldots\ N)$ is a *primitive application.*

ISWIM expressions are difficult to read due to the absence of n-ary procedures and the resulting proliferation of λ's and parentheses. To facilitate our work with ISWIM expressions, we will often, but not always, use the following two conventions:

1. A λ-abstractions with multiple parameters is an abbreviation for nested abstractions, e.g.,
$$\lambda xyz.M = (\lambda x.(\lambda y.(\lambda z.M))).$$
This abbreviation is right-associative.

2. An n-ary application is an abbreviation for nested applications, *e.g.*,
$$(M_1\ M_2\ M_3) = ((M_1\ M_2)\ M_3).$$
This abbreviation is left-associative.

The set Λ is only a completely specified language once we define the sets of basic and functional constants. To obtain an ISWIM over SA, we would set

use of $p = \lambda xy.(o_p\ x\ y)$ as values

$$\begin{aligned} \mathit{BConsts} &= \{\ulcorner n \urcorner \mid n \in \mathbb{Z}\} \\ \mathit{FConsts}^1 &= \{1^+, 1^-\} \\ \mathit{FConsts}^2 &= \{+, -\}. \end{aligned}$$

A somewhat more realistic version would add further primitive operators (of arity 2), *e.g.*, $*$ (mulitplication), $/$ (division), $\uparrow$ (exponentiation), *etc*. Some Λ expressions over this extended SA language would be

$$(\lambda x.(+\ (\uparrow\ x\ \ulcorner 2 \urcorner)\ (*\ \ulcorner 3 \urcorner\ x))),$$

a polynomial over x, and

$$
\begin{array}{l}
(\lambda e. \\
\quad (\lambda f. \\
\quad\quad (\lambda x. \\
\quad\quad\quad (/ \quad (-\ (f\ (+\ x\ e))\ (f\ (-\ x\ e))) \\
\quad\quad\quad\quad\quad (*\ \ulcorner 2 \urcorner\ e))))),
\end{array}
$$

a family of numerical differential operators based on (relatively large) finite differences.

Since, as we have seen in the previous chapter, the notion of syntactic compatibility of expression relations plays a crucial role, we introduce it together with the syntax, independently of any specific expression relations. Not surprisingly, the notion of compatibility of relations with the syntactic constructions of Λ is similar to SA compatibility. In addition to compatibility with primitive applications, a relation between Λ expressions is Λ-compatible if it is applicable inside of procedure bodies and regular applications.

Definition 2.1.2 (Λ-*compatible closure of relations:* $\longrightarrow_r$) Let $\mathbf{r}$ be a binary relation over Λ. Then, the compatible closure of $\mathbf{r}$, notation: $\longrightarrow_r$, is a superset of $\mathbf{r}$ such that $M \longrightarrow_r M'$ implies

$$
\begin{array}{rcl}
(\lambda x.M) & \longrightarrow_r & (\lambda x.M') \\
(M\ N) & \longrightarrow_r & (M'\ N), \\
(N\ M) & \longrightarrow_r & (N\ M'), \\
(o^n\ N_1 \ldots N_i\ M\ N_{i+2} \ldots N_n) & \longrightarrow_r & (o^n\ N_1 \ldots N_i\ M'\ N_{i+2} \ldots N_n) \\
 & & \text{for } 0 \leq i \leq n
\end{array}
$$

for all $N, N_1, \ldots, N_n \in \Lambda$ and $x \in \mathit{Vars}$. ∎

Put abstractly, if a relation is compatible with the syntactic constructions of a language, it is possible to relate sub-expressions in arbitrary textual contexts. Since this notion of a context also plays an important role in the following sections and chapters, we introduce it now and show how to express the notion of compatibility based on it.

Definition 2.1.3 (*ISWIM Contexts*) Let $[\ ]$ (hole) be a new member of the alphabet distinct from variables, constants, λ, ., and (,).

The set of *contexts* is a set of ISWIM expressions with a hole:

$$C ::= [\ ] \mid (M\ C) \mid (C\ M) \mid (\lambda x.C) \mid (o^n\ M_1\ \ldots\ M_i\ C\ M_{i+1}\ \ldots\ M_n) \text{ for } 0 \leq i \leq n$$

where $M \in \Lambda$, $x \in \mathit{Vars}$. C, C', and similar letters range over contexts. To make their use as contexts explicit, we sometimes write $C[\ ]$.

Filling the hole ($[\ ]$) of a context $C[\ ]$ with the expression M yields an ISWIM expression, notation: $C[M]$. ∎

For example, $(\lambda x.[\ ])$ and $(\lambda x.(+\ [\ ]\ \ulcorner 1 \urcorner))$ are contexts over ISWIM based on SA. Filling the former with x yields the identity procedure; filling the latter results in a procedure that increments its inputs by one.

The connection between compatible closures and contexts is simple: $\mathbf{R}$ is compatible when $M\ \mathbf{R}\ N$ implies $C[M]\ \mathbf{R}\ C[N]$ and vice versa.

Proposition 2.1.4 *Let* $\mathbf{r}$ *be a binary relation on* Λ *and let* $\longrightarrow_r$ *be its compatible closure. Then, for expressions* M *and* N, $M \longrightarrow_r N$ *if and only if there are two expressions* M' *and* N' *and a context* $C[\ ]$ *such that* $M = C[M']$, $N = C[N']$, *and* $(M', N') \in \mathbf{r}$.

Exercise 2.1.1 Prove Proposition 2.1.4. ∎

2.2 Lexical Scope and α-Equivalence

Does it matter whether we write

$$(\lambda x.(+\ (\uparrow\ x\ \ulcorner 2 \urcorner)\ (*\ \ulcorner 3 \urcorner\ x)))$$

or

$$(\lambda y.(+\ (\uparrow\ y\ \ulcorner 2 \urcorner)\ (*\ \ulcorner 3 \urcorner\ y))),$$

i.e., is it important which variable we choose as the parameter of a procedure? Since both λ-expressions intuitively define the same polynomial function, it should not make any difference which one a programmer chooses to represent his intentions. The purpose of this section is to capture this simple but important equivalence relation between ISWIM expressions in a formal characterization. Equating such expressions will on one hand clarify the role of variables and on the other simplify working with the calculus for ISWIM.

An algorithmic method for obtaining the second from the first expression is to replace all occurrences of the parameter x with y. Unfortunately this simple idea does not really characterize the equivalence relationship that we are looking for. Consider the procedure

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

which intuitively absorbs an argument, names it x, and then returns a procedure that, upon application, applies its argument, y, to x. If we naïvely replaced the parameter x by y in this procedure, we would obtain

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

But the latter expression is a procedure that when applied to an argument returns a procedure that applies its argument to itself. In short, during the replacement process we confused two variables, and as a result the substitution disturbs relationships between a parameter and its occurrences in procedure bodies.

In order to resolve the problem, we need to introduce some terminology and notation so that we can discuss the situation in proper terms. The crucial notion is that of a free variable. A variable occurs *free* in an expression if it is not connected to a parameter. The function FV maps an expression to the set of its free variables:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M) - \{x\} \\ FV(M\ N) & = & FV(M) \cup FV(N) \\ FV(b) & = & \emptyset \\ FV(o\ M_1 \ldots M_n) & = & FV(M_1) \cup \ldots \cup FV(M_n) \end{array}$$

In contrast, a variable that is the parameter of some procedure is a *bound* variable. The occurrence of a parameter right behind a λ is the *binding occurrence*, the others are *bound occurrences*. When a variable x occurs free in some procedure body whose λ-header binds x, we say that *the former occurrence is in the lexical scope of the latter.*

closed expressions? Λ^0 def format?

Equipped with the new terminology, it is easy to state what went wrong with our second example above. When we replace parameter names we need to make sure that the new parameter does not occur free in the procedure body. That is, when x occurs in the lexical scope of some binding occurrence for x, then this should also hold after the replacement of parameter names. Put positively, when we replace a new variable to play the role of parameter in a procedure, we need to replace all *free* occurrences of the parameter in its lexical scope by a variable that does not occur in the procedure body. Based on this insight, it is easy to formulate the equivalence relation between expressions that ignores the identity of parameter names.

The preliminary step is to define variable substitution. We define it as a function by induction on the size of Λ expressions, according to the structure of the definition. When it is necessary to replace variables under λ-abstractions, we first rename the parameter of the procedure so that it does not interfere with the replacement variable or other free variables in the expression. Since any given term can only contain a finite number of λ-abstractions and *Vars* is infinite, it is always possible to pick such a new variable.

Definition 2.2.1 (*Variable Substitution*) Let $x_1, x_2, \ldots$ an ordered arrangement of all variables in *Vars*.

Variable substitution is a function that maps a Λ expression and two variables to a Λ expression. We use the infix notation $M[x \leftarrow y]$ to denote the replacement of free y by z in M:

$$\begin{array}{rcl} y[y \leftarrow z] & = & z \\ x[y \leftarrow z] & = & x \text{ if } x \neq y \\ (\lambda y.N)[y \leftarrow z] & = & \lambda y.N \end{array}$$

$$
\begin{aligned}
(\lambda x.N)[y \leftarrow z] &= \lambda x_i.N[x \leftarrow x_i][y \leftarrow z] \\
&\qquad \text{for the first } x_i \notin \{z\} \cup FV(N) \\
(L\ N)[y \leftarrow z] &= (L[y \leftarrow z]\ N[y \leftarrow z]) \\
b[y \leftarrow z] &= b \\
(o\ N_1 \ldots N_n)[y \leftarrow z] &= (o\ N_1[y \leftarrow z]\ \ldots\ N_n[y \leftarrow z])
\end{aligned}
$$

∎

The substitution function is well-defined because the size of $N[z \leftarrow y]$ is equal to the size of N. Given variable substitution, it is easy to define the desired congruence on Λ expressions.

Definition 2.2.2 (*α-equivalence*) The relation α relates Λ abstractions that only differ in their choice of parameter names:

$$(\lambda x.M) \quad \alpha \quad (\lambda y.M[x/y])$$

The α-equivalence relation $=_\alpha$ is the least equivalence relation that contains α and also is compatible with Λ ∎

Here are some examples of α-equivalences:

$$
\begin{aligned}
\lambda x.x &=_\alpha \lambda y.y \\
\lambda x.(\lambda y.x) &=_\alpha \lambda z.(\lambda w.z) \\
\lambda x.x(\lambda y.y) &=_\alpha \lambda y.y(\lambda y.y)
\end{aligned}
$$

To facilitate our work with Λ and its calculus, we adopt the convention of identifying all α-equivalent expressions. That is, Λ expressions in concrete syntax are representatives of the same α-equivalence class of expressions.

> **α-Equivalence Convention:**
>
> For Λ expressions M and N, $M = N$ means $M =_\alpha N$.

For the convention to make sense, we need to establish that operations on equivalence classes do not depend on the choice of representative. Thus far the only operations on expressions are syntactic constructions, which produce larger expressions from smaller ones, and variable substitution. It is trivial to check that these operations do not depend on the chosen representatives of α-equivalence classes.

Exercise 2.2.1 Let $M, M_1, \ldots, M_m$ and $N, N_1, \ldots, N_m$ be two series of expressions. Prove that if $M =_\alpha N, M_i =_\alpha N_i$, then

$$\lambda x.M =_\alpha \lambda x.N,\ (M_1\ M_2) =_\alpha (N_1\ N_2),\ \text{and}\ (o\ M_1\ \ldots\ M_m) =_\alpha (o\ N_1\ \ldots\ N_m).$$

Moreover, for all x and y,

$$\lambda x.M =_\alpha \lambda y.M[x \leftarrow y].$$

∎

In ISWIM, λ is the only construct that binds names. In other programming languages, there are many more language constructs that introduce binding occurrences into expressions. We will encounter some more binding constructs in latter chapters. For now we have developed a sufficient amount of notation and terminology for ISWIM to tackle the design of an ISWIM calculus.

2.3 The λ-Value Calculus

stuck state: (b V), (f V ...) if delta is undefined

Given the procedure $(\lambda x.(+\ (\uparrow\ x\ \ulcorner 2 \urcorner)\ (*\ \ulcorner 3 \urcorner\ x)))$ and the argument $\ulcorner 5 \urcorner$, it is easy to see how to compute the outcome of the application

$$((\lambda x.(+\ (\uparrow\ x\ \ulcorner 2 \urcorner)\ (*\ \ulcorner 3 \urcorner\ x)))\ \ulcorner 5 \urcorner)\ .$$

We replace the parameter name by the actual argument in the body of the procedure and continue to calculate with the modified body. In our example, the modified body is the SA expression

$$(+\ (\uparrow\ \ulcorner 5 \urcorner\ \ulcorner 2 \urcorner)\ (*\ \ulcorner 3 \urcorner\ \ulcorner 5 \urcorner))).$$

The rest of the calculation can now be carried out in an SA calculus with rules for exponentation ($\uparrow$).

two subsections: substitution beta-value

Thus, the general receipe for the evaluation of function applications seems to be quite simple. If $\lambda x.M$ is the procedure and N is the argument, simply substitute all free occurrences of x in M by N and evaluate the resulting expression. Unfortunately, this description is too simple. Consider the procedure $(\lambda x.\ulcorner 5 \urcorner)$. Applying it to $\ulcorner 3 \urcorner$ yields $\ulcorner 5 \urcorner$, indeed, applying it to any numeral results in $\ulcorner 5 \urcorner$. But what if the application is

$$((\lambda x.\ulcorner 5 \urcorner)\ (/\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner)) \quad ?$$

An ordinary mathematician would clearly answer that an expression like $(f\ (/\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner))$ does not make sense and should be considered an erroneous, no matter what the fucntion f does. Following this line of argument, Landin and many other language designers chose not to consider all expressions as proper arguments but only a certain subset, which we call *values*.

Put differently, procedures do not accept arbitrary expressions as arguments but only values. Before we can decide what the outcome of a procedure application is, we will need to know that the argument expression has a value otherwise we may inadvertently delete erroneous expressions like $(/\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner)$ from our programs during calculations. Dually, a procedure application will not produce arbitrary expressions but values only.

Since ISWIM is supposed to extend SA and similar programming languages with facilities for defining procedures on primitive data and procedures on procedures on primitive data *etc.*, we choose to consider all primitive constants *and* all procedures to be values. The revised rule of procedure application is then: a procedure application to a value has the same value as the procedure body after substituting all free occurrences of the parameter by the argument value. This rule also shows that variables are placeholders for values, and it is thus legitimate to include variables in the set of values.

A formalization of the rule for procedure application requires definitions of the set of values and of the substitution process. First, we define the set of values, formalizing the prceding argument.

Define programs to be closed expressions

Definition 2.3.1 (*Values* (*Vals*)) The set of ISWIM values, $Vals \subseteq \Lambda$, is the collection of basic constants ($b \in BConsts$), variables ($x \in Vars$), and λ-abstractions in Λ:

$$V ::= b \mid x \mid \lambda x.M.$$

In addition to V, U, V, W, X also range over values.
Also, $Vals^0 = \{V \in Vals \mid FV(V) = \emptyset\}$. ∎

The definition also expresses the idea that applications—of λ-expressions and primitive functions—are the only kind of expressions that require active calculations. The others are just plain values and do not need to be evaluated any further.

Next we need to generalize the substitution process from substitution of variables by variables to substitution of variables by values. It is straightforward to adapt Definition 2.2.1. In anticipation of latter sections and chapters, we define the substitution of variables by arbitrary expressions even though this general definition is not required for ISWIM.

Definition 2.3.2 (*Substitution*) Let $x_1, x_2, \ldots$ be an ordered arrangement of all variables in *Vars*.

Substitution is a function that maps a Λ expression, a variable, and another expressions to a Λ expression. We use the infix notation $M[x \leftarrow K]$ to denote the replacement of free y by K in M:

$$\begin{array}{rcl}
y[y \leftarrow K] & = & K \\
x[y \leftarrow K] & = & x \text{ if } x \neq y \\
(\lambda y.N)[y \leftarrow K] & = & \lambda y.N \\
(\lambda x.N)[y \leftarrow K] & = & \lambda x_i.N[x \leftarrow x_i][y \leftarrow K] \text{ if } x \neq y \\
 & & \quad \text{where } x_i = x \text{ if } x \notin FV(K), \\
 & & \quad \text{otherwise, for the first } x_i \notin FV(K) \cup FV(N)
\end{array}$$

$$\begin{array}{rcl}(L\ N)[y \leftarrow K] & = & (L[y \leftarrow K]\ N[y \leftarrow K]) \\ b[y \leftarrow K] & = & b \\ (o\ N_1 \dots N_n)[y \leftarrow K] & = & (o\ N_1[y \leftarrow K]\ \dots\ N_n[y \leftarrow K])\end{array}$$

∎

substituion commutes with alpha equiv convention on `M[x<-N]`

After clarifying the nature of arguments and the substitution process, we can finally define the basic rule for the evaluation of procedure applications. The laws is commonly known as the β-value axiom; we abbreviate this to β_v.

Definition 2.3.3 (β*-value* (β_v)) The relation β_v relates Λ applications to arbitrary ISWIM expressions:

$$((\lambda x.M)\ V) \quad \beta_v \quad M[x \leftarrow V]$$

where $M \in \Lambda$, $V \in \mathit{Vals}$, and $x \in \mathit{Vars}$. ∎

To complete the definition of the ISWIM calculus, we also need to deal with the application of primitive functions to arguments. To be consistent with the idea that ISWIM is parameterized over a primitive language for basic data, we only assume that the calculus for this language is defined via some function δ, which determines the result of an application of a primitive function to primitive data. To allow for some interaction between the two sub-languages, the result of a primitive application may be an arbitrary closed ISWIM value, *e.g.*, a λ-abstraction, which may then be applied to other arguments.

a more general delta function in Chapter x

Definition 2.3.4 (δ*-function*) A δ-function f for the set of function symbols $\bigcup_{n \in \mathbb{N}} \mathit{FConsts}^n$ is a family of partial $(n+1)$-ary functions, f^n, for each n where $\mathit{FConsts}^n \neq \emptyset$, such that f^n maps an n-ary functional constant and n basic constants to a closed value:

$$f^n : \mathit{FConsts}^n \times \underbrace{\mathit{BConsts} \times \ldots \times \mathit{BConsts}}_{n} \longrightarrow \mathit{Vals}^0.$$

If $f(o, b_1, \ldots, b_n)$ does not exist, we also say o is undefined for $b_1, \ldots, b_n$. ∎

SA's basic notion of reduction, $\mathbf{a}$, is a trivial example of a family of δ-functions. We use $\mathbf{a}^1$ for the subset of $\mathbf{a}$ that applies to unary operations and $\mathbf{a}^2$ for the subset of $\mathbf{a}$ the applies to binary operations. For the extension of SA with division, multiplication, and exponentiation, $\mathbf{a}^2$ would have to be extended to a relation $\mathbf{b}^2$ that covers the additional operations. If we also wanted a branching facility, we could introduce the basic function $\mathsf{zero?}$ and extend $\mathbf{a}^1$ to $\mathbf{b}^1$:

$$\begin{array}{lll}(\mathsf{zero?}\ \ulcorner 0 \urcorner) & \mathbf{b}^1 & \lambda xy.x \\ (\mathsf{zero?}\ \ulcorner l \urcorner) & \mathbf{b}^1 & \lambda xy.y\end{array}$$

Syntax

$$
\begin{array}{rcl}
M & ::= & x \mid (\lambda x.M) \mid (M\ M) \\
 & & \mid b \mid (o^1\ M) \mid (o^2\ M\ M) \\
\\
b & ::= & \ulcorner n \urcorner \qquad \text{for } n \in \mathbb{N} \\
o^1 & ::= & 1^+ \mid 1^- \mid \mathsf{zero?} \\
o^2 & ::= & + \mid - \mid * \mid / \mid \uparrow
\end{array}
$$

Syntactic Abbreviations

For all K, L, M,

$$(\mathsf{if0}\ \ K\ L\ M) = (((zero?\ K)\ (\lambda d.L)\ (\lambda d.M))(\lambda x.x)) \quad \text{where } d \notin FV(L) \cup FV(M)$$

δ-Function for Data Language

For $m, n, l \in \mathbb{Z}$, with $l \neq 0$,

$$
\begin{array}{rcl}
(1^+\ \ulcorner m \urcorner) & \mathbf{b}^1 & \ulcorner m+1 \urcorner \\
(1^-\ \ulcorner m \urcorner) & \mathbf{b}^1 & \ulcorner m-1 \urcorner \\
(\mathsf{zero?}\ \ \ulcorner 0 \urcorner) & \mathbf{b}^1 & \lambda xy.x \\
(\mathsf{zero?}\ \ \ulcorner l \urcorner) & \mathbf{b}^1 & \lambda xy.y \\
\\
(+\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m+n \urcorner \\
(-\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m-n \urcorner \\
(*\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m \cdot n \urcorner \\
(/\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m/l \urcorner \\
(\uparrow\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m^n \urcorner
\end{array}
$$

Figure 2.1: ISWIM over integer numerals and a basic set of numeric functions

For this concrete ISWIM language, the δ-function $\mathbf{b}$ is only undefined for division / and the arguments m and $\ulcorner 0 \urcorner$. Figure 2.1 summarizes the conventions for this version of ISWIM; we will make use of it below.

The β_v reduction and a δ function for the primitive data sub-language are the basis for the ISWIM calculus. Due to Plotkin's work on the relationship between programming languages and λ-calculi, it has become known as the λ_v-calculus.

Definition 2.3.5 (*The λ_v-calculus, $\boldsymbol{\lambda_v}$*) Let δ_n $(n \in \mathbb{N})$ be a family of δ-fucntions for

ISWIM.

The basic notion of reduction for the λ_v-calculus is the union of β_v and δ:

$$\mathbf{v} = \beta_v \cup \delta.$$

Following the conventions introduced by Definition 2.1.2, the *one-step* **v**-*reduction* $\longrightarrow_v$ is the compatible closure of **v**. The **v**-*reduction* is denoted by $\longrightarrow\!\!\!\!\rightarrow_v$ and is the reflexive, transitive closure of $\longrightarrow_v$; $=_v$ is the smallest congruence relation generated by $\longrightarrow_v$.

Notation: If $M_1 =_v M_2$, we also write $\boldsymbol{\lambda_v} \vdash M_1 = M_2$ (pronounced as "λ_v proves that M_1 equals M_2") to emphasize the fact that the calculus is an equational proof system. ∎

As in the case of our simple language SA, the evaluation of an ISWIM program requires a proof of equivalence between a program and a value. For ISWIM, the definition is complicated by the presence of values that are procedures. Consider the program

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

It is clearly equivalent to the two distinct values

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

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 ISWIM and basically all other practical programming systems based on ISWIM implement: ISWIM's *eval* function only returns a special token if a program is equal to a functional value.

Definition 2.3.6 (*eval$_v$*) Let the set of ISWIM answers be the basic constants plus the symbol `closure`:

$$A = BConsts \cup \{\texttt{closure}\}.$$

The partial function $eval_v : \Lambda^0 \longrightarrow A$ is defined as follows:

$$eval_v(M) = \begin{cases} b & \text{if } M =_v b \\ \texttt{closure} & \text{if } M =_v \lambda x.N \end{cases}$$

If $eval_v(M)$ does not exist, we say *eval* diverges on M, or, M diverges. ∎

Since the calculus of procedures is much less familiar than the calculus of arithmetic, we define a concrete instance of ISWIM in the next section and show how to work with the resulting λ_v-calculus. After this interlude on working *in* the calculus, we will return to the study of the properties of the calculus.

Exercise 2.3.1 Implement $eval_v$ in your favorite programming language. ∎

2.4 Using the Calculus

Programming in a procedure-oriented language like ISWIM is well-suited for encoding mathematical relationships and exercises. For an example of this kind of programming, recall the family of differential operators based on finite differences that were defined in Section 2.1:

$$\begin{array}{ll} D = & (\lambda e. \\ & \quad (\lambda f. \\ & \quad\quad (\lambda x. \\ & \quad\quad\quad (/ \;\; (-\ (f\ (+\ x\ e))\ (f\ (-\ x\ e))) \\ & \quad\quad\quad\quad\quad (*\ \ulcorner 2 \urcorner\ e))))), \end{array}$$

The underlying data language is an extension of SA based on the δ-fucntion **b** defined at the end of the preceding section.

Applying D to an epsilon value, say $\ulcorner 1 \urcorner$, yields an instance of a differential operator:

$$\begin{array}{ll} (D\ \ulcorner 1 \urcorner) =_v & (\lambda f. \\ & \quad (\lambda x. \\ & \quad\quad (/ \;\; (-\ (f\ (+\ x\ 1))\ (f\ (-\ x\ 1))) \\ & \quad\quad\quad\quad (*\ \ulcorner 2 \urcorner\ 1)))), \end{array}$$

A further simplification is possible, due to the δ-function:

$$\begin{array}{ll} \ldots =_v & (\lambda f. \\ & \quad (\lambda x. \\ & \quad\quad (/ \;\; (-\ (f\ (+\ x\ 1))\ (f\ (-\ x\ 1))) \\ & \quad\quad\quad\quad \ulcorner 2 \urcorner))), \end{array}$$

Programming algorithms that deal with inductively defined sets of data is another rich area of examples for functional programming. Although ISWIM does not have a facility for defininig recursive procedures, it is still posible to do so due to the device of self-application, a trick that was already known to Church. Using self-application, it is possible to define a procedure that can create a recursive procedure from a non-recursive one. To understand the principle behind this interesting procedure and how it solves the problem of defining recursive procedures, we need to discuss the nature of recursive definitions. The essence of a recursive procedure definition is the ability of the function to refer to itself. A self-reference can easily be accomplished via names: we simply write down a name on one side of an equality symbol and the procedure definition, which uses the name, on the other:

$$f = (\lambda x. \ldots f \ldots f \ldots f).$$

What does such an equation mean? An algebraic reading based on our ISWIM calculus says that this equation specifies an unknown quantity, F, and that a solution could be an ISWIM expression that satisfies the equation.

To solve the equation, we first rewrite it using β_v such that the name of the recursive procedure only occurs once on the right side of the equation and we swap the two sides of the equation:

$$((\lambda g.(\lambda x.\ldots g\ldots g\ldots g))\ f) = f.$$

It is now obvious that an ISWIM expression that can play the role of f is a *fixed point* of the procedure

$$(\lambda g.(\lambda x.\ldots g\ldots g\ldots g)).$$

The λ-calculus solution to this problem is to define a procedure that can find fixed points for all such definitions. The ISWIM version is defined as follows:

$$\begin{aligned}\mathsf{Y}_v = \ & (\lambda f. \\ & \quad (\lambda x. \\ & \quad\quad (\ (\lambda g.\ (f\ (\lambda x.\ ((g\ g)\ x)))) \\ & \quad\quad\quad (\lambda g.\ (f\ (\lambda x.\ ((g\ g)\ x))))\ x))),\end{aligned}$$

i.e., applying Y_v to a procedure of the shape $(\lambda gx.\ldots g\ldots g\ldots g)$ yields a solution for the above equation.

Proposition 2.4.1 (Fixed Point Theorem) *For all $K = \lambda gx.L$,*

$$\boldsymbol{\lambda_v} \vdash (K\ (\mathsf{Y}_v\ K)) = K.$$

Proof. The proof is a straightforward calculation where all of the basic proof steps are β_v-steps:

$$\begin{aligned}\boldsymbol{\lambda_v} \vdash \quad & (\mathsf{Y}_v\ K) \\ = \quad & ((\lambda f.\lambda x.(((\lambda g.(f\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(f\ (\lambda x.((g\ g)\ x))))\ x))\ K) \\ = \quad & \lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x) \\ = \quad & \lambda x.((K\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\ = \quad & \lambda x.(((\lambda gx.L)\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\ = \quad & \lambda x.((\lambda x.L[g \leftarrow (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x))])\ x) \\ = \quad & \lambda x.L[g \leftarrow (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))] \\ = \quad & ((\lambda gx.L)\ (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))) \\ = \quad & (K\ (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))) \\ = \quad & (K\ (\mathsf{Y}_v\ K)). \quad \blacksquare\end{aligned}$$

The proof of the Fixed Point Theorem looks overly complicated partly because arguments to procedures must be values for β_v to apply. Thus instead of calculating

$$\begin{array}{rl}\boldsymbol{\lambda_v} \vdash & \lambda x.((K\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\ = & \lambda x.((K\ (\mathsf{Y}_v\ K))\ x) \\ = & \lambda x.((\lambda gx.L)\ (\mathsf{Y}_v\ K))\ x),\end{array}$$

we need to carry around the full value that $(\mathsf{Y}_v\ K)$ evaluates to. To avoid this complication in future calculations, we prove 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.

Lemma 2.4.2 *Let $M \in \Lambda$, $V \in$ Values. If $M =_v V$ then for all $\lambda x.N$,*

$$((\lambda x.N)\ M) =_v N[x \leftarrow M].$$

Proof. The initial portion of the proof is a simple calculation:

$$((\lambda x.N)\ M) =_v ((\lambda x.N)\ V) =_v N[x \leftarrow V] =_v N[x \leftarrow M].$$

The last step in this calculation, however, needs a sparate proof by induction on N, showing that $N[x \leftarrow M] =_v N[x \leftarrow L]$ if $M =_v L$:

$N =$	by inductive hypo.	
c		$c[x \leftarrow M] = c = c[x \leftarrow L]$
x		$x[x \leftarrow M] = M =_v L = x[x \leftarrow L]$
$y \neq x$		$y[x \leftarrow M] = y = y[x \leftarrow L]$
$\lambda y.N'$	$N'[x \leftarrow M] =_v N'[x \leftarrow L]$	$(\lambda y.N')[x \leftarrow M]$
		$=_v (\lambda y.N'[x \leftarrow M])$
		$=_v (\lambda y.N'[x \leftarrow L])$
		$=_v (\lambda y.N')[x \leftarrow L]$
$(N_1\ N_2)$	$N_i[x \leftarrow M] =_v N_i[x \leftarrow L]$	$(N_1\ N_2)[x \leftarrow M]$
		$=_v (N_1[x \leftarrow M]\ N_2[x \leftarrow M])$
		$=_v (N_1[x \leftarrow L]\ N_2[x \leftarrow L])$
		$=_v (N_1 N_2)[x \leftarrow L]$

∎

We illustrate the definition and use of recursive procedures in the ISWIM version of Figure 2.1. For convenience, we introduce an if-expression in the tradition of Lisp as an abbreviation:

$$(\mathsf{if0}\ \ M\ N\ L) = ((\mathsf{zero?}\ \ M)\ (\lambda d.N)\ (\lambda d.L)\ (\lambda x.x))$$

for $d \notin FV(N) \cup FV(L)$. It is easy to verify that the expected equations hold for if0 expressions.

Proposition 2.4.3 *For all M and N,*

$$\begin{aligned} \boldsymbol{\lambda_v} \vdash (\mathsf{if0}\ \ulcorner 0 \urcorner\ M\ N) &= M \\ \boldsymbol{\lambda_v} \vdash (\mathsf{if0}\ \ulcorner m \urcorner\ M\ N) &= N \quad \textit{for all } m > 0. \end{aligned}$$

Proof. We only prove the first equation, leaving the second one as a simple exercise:

$$\begin{aligned} \boldsymbol{\lambda_v} \vdash \quad & (\mathsf{if0}\ \ulcorner 0 \urcorner\ M\ N) \\ = \ & ((zero?\ulcorner 0 \urcorner)\ (\lambda d.M)\ (\lambda d.N)(\lambda x.x)) \\ & \text{ where } d \notin FV(M) \cup FV(N) \\ = \ & ((\lambda xy.x)\ (\lambda d.M)\ (\lambda d.N)(\lambda x.x)) \\ = \ & ((\lambda d.M)(\lambda x.x)) \\ = \ & M[d \leftarrow (\lambda x.x)] \\ = \ & M \end{aligned}$$

because $d \notin FV(M)$. ∎

To demonstrate the power of the calculus and its shortcomings, we begin our short series of recursive programming examples with a programmer-defined version of addition for positive numbers. The addition function takes two arguments. If the first one is $\ulcorner 0 \urcorner$, it can return the second one. Otherwise, it can recursively add the predecessor of its first argument and the second argument and return the successor of this result. Equationally, the specification is simple:

$$a = \lambda xy.(\mathsf{if0}\ \ x\ y\ (1^+\ (a\ (1^-\ x)y))).$$

In other words, we have a recursive specification, whose solution is the fixed point of the procedure

$$\begin{aligned} P \ = \ & \lambda a. \\ & \quad \lambda xy. \\ & \quad\quad (\mathsf{if0}\ \ x\ y\ (1^+\ (a\ (1^-\ x)\ y))). \end{aligned}$$

And indeed, this programmer-defined version of addition behaves just like the built-in version of addition on positive integers.

Proposition 2.4.4 *For $m, n \geq 0$,*

$$\boldsymbol{\lambda_v} \vdash ((\mathsf{Y}_v P)\ \ulcorner m \urcorner\ \ulcorner n \urcorner) = \ulcorner m + n \urcorner = (+\ \ulcorner m \urcorner\ \ulcorner n \urcorner).$$

Proof. The proof proceeds by induction on m. Thus assume $m = 0$. Then,

$$\begin{aligned} \boldsymbol{\lambda_v} \vdash ((\mathsf{Y}_v P)\ \ulcorner m \urcorner\ \ulcorner n \urcorner) &= ((\mathsf{Y}_v P)\ \ulcorner 0 \urcorner\ \ulcorner n \urcorner) \\ &= ((P\ (\mathsf{Y}_v\ P))\ \ulcorner 0 \urcorner\ \ulcorner n \urcorner) \end{aligned}$$

$$\begin{aligned}
&= ((\lambda xy.(\mathsf{if0}\ \ x\ y\ (1^+\ ((\mathsf{Y}_v\ P)\ (1^-\ x)\ y))))\ \ulcorner 0\urcorner\ \ulcorner n\urcorner)\\
&= (\mathsf{if0}\ \ \ulcorner 0\urcorner\ \ulcorner n\urcorner\ (1^+\ ((\mathsf{Y}_v\ P)\ (1^-\ \ulcorner 0\urcorner)\ \ulcorner n\urcorner)))\\
&= \ulcorner n\urcorner\\
&= \ulcorner m+n\urcorner
\end{aligned}$$

If $m \geq 0$, then

$$\begin{aligned}
\boldsymbol{\lambda_v} \vdash ((\mathsf{Y}_v P)\ \ulcorner m\urcorner\ \ulcorner n\urcorner) &= ((\mathsf{Y}_v P)\ \ulcorner 0\urcorner\ \ulcorner n\urcorner)\\
&= (1^+\ ((\mathsf{Y}_v\ P)\ (1^-\ \ulcorner m\urcorner)\ \ulcorner n\urcorner))\\
&= (1^+\ ((\mathsf{Y}_v\ P)\ \ulcorner m-1\urcorner\ \ulcorner n\urcorner))\\
&= (1^+\ \ulcorner m-1+n\urcorner))\ \text{by inductive hypothesis}\\
&= \ulcorner m+n\urcorner.\ \blacksquare
\end{aligned}$$

The reasoning about ISWIM programs is typical. The statement of the proposition is expressed in the meta-mathematical language; similarly, the indcutive reasoning is carried out in the meta-mathematical language, not in the calculs itself. On the other hand, the proposition also indicates what the calculus *cannot* prove. While the proposition says that $(\mathsf{Y}_v\ P)$ and $+$ produce the same outputs for the same inputs, it does not prove that the built-in addition operation and $(\mathsf{Y}_v\ P)$ are identical:

$$\boldsymbol{\lambda_v} \not\vdash (\lambda xy.(+\ x\ y)) = (\mathsf{Y}_v\ P).$$

We will return to this lack of power at the end of the chapter. + vs lambda xy.+ x y

To clarify how self-application leads to interesting phenomenon, consider the expression

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

It applies the procedure $(\lambda x.(x\ x))$ to itself, which in turn applies its argument to itself. Ω is an application and it is possible to reduce the application via β_v. But alas, nothing happens:

$$\Omega \longrightarrow_v \Omega,$$

Ω only reduces to itself. Hence, we have found a first program that is not equal to a value, and that therefore causes the evaluator to diverge.

2.5 Consistency

Following the tradition of simple arithmetic, 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 SA's consistency theorem for ISWIM's $eval_v$. This theorem, far more complex than the one for SA, is the subject of this section. Its proof follows the same outline as the proof of the SA consistency theorem. We start with the basic theorem assuming a Church-Rosser property for ISWIM's calculus.

Theorem 2.5.1 (Consistency) *The relation $eval_v$ is a partial function.*

Proof. Let us assume that the Church-Rosser Property holds, that is, if $M =_v N$ then there exists an expression L such that $M \longrightarrow\!\!\!\!\!\rightarrow_v L$ and $N \longrightarrow\!\!\!\!\!\rightarrow_v L$.

Assume $eval_v(M) = a_1$ and $eval_v(M) = a_2$ for answers a_1, a_2. We need to show that $a_1 = a_2$. Based on the definition of ISWIM answers, we distinguish two cases:

$a_1, a_2 \in BConsts$**:** It follows from Church-Rosser that two basic constants are provably equal if and only if they are identical since both are not reducible.

normal form

$a_1 =$ `closure`**,** $a_2 \in BConsts$**:** By the definition of $eval_v$,

$$M =_v a_2 \text{ and } M =_v \lambda x.N \text{ for some } N.$$

Here, $a_2 =_v \lambda x.N$. Again by Church-Rosser and the irreducibility of constants,

$$\lambda x.N \longrightarrow\!\!\!\!\!\rightarrow_v a_2.$$

But by definition of the reduction relation $\longrightarrow\!\!\!\!\!\rightarrow_v$, $\lambda x.N \longrightarrow\!\!\!\!\!\rightarrow_v K$ 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.

These are all possible cases, and hence, it is always true that $a_1 = a_2$: $eval_v$ is a function. ∎

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

Theorem 2.5.2 (Church-Rosser for λ_v-calculus) *Let M, N be Λ expressions. If $M =_v N$ then there exists an expression L such that $M \longrightarrow\!\!\!\!\!\rightarrow_v L$ and $N \longrightarrow\!\!\!\!\!\rightarrow_v L$.*

Proof. The proof is basically a replica of the proof of SA's Church-Rosser theorem. It assumes a diamond property for the reduction relation $\longrightarrow\!\!\!\!\!\rightarrow_v$. ∎

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

Theorem 2.5.3 (Diamond Property for $\longrightarrow\!\!\!\!\!\rightarrow_v$) *Let L, M, and N be ISWIM expressions. If $L \longrightarrow\!\!\!\!\!\rightarrow_v M$ and $L \longrightarrow\!\!\!\!\!\rightarrow_v N$ then there exists an expression K such that $M \longrightarrow\!\!\!\!\!\rightarrow_v K$ and $N \longrightarrow\!\!\!\!\!\rightarrow_v K$.*

For SA the diamond property holds for the single-step relation, from which the diamond property for the transitive closure easily follows. On the other hand, it is almost obvious that the diamond property *cannot* hold for λ_v-calculus. 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 reducing either one of them we get the expressions

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

While both expressions contain redexes, the diagram in Figure 2.2 shows that the one-step reductions starting from these expressions do not lead to a common expression. Hence, the one-step relation does not satisfy the diamond property.

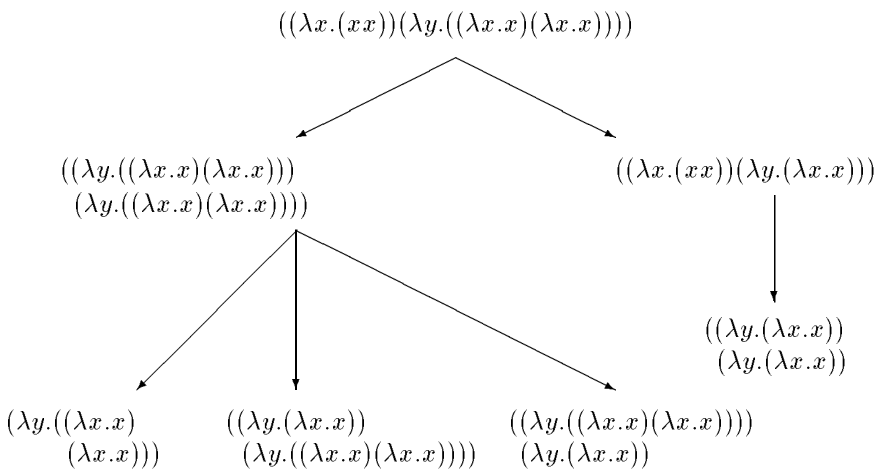


Figure 2.2: A Failure of the Diamond Property for Onestep **v** Reductions

Since the problem of the one-step relation based on **v** is caused by reductions that multiply redexes, it is quite natural to consider an extension of the one-step reduction that contracts several non-overlapping redexes in parallel. If this extension satisfies the diamond property and its transitive-reflexive closure is the same as that of the one-step relation, then we should be able to prove the diamond theorem for the **v**-reduction.

Definition 2.5.4 (*Parallel* **v**-*reduction:* $\underset{1}{\twoheadrightarrow}$) The parallel closure of **v**, notation: $\underset{1}{\twoheadrightarrow}$, is a superset of **v** such that

1. $M \underset{1}{\twoheadrightarrow} M$;
2. $(o\ b_1 \ldots b_n) \underset{1}{\twoheadrightarrow} \delta(o, b_1, \ldots, b_n)$ if δ is defined on o and $b_1, \ldots, b_n$;
3. $((\lambda x.M)\ U) \underset{1}{\twoheadrightarrow} N[x \leftarrow V]$ if $M \underset{1}{\twoheadrightarrow} N$ and $U \underset{1}{\twoheadrightarrow} V$ (for all x);
4. $(M\ M') \underset{1}{\twoheadrightarrow} (N\ N')$ if $M \underset{1}{\twoheadrightarrow} N$ and $M' \underset{1}{\twoheadrightarrow} N'$;
5. $(\lambda x.M) \underset{1}{\twoheadrightarrow} (\lambda x.N)$ if $M \underset{1}{\twoheadrightarrow} N$;
6. $(o\ M_1\ \ldots\ M_m) \underset{1}{\twoheadrightarrow} (o\ M_1'\ \ldots\ M_m')$ if for all i, $M_i \underset{1}{\twoheadrightarrow} M_i'$.

∎

And indeed, the parallel reduction does satisfy the diamond property.

Lemma 2.5.5 (Diamond Property for $\underset{1}{\twoheadrightarrow}$) *Let L, M, and N be SA expressions. If $L \underset{1}{\twoheadrightarrow} M$ and $L \underset{1}{\twoheadrightarrow} N$ then there exists an expression K such that $M \underset{1}{\twoheadrightarrow} K$ and $N \underset{1}{\twoheadrightarrow} K$.*

Proof. The proof is an induction on the tree structure of the proof that $L \underset{1}{\twoheadrightarrow} M$ and proceeds by case analysis on the last step in this proof:

$L = M$: Set $K = N$.

$(o\ b_1 \ldots b_n) \underset{1}{\twoheadrightarrow} \delta(o, b_1, \ldots, b_n)$: Since δ is a function there is no other possible reduction for L, *i.e.*, $L = N = M$.

$((\lambda x.L')\ U) \underset{1}{\twoheadrightarrow} M'[x \leftarrow V]$ because $L' \underset{1}{\twoheadrightarrow} M'$, $U \underset{1}{\twoheadrightarrow} V$: There are two possiblities for the second reduction path. Either the two parts of the application reduce separately or the application also reduces the β_v-redex:

> **$((\lambda x.L')\ U) \underset{1}{\twoheadrightarrow} ((\lambda x.Y')\ W)$ because $L' \underset{1}{\twoheadrightarrow} N'$, $U \underset{1}{\twoheadrightarrow} W$:** In this case, L', M', and N' satisfy the antecedent of the hypothesis and so do U, V, and W. Hence, there are expressions K' and X that complete the upper half of these two diamons. If we also knew that substitution and parallel reduction commute, we could conclude that $K = K'[x \leftarrow X]$ was the desired expression because then $((\lambda x.N')\ W) \underset{1}{\twoheadrightarrow} K'[x \leftarrow X]$ and $M'[x \leftarrow V] \underset{1}{\twoheadrightarrow} K'[x \leftarrow X]$.
>
> **$((\lambda x.L')\ U) \underset{1}{\twoheadrightarrow} N'[x \leftarrow W]$ because $L' \underset{1}{\twoheadrightarrow} N'$, $U \underset{1}{\twoheadrightarrow} W$:** As in the first subcase, L', M', and N' and U, V, and W determine upper halves of diamonds, respectively. By induction hypothesis, each yields two lower halves of diamonds and related terms K' and X. And again, if substitution and parallel reduction commute, setting $K = K'[x \leftarrow X]$ 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 2.5.6.

$(L_1\ L_2) \underset{1}{\twoheadrightarrow} (M_1\ M_2)$ because $L_i \underset{1}{\twoheadrightarrow} M_i$**:** This case is symmetric to the previous case. The proof of the preceding case can easily be adapted.

$(o\ L_1 \ldots L_m) \underset{1}{\twoheadrightarrow} (o\ M_1 \ldots M_m)$ because $L_i \underset{1}{\twoheadrightarrow} M_i$ **for** $1 \leq ileqm$**:** Two cases:

1. $L_1, \ldots, L_m$ are basic constants, N is the δ-contractum of L. Then $M = L$ and $K = N$.

2. $N = (o\ N_1 \ldots N_m)$ An n-fold use of the inductive hypothesis yields the conclusion.

$(\lambda x.L') \underset{1}{\twoheadrightarrow} (\lambda x.M')$ because $L' \underset{1}{\twoheadrightarrow} M'$**:** First, it is clear that $N = (\lambda x.N')$. Second, the induction hypothesis applies to L', M', and N' and yields an expression K' so that we can set $K = (\lambda x.K')$. ∎

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 2.5.6 *If* $M \underset{1}{\twoheadrightarrow} N$ *and* $U \underset{1}{\twoheadrightarrow} V$ *then* $M[x \leftarrow U] \underset{1}{\twoheadrightarrow} N[x \leftarrow V]$.

Proof. The proof is an induction on M and proceeds by case analysis on the last step in the proof of $M \underset{1}{\twoheadrightarrow} N$:

$M = N$**:** In this special case, the claim says that if $U \underset{1}{\twoheadrightarrow} V$ then $M[x \leftarrow U] \underset{1}{\twoheadrightarrow} M[x \leftarrow V]$. The proof of this specialized claim is an induction on the structure of M. It proceeds in the same manner as the proof Lemma 2.4.2.

$M = (o\ b_1 \ldots b_n), N = \delta(o, b_1, \ldots, b_n)$**:** Here, M and N are closed, so $M[x \leftarrow U] = M$, $N[x \leftarrow V] = N$. The result follows by the assumption of the case.

$M = ((\lambda y.K)\ W), N = L[y \leftarrow W']$ because $K \underset{1}{\twoheadrightarrow} L$**,** $W \underset{1}{\twoheadrightarrow} W'$**:** A simple calculation shows that the claim holds:

$$\begin{array}{rcll} M[x \leftarrow U] & = & ((\lambda y.K[x \leftarrow U])\ W[x \leftarrow U]) & \\ & \underset{1}{\twoheadrightarrow} & L[x \leftarrow V][y \leftarrow W'[x \leftarrow V]] & \\ & = & L[y \leftarrow W'][x \leftarrow V] & (\dagger) \\ & = & N[x \leftarrow V] & \end{array}$$

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

$M = (M_1\ M_2), N = (N_1\ N_2)$ because $M_i \twoheadrightarrow_1 N_i$: By the induction hypothesis, $M_i[x \leftarrow U] \twoheadrightarrow_1 N_i[x \leftarrow V]$ for both $i = 1$ and $i = 2$. Thus,

$$\begin{array}{rcl} M[x \leftarrow U] & = & (M_1[x \leftarrow U]\ M_2[x \leftarrow U]) \\ & \twoheadrightarrow_1 & (N_1[x \leftarrow V]\ N_2[x \leftarrow V]) \\ & = & (N_1\ N_2)[x \leftarrow V] \\ & = & N[x \leftarrow V] \end{array}$$

$M = (o\ M_1 \ldots M_m), N = (o\ M_1' \ldots M_m')$ because $M_i \twoheadrightarrow_1 N_i$: The proof of this case is an appropriate adaptation of that of the previous case.

$M = (\lambda x.K), N = (\lambda x.L)$ because $K \twoheadrightarrow_1 L$: The proof of this case is an appropriate adaptation of that of the previous case. ∎

Exercise 2.5.1 Complete Case 1 of Lemma 2.5.6. ∎

Exercise 2.5.2 Prove that if $y \notin FV(L)$ then

$$K[x \leftarrow L][y \leftarrow M[x \leftarrow L]] = K[y \leftarrow M][x \leftarrow L].$$

(Recall that $M = N$ means $M =_\alpha N$.) ∎

2.6 Observational Equivalence: The Meaning of Equations

Besides program evaluation the transformation of programs plays an important role in the practice of programming and programming language implementation. For example, if M is a procedure and 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, the λ_v-calculus clearly provides a solution: If both expressions were programs and we knew $M =_v N$, then $eval_v(M) = eval_v(N)$ 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 programm can only *observe* the output and is primarily interested in the output. He 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 magically produce some value. 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.

Definition 2.6.1 (*Observational Equivalence*) Two expressions, M and M', are *observationally equivalent*, written: $M \simeq_v M'$, if and only if they are indistinguishable in all contexts C such that $C[M]$ and $C[M']$ are programs,

$$eval_v(C[M]) = eval_v(C[M']).$$

∎

Observational equivalence obviously extends program equivalence: If programs M and N are observationally equivalent, then they are programs in the empty context and $eval_v(M) = eval_v(N)$. But this is also the least we should expect from a theory of expressions with equivalent functionality. In addition, and as the name already indicates, observational equivalence is an equivalence relation. Finally, if two expressions are observationally equivalent, then embedding the expresssions 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.

Proposition 2.6.2 *For all ISWIM expressions K, L, M,*

1. $M \simeq_v M$;
2. *if* $L \simeq_v M$ *and* $M \simeq_v N$ *then* $L \simeq_v N$;
3. *if* $L \simeq_v M$ *then* $M \simeq_v L$; *and*
4. *if* $M \simeq_v N$ *then* $C[M] \simeq_v C[N]$ *for all contexts* C.

Proof. Points 1 through 3 are trivial. As for point 4, assume $M \simeq_v N$ and let $C'[\ ]$ be an arbitrary context such that $C'[C[M]]$ and $C'[C[N]]$ are programs. Then, $C'[C[\ ]]$ is a program context for M and N. By assumption $M \simeq_v N$ and therefore,

$$eval_v(C'[C[M]]) = eval_v(C'[C[N]]),$$

which is what we had to prove. ∎

It turns out that observational equivalence is the largest equivalence relation, that is, the one that equates the most expressions, and still satisfies the general criterion that we outlined above.

Proposition 2.6.3 *Let $\equiv$ be a congruence relation such that $M \equiv N$ for programs M, N implies $eval_v(M) = eval_v(N)$. If $M \equiv N$ then $M \simeq_v N$.*

Proof. We assume $M \not\simeq_v N$ and show $M \not= N$. By assumption there exists a separating context C. That is, $C[M]$ and $C[N]$ are programs but $eval_v(C[M]) \neq eval_v(C[N])$. Therefore, $C[M] \not= C[N]$. But now by the assumptions about the congruence relation, $M \not= N$. ∎

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 the λ_v-calculus is sound with respect to observational equivalence. Unfortunately, it is also incomplete, that is, the calculus cannot *prove* all observational equivalence relations.

Theorem 2.6.4 (Soundness, Incompleteness) *If* $\boldsymbol{\lambda_v} \vdash M = M'$ *then* $M \simeq_v M'$. *The inverse direction does not hold.*

Proof. By definition, $\boldsymbol{\lambda_v}$ is a congruence relation. Moreover, $eval_v$ is defined based on $\boldsymbol{\lambda_v}$ such that if $\boldsymbol{\lambda_v} \vdash M = N$ and both are programs, then $eval_v(M) = eval_v(N)$.

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. Both terms diverge and are observationally equivalent. But both reduce to themselves only and therefore cannot be provably equal in $\boldsymbol{\lambda_v}$.

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 the calculus. ∎

Notes

Landin: design of iswim

Plotkin: design and analysis of iswim calculus

Curry: good treatment of substitution

Despite claims to the contrary in the literature, naïve substitution does *not* capture the notion of dynamic binding in Lisp. Cartwright [] offers an explanation.

Barendregt: comprehensive study of Church's λ-calculus as a logical system; numerous techniques applicable to this calculus though the book does not cover lcal as a calculus for a programming language conventions on treatment of terms!

Chapter 3

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. He can pursue any strategy for such a calculation that he finds profitable and can switch 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 β_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 $eval_v$ 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 along too many different reduction paths.

The solution of the problem is to find a strategy that reduces a program to an answer if it is equal to one. In other words, given a program, it is possible to pick a redex according to a fixed strategy such that the reduction of this redex brings the evaluation one step closer to the result if it exists. 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 basically 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. Still, the textual machine provides the foundation for a first implementation of the evaluator. A systematic elimination of inefficiencies and overlaps naturally leads to machines with more efficient representations of intermediate states. 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.

Background Reading: Plotkin, Felleisen & Friedman

3.1 Standard Reductions

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

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

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:

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

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 but 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.

Definition 3.1.1 (*Evaluation Contexts*) The set of evaluation contexts is the following subset of the set of ISWIM contexts:

$$E ::= [\ ] \mid (V\ E) \mid (E\ M) \mid (o\ V \dots V\ E\ M \dots M)$$

where V is a value and M is an arbitrary expression. ∎

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.

Lemma 3.1.2 (Unique Evaluation Contexts) *Every closed ISWIM expression M is either a value, or there exists a unique evaluation context E such that $M = E[(V_1\ V_2)]$ or $M = E[(o^n\ V_1\ \dots\ V_n)]$ where $V_1, \dots, V_n$ are values.*

Proof. The proof proceeds by induction on the structure of expressions. If M is not a value, it must be an application. Assume that it is a primitive application of the shape:

$$(o^n\ N_1\ \dots\ N_n)$$

where $N_1, \dots, N_n$ are the arguments. If all arguments are values, *i.e.*, $N_i \in \mathit{Vals}$, we are done. Otherwise there is a leftmost argument expression, say N_i for some $i, 1 \leq i \leq n$, that is not a value. By inductive hypothesis, the expression N_i can be partitioned into a unique evaluation context E' and an application of the right shape, say L. Then

$$E = (o^n\ N_1\ \dots\ N_{i-1}\ E'\ N_{i+1}\ \dots\ N_n)$$

is an evaluation context and $M = E[L]$. The context is unique since i is the minimal index such that $N_1, \dots, N_{i-1} \in \mathit{Vals}$ and E' is unique by inductive hypothesis.

The case for arbitrary applications proceeds analoguously. ∎

It follows from the lemma 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 special reduction relation.

Definition 3.1.3 (*Standard Reduction Function*) The *standard reduction function* maps closed non-values to expressions:

$$M \longmapsto_v N \text{ iff for some } E, M \equiv E[M'], N \equiv E[N'], \text{ and } (M', N') \in \mathbf{v}.$$

We pronounce $M \longmapsto_v N$ as "M standard reduces to N." As usuall, $M \longmapsto_v^* N$ means M reduces to N via the transitive-reflexive closure of the standard reduction function. ∎

By Lemma 3.1.2 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: a program can be evaluated by standard reducing it to a value.

Theorem 3.1.4 (Standard Reduction) *For any program M, $M \longrightarrow\!\!\!\!\!\rightarrow_v U$ for some value U if and only if $M \longmapsto_v^* V$ for some value V and $V \longrightarrow\!\!\!\!\!\rightarrow_v U$. In particular, if $U \in BConsts$ then $M \longrightarrow\!\!\!\!\!\rightarrow_v U$ if and only if $M \longmapsto_v^* U$.*

One way to exploit this theorem is to define an alternative evaluator. The alternative *eval* function maps a program to the normal form of the program with respect to the standard reduction relation, if it exists.

Definition 3.1.5 ($eval_v^s$) The partial function $eval_v^s : \Lambda^0 \longrightarrow A$ is defined as follows:

$$eval_v^s(M) = \begin{cases} b & \text{if } M \longmapsto_v^* b \\ \texttt{closure} & \text{if } M \longmapsto_v^* \lambda x.N \end{cases}$$

∎

It is a simple corollary of Theorem 3.1.4 that this new evaluator function is the same as the original one.

Corollary 3.1.6 $eval_v = eval_v^s$

Proof. Assume $eval_v(M) = b$ for some $b \in BConsts$. By Church-Rosser and the fact that constants are normal-forms with respect to $\mathbf{v}$, $M \longrightarrow\!\!\!\!\!\rightarrow_v b$. By Theorem 3.1.4, $M \longmapsto_v^* b$ and thus, $eval_v^s(M) = b$. Conversely, if $eval_v^s(M) = b$ then $M \longmapsto_v^* b$. Hence, $M =_v b$ and $eval_v(M) = b$. A similar proof works for the case where $eval_v(M) = \texttt{closure} = eval_v^s(M)$. In summary, $(M, b) \in eval_v$ if and only if $(M, b) \in eval_v^s$. ∎

Proof of the Standard Reduction Theorem The proof of the Standard Reduction Theorem requires a complex argument. The difficult part of the proof is clearly the left to right direction since the right to left direction is implied by the fact that the standard reduction function and its transitive-reflexive closure are subsets of the reduction relation proper. A naïve proof attempt could proceed by induction on the number of one-step reductions from M to U. Asssume the reduction sequence is as follows:

$$M = M_0 \longrightarrow_v M_1 \longrightarrow_v \ldots \longrightarrow_v M_m = U.$$

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 $M_1 \longrightarrow\!\!\!\!\rightarrow_v U$ there exists a value V such that $M_1 \longmapsto_v^* V$, it is obvious that for many pairs M_0, M_1 it is *not* the case that $M_0 \longmapsto_v^* M_1$. 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. 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.

Definition 3.1.7 (*Standard Reduction Sequences*) The set of standard reduction sequences is a set of expression sequences that satisfies the following four inductive conditions:

1. Every basic constant $b \in \mathit{BConsts}$ is a standard reduction sequence.
2. Every variable $x \in \mathit{Vars}$ is a standard reduction sequence.
3. If $M_1, \ldots, M_m$ is a standard reduction sequence, then so is $\lambda x.M_1, \ldots, \lambda x.M_m$.
4. If $M_1, \ldots, M_m$ and $N_1, \ldots, N_n$ are standard reduction sequences, then so is
$$(M_1\ N_1), (M_2\ N_1), \ldots, (M_m\ N_1), (M_m\ N_2), \ldots, (M_m\ N_n).$$
5. If $M_{i,1}, \ldots, M_{i,m_i}$ are standard reduction sequences for $1 \leq i \leq m$ and $m_i \geq 1$ then
$$\begin{array}{l}(o^m\ M_{1,1}\ M_{2,1}\ \ldots\ M_{m,1}),\\ (o^m\ M_{1,2}\ M_{2,1}\ \ldots\ M_{m,1}),\\ \ldots,\\ (o^m\ M_{1,m_1}\ M_{2,1}\ \ldots\ M_{m,1}),\\ (o^m\ M_{1,m_1}\ M_{2,2}\ \ldots\ M_{m,1}),\\ \ldots,\\ (o^m\ M_{1,m_1}\ M_{2,M_2}\ \ldots\ M_{m,m_m})\end{array}$$
is a standard reduction sequence.

6. If $M_1, \ldots, M_m$ is a standard reduction sequence and $M_0 \longmapsto_v M_1$, then

$$M_0, M_1, \ldots, M_m$$

is a standard reduction sequence.

∎

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.

Theorem 3.1.8 (Plotkin) *$M \twoheadrightarrow_v N$ if and only if there is a standard reduction sequence $L_1, \ldots, L_l$ such that $M = L_1$ and $N = L_l$.*

Proof. 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, a property that obviously follows from the definition. To prove the left-to-right direction, assume $M \twoheadrightarrow_v N$. By the definition of the reduction relation, this means that there exist expressions $M_1, \ldots, M_m$ such that

$$M \longrightarrow_v M_1 \longrightarrow_v \ldots \longrightarrow_v M_m \longrightarrow_v N.$$

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

$$M \underset{1}{\twoheadrightarrow} M_1 \underset{1}{\twoheadrightarrow} \ldots \underset{1}{\twoheadrightarrow} M_m \underset{1}{\twoheadrightarrow} N.$$

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 3.1.10. ∎

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.

Definition 3.1.9 (*Size of $M \underset{1}{\twoheadrightarrow} N$*) The size of the derivation that M parallel reduces to N, $M \underset{1}{\twoheadrightarrow} N$, is defined inductively according to the last derivation step and the size of subderivations:

$$
\begin{array}{ll}
M \underset{1}{\twoheadrightarrow} M & 0 \\
(o\ b_1 \ldots b_n) \underset{1}{\twoheadrightarrow} \delta(o, b_1, \ldots, b_n) & 1 \\
(\lambda x.M)U \underset{1}{\twoheadrightarrow} N[x \leftarrow V] & s_M + \#(x, N) \times s_U + 1 \\
\quad \text{because} & \\
\qquad M \underset{1}{\twoheadrightarrow} N \text{ is of size } s_M & \\
\qquad U \underset{1}{\twoheadrightarrow} V \text{ is of size } s_U & \\
(M\ N) \underset{1}{\twoheadrightarrow} (M'\ N') & s_M + s_N \\
\quad \text{because} & \\
\qquad M \underset{1}{\twoheadrightarrow} M' \text{ is of size } s_M & \\
\qquad N \underset{1}{\twoheadrightarrow} N' \text{ is of size } s_N & \\
(o\ M_1 \ldots M_m) \underset{1}{\twoheadrightarrow} (o\ M_1' \ldots M_m') & \Sigma_{i=1}^{m} s_{M_i} \\
\quad \text{because} & \\
\qquad M_i \underset{1}{\twoheadrightarrow} M_i' \text{ is of size } s_{M_i} & \\
(\lambda x.M) \underset{1}{\twoheadrightarrow} (\lambda x.N) & s_M \\
\quad \text{because} & \\
\qquad M \underset{1}{\twoheadrightarrow} N \text{ is of size } s_M &
\end{array}
$$

$\#(x, M)$ denotes the number of free x in M. ∎

Now we are ready to prove the main lemma.

Lemma 3.1.10 *If $M \underset{1}{\twoheadrightarrow} N$ and $N, N_2, \ldots, N_n$ is a standard reduction sequence, then there is a standard reduction sequence $L_1, \ldots, L_l$ such that $M = L_1$ and $L_l = N_n$.*

Proof. The proof is a lexicographic induction on the length of the given standard reduction sequence (n), the size of the derivation $M \underset{1}{\twoheadrightarrow} N$ and the structure of M. It proceeds by case analysis on the last step in the derivation of $M \underset{1}{\twoheadrightarrow} N$:

$M = N$**:** This case is trivial.

$M = (o\ b_1\ \ldots b_m), N = \delta^m(f, b_1, \ldots, b_m)$**:** 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.

$M = ((\lambda x.K)\ U), N = L[x \leftarrow V]$ because $K \underset{1}{\twoheadrightarrow} L$**,** $U \underset{1}{\twoheadrightarrow} V$**:** 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 Lemma 2.5.6, the latter expression also parallel reduces to $L[x \leftarrow V]$. Indeed, we can prove a stronger version of Lemma 2.5.6, see Lemma 3.1.13 below, that shows that the size of this derivation is strictly smaller than the size of the derivation of $((\lambda x.K)\ U) \underset{1}{\twoheadrightarrow} L[x \leftarrow V]$. Thus, the induction hypothesis applies and there must be a standard reduction sequence $L_2, \ldots, L_l$ such that $K[x \leftarrow U] = L_2$ and $L_l = N_n$. Since $M \longmapsto_v K[x \leftarrow U]$, the expression sequence $M, L_2, \ldots, L_l$ is the required standard reduction sequence.

$M = (M' \; M''), N = (N' \; N'')$ because $M' \longrightarrow\!\!\!\!\!\rightarrow_1 N'$, $M'' \longrightarrow\!\!\!\!\!\rightarrow_1 N''$: Since the standard reduction sequence $N, N_2, \ldots, N_n$ could be formed in two different ways, we must consider two subcases.

$N \longmapsto_v N_2$: This case relies on the following Lemma 3.1.11, which shows that if $M \longrightarrow\!\!\!\!\!\rightarrow_1 N$ and $N \longmapsto_v N_2$ then there exists an expression K such that $M \longmapsto_v K$ and $K \longrightarrow\!\!\!\!\!\rightarrow_1 N_2$. Now, $K \longrightarrow\!\!\!\!\!\rightarrow_1 N_2$ and $N_2, N_3, \ldots, N_n$ is a standard reduction sequence that is shorter than the original one. Hence, by the induction hypothesis there exists a standard reduction sequence $L_2, \ldots, L_l$ such that $K = L_2$ and $L_l = N_n$. Moreover, $M \longmapsto_v K$, and hence $M, L_2, \ldots, L_l$ is the desired standard reduction sequence.

otherwise: $N', \ldots, N'_k$ and $N'', \ldots, N''_j$ are standard reduction sequences such that $N, N_2, \ldots, N_n$ is identical to $(N' \; N''), \ldots, (N'_k \; N''), (N'_k \; N''_2), \ldots, (N'_k \; N''_j)$. By the assumptions and the induction hypothesis, which applies because M' and M'' are proper subterms of M, there exist standard reduction sequences $L'_1, \ldots, L'_{l_1}$ and $L''_1, \ldots, L''_{l_2}$ 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 $L_1, \ldots, L_l$ such that $L_1 = (L'_1 \; L''_1) = (M' \; M'')$ and $L_l = (L'_{l_1} \; L''_{l_2}) = (N'_k \; N''_j) = N_n$, which is precisely what the lemma demands.

$M = (o \; M_1 \ldots M_m), N = (o \; N'_1 \ldots N'_m)$ because $M_i \longrightarrow\!\!\!\!\!\rightarrow_1 N_i$: 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.

$M = (\lambda x.K), N = (\lambda x.L)$ because $K \longrightarrow\!\!\!\!\!\rightarrow_1 L$: By the definition of standard reduction sequences, all of the expressions N_i are λ-abstractions, say, $N_i = \lambda x.N'_i$. Hence, $K \longrightarrow\!\!\!\!\!\rightarrow_1 L$ and $L, N'_2, \ldots, N'_n$ 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 $L'_1, \ldots, L'_l$ such that $K = L'_1$ and $L'_l = N'_n$. 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 3.1.11 *If $M \longrightarrow\!\!\!\!\!\rightarrow_1 N$ and $N \longmapsto_v L$ then there exists an expression N^* such that $M \longmapsto_v^* N^*$ and $N^* \longrightarrow\!\!\!\!\!\rightarrow_1 L$.*

Proof. The proof is a lexicographic induction on the size of the derivation of $M \longrightarrow\!\!\!\!\!\rightarrow_1 N$ and the structure of M. It proceeds by case analysis on the last step in the parallel reduction derivation.

$M = N$: In this case, the conclusion vacuously holds.

$M = (o\ b_1\ \ldots\ b_m), N = \delta^m(o, b_1, \ldots, b_m)$: Since the result of a δ-step is a value, it is impossible that N standard reduces to some other term.

$M = ((\lambda x.K)\ U), N = L[x \leftarrow V]$ because $K \twoheadrightarrow_1 L$, $U \twoheadrightarrow_1 V$: M is also a β_v-redex, and therefore $M \longmapsto_v K[x \leftarrow U]$. Next, by Lemma 2.5.6

$$K[x \leftarrow U] \twoheadrightarrow_1 L[x \leftarrow V],$$

and the derivation of this parallel reduction is smaller than the derivation of $M \twoheadrightarrow_1 N$ by Lemma 3.1.13. By induction hypothesis, there must be an expression N^* such that $K[x \leftarrow U] \longmapsto_v N^* \twoheadrightarrow_1 L[x \leftarrow V]$. Hence we can prove the desired conclusion as follows:

$$M \longmapsto_v K[x \leftarrow U] \longmapsto_v N^* \twoheadrightarrow_1 L[x \leftarrow V].$$

$M = (M'\ M''), N = (N'\ N'')$ because $M' \twoheadrightarrow_1 N'$, $M'' \twoheadrightarrow_1 N''$: Here we distinguish three subcases according to the standard reduction step from N to L:

$N = ((\lambda x.K')\ N'')$ where $N'' \in$ *Values*: That is, N is a β_v-redex and L is its contractum. By the assumed parallel reductions and Lemma 3.1.12 below, there exist $(\lambda x.K)$ and N^{**} such that $M' \longmapsto_v^* (\lambda x.K) \twoheadrightarrow_1 (\lambda x.K')$ and $M'' \longmapsto_v^* N^{**} \twoheadrightarrow_1 N''$. Hence,

$$\begin{array}{rcl} (M'\ M'') & \longmapsto_v^* & ((\lambda x.K)\ M'') \\ & \longmapsto_v^* & ((\lambda x.K)\ N^{**}) \\ & \longmapsto_v & K[x \leftarrow N^{**}] \\ & \twoheadrightarrow_1 & K'[x \leftarrow N''], \end{array}$$

which is precisely what the lemma claims.

$N = (E[K]\ N'')$: Then, $L = (E[K']\ N'')$ where $(K, K'') \in \mathbf{v}$ and

$$M' \twoheadrightarrow_1 E[K] \longmapsto_E [K'].$$

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

$$M' \longmapsto_v^* N_1^* \twoheadrightarrow_1 E[K'],$$

which implies that

$$(M'\ M'') \longmapsto_v^* (N_1^*\ M'') \twoheadrightarrow_1 (E[K']\ N'').$$

In other words, $N^* = (N_1^*\ M'')$.

$N = (N'\ E[K])$ and $N' \in$ *Values*: In this case, $L = (N'\ E[K'])$ where $(K, K'') \in$ **v** and

$$M'' \longrightarrow\!\!\!\!\!\rightarrow_1 E[K] \longmapsto_E [K'].$$

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

$$M'' \longmapsto_v^* N_2^* \longrightarrow\!\!\!\!\!\rightarrow_1 E[K'].$$

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

$$M' \longmapsto_v^* N_1^* \longrightarrow\!\!\!\!\!\rightarrow_1 N'.$$

Putting it all together we get,

$$\begin{array}{rcl} (M'\ M'') & \longmapsto_v^* & (N_1^*\ M'') \\ & \longmapsto_v^* & (N_1^*\ N_2^*) \\ & \longmapsto_v & (N_1^*\ N_2^*) \\ & \longrightarrow\!\!\!\!\!\rightarrow_1 & (N'\ E[K']). \end{array}$$

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

$M = (o\ M_1 \ldots M_m), N = (o\ N_1' \ldots N_m')$because$M_i \longrightarrow\!\!\!\!\!\rightarrow_1 N_i$: 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 3.1.12 shows that a term that parallel reduces to a basic constant also standard reduces to it, the rest of the argument is straightforward.

$M = (\lambda x.K), N = (\lambda x.L)$ because $K \longrightarrow\!\!\!\!\!\rightarrow_1 L$: This case is again impossible because the standard reduction function is undefined on values.

This completes the case analysis and the proof. ∎

Lemma 3.1.12 *Let M be an application.*

1. *If $M \longrightarrow\!\!\!\!\!\rightarrow_1 (\lambda x.N)$ then there exists an expression $\lambda x.L$ such that $M \longmapsto_v^* (\lambda x.L) \longrightarrow\!\!\!\!\!\rightarrow_1 (\lambda x.N)$.*
2. *If $M \longrightarrow\!\!\!\!\!\rightarrow_1 N$ where $N = x$ or $N = c$ then $M \longmapsto_v^* N$.*

Proof. 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 this 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 $M' \longrightarrow\!\!\!\!\!\rightarrow_1 N'$ and $U \longrightarrow\!\!\!\!\!\rightarrow_1 V$ by Lemma 2.5.6. Hence,

$$((\lambda y.M')\ U) \longmapsto_v M'[y \leftarrow U] \longrightarrow\!\!\!\!\!\rightarrow_1 N'[y \leftarrow V] = \lambda x.N.$$

By Lemma 3.1.13, we also know that the latter derivation is shorter than the original parallel reduction step and therefore, by inductive hypothesis,

$$((\lambda y.M')\ U) \longmapsto_v M'[y \leftarrow U] \longmapsto_v^* N.$$

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 3.1.13 *If $M \longrightarrow\!\!\!\!\!\rightarrow_1 N$ has size s_M and $U \longrightarrow\!\!\!\!\!\rightarrow_1 V$ has size s_N then (i) $M[x \leftarrow U] \longrightarrow\!\!\!\!\!\rightarrow_1 N[x \leftarrow V]$ and (ii) size s of the derivation of the latter is less than or equal to $s_M + \#(x, N) \times s_U$.*

Remarks: (1) This lemma is a strengthening of Lemma 2.5.6. For completeness, we repeat the proof of the original lemma and add components about the size of the parallel reduction derivation.
(2) The lemma implies that the size of the derivation of $((\lambda x.M)\ U) \longrightarrow\!\!\!\!\!\rightarrow_1 N[x \leftarrow V]$ is strictly larger than the size of the derivation of $M[x \leftarrow U] \longrightarrow\!\!\!\!\!\rightarrow_1 N[x \leftarrow V]$.
Proof. The proof is an induction on M and proceeds by case analysis on the last step in the derivation of $M \longrightarrow\!\!\!\!\!\rightarrow_1 N$:

$M = N$: In this special case, the claim says that if $U \longrightarrow\!\!\!\!\!\rightarrow_1 V$ then $M[x \leftarrow U] \longrightarrow\!\!\!\!\!\rightarrow_1 M[x \leftarrow V]$. *As to the claim about size: the assumption implies that $s_M = 0$, and therefore it must be true that $s \le \#(x, M) \times s_U$.* The derivation of this specialized claim is an induction on the structure of M:

$M = c$: $M[x \leftarrow U] = c \longrightarrow\!\!\!\!\!\rightarrow_1 c = M[x \leftarrow V]$;
size: $s = 0$;

$M = x$: $M[x \leftarrow U] = U \longrightarrow\!\!\!\!\!\rightarrow_1 V = M[x \leftarrow V]$;
size: $s = s_U = \#(x, M) \times s_U$;

$M = y \neq x$: $M[x \leftarrow U] = y \longrightarrow\!\!\!\!\!\rightarrow_1 y = M[x \leftarrow V]$;
size: $s = 0$;

$M = (\lambda y.K)$: $M[x \leftarrow U] = (\lambda y.K[x \leftarrow U]) \longrightarrow\!\!\!\!\!\rightarrow_1 (\lambda y.K[x \leftarrow V]) = M[x \leftarrow V]$ by induction hypothesis for K;
size: $s = \#(x, K) \times s_U = \#(x, M) \times s_U$;

$M = (K\ L)$: $M[x \leftarrow U] = (K[x \leftarrow U]\ L[x \leftarrow U]) \xrightarrow[1]{} \!\!\!\!\! \rightarrow (K[x \leftarrow V]\ L[x \leftarrow V]) = M[x \leftarrow V]$ by induction hypothesis for K, L;
size: $s = \#(x, K) \times s_U + \#(x, L) \times s_U = \#(x, M) \times s_U$.

$M = (o\ b_1 \ldots b_m), N = \delta(o, b_1, \ldots, b_m)$: 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: $s = 0$.

$M = ((\lambda y.K)\ W), N = L[y \leftarrow W']$ because $K \xrightarrow[1]{}\!\!\!\!\!\rightarrow L$, $W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'$: A simple calculation shows that the claim holds:

$$\begin{array}{rcl} M[x \leftarrow U] & = & ((\lambda y.K[x \leftarrow U])\ W[x \leftarrow U]) \\ & \xrightarrow[1]{}\!\!\!\!\!\rightarrow & L[x \leftarrow V][y \leftarrow W'[x \leftarrow V]] \\ & = & L[y \leftarrow W'][x \leftarrow V] \qquad (\dagger) \\ & = & N[x \leftarrow V] \end{array}$$

substitution lemma

Equation ($\dagger$) 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:

$$\begin{array}{rl} s_K: & K \xrightarrow[1]{}\!\!\!\!\!\rightarrow L \\ s'_K: & K[x \leftarrow U] \xrightarrow[1]{}\!\!\!\!\!\rightarrow L[x \leftarrow V] \\ s_W: & W \xrightarrow[1]{}\!\!\!\!\!\rightarrow W' \\ s'_W: & W[x \leftarrow U] \xrightarrow[1]{}\!\!\!\!\!\rightarrow W'[x \leftarrow V] \end{array}$$

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

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

$M = (M_1\ M_2), N = (N_1\ N_2)$ because $M_i \xrightarrow[1]{}\!\!\!\!\!\rightarrow N_i$: By the induction hypothesis, $M_i[x \leftarrow U] \xrightarrow[1]{}\!\!\!\!\!\rightarrow N_i[x \leftarrow V]$ for both $i = 1$ and $i = 2$. Thus,

$$M[x \leftarrow U] \quad = \quad (M_1[x \leftarrow U]\ M_2[x \leftarrow U])$$

$$\begin{array}{rl} \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} & (N_1[x \leftarrow V]\ N_2[x \leftarrow V]) \\ = & (N_1\ N_2)[x \leftarrow V] \\ = & N[x \leftarrow V] \end{array}$$

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

$$\begin{array}{rl} s_i: & M_i \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_i \\ s'_i: & M_i[x \leftarrow U] \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_i[x \leftarrow V] \end{array}$$

Thus, by induction hypothesis,

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

The rest follows from a simple calculation:

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

$M = (o\ M_1 \ldots M_m), N = (o\ N'_1 \ldots N'_m)$ because $M_i \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} N_i$: The proof of this case proceeds as the preceding one.

$M = (\lambda x.K), N = (\lambda x.L)$ because $K \underset{1}{\longrightarrow\!\!\!\!\!\rightarrow} L$: The proof of this case is also an appropriate adaptation of the proof for the application 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 3.1.4, the correctness proof for the textual machine.

First, it provides a powerful tool for proving and disproving equations in the λ_v-calculus. Second, we can now show that evaluation is indeed equivalent to standard reduction steps.

Theorem 3.1.4 *For any program M, $M \longrightarrow\!\!\!\!\!\rightarrow_v U$ for some value U if and only if $M \longmapsto_v^* V$ for some value V and $V \longrightarrow\!\!\!\!\!\rightarrow_v U$. In particular, if $U \in BConsts$ then $M \longrightarrow\!\!\!\!\!\rightarrow_v U$ if and only if $M \longmapsto_v^* U$.*

Proof. To prove the left to right direction, assume that

$$M \longrightarrow\!\!\!\!\!\rightarrow_v U$$

and that M is not a value. It follows from the Standard Reduction Theorem that there exists a standard reduction sequence $L_1, \ldots, L_l$ such that $M = L_1$ and $L_l = U$. 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

$$M = L_1 \longmapsto_v^* L_i.$$

Set $V = L_i$; trivially, $L_i \twoheadrightarrow_v U$. Moreover, constants can only be the end of a standard reduction sequence of values if the standard reduction sequence is a singleton. Hencem the second part of the theorem concerning basic constants obviously holds. ∎

In summary, we have now 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.

Exercise 3.1.1 Implement the standard reduction function $\longmapsto_v$ as a procedure *standard* that maps ISWIM programs to ISWIM programs. Implement $eval_v^s$ based on the *standard*. Pick some simple benchmarks and determine their running time on the machine.
Hint: Make sure that the computations in the test programs do not dominate the execution of the transition function. ∎

3.2 Machines

need to say that the machines are functions

Every evaluation cycle in our 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 $(o\, V_1 \ldots V_m)$, whose subterms are values.

3. If the application is a β_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 problem with 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

$$(1^+\ ((\lambda x.((\lambda y.((\lambda z.x)\ \ulcorner 3 \urcorner))\ \ulcorner 2 \urcorner))\ \ulcorner 1 \urcorner)),$$

which is a fairly typical example:

$$
\begin{array}{ll}
 & (1^+ \ \underline{((\lambda x.((\lambda y.((\lambda z.x) \ \ulcorner 3 \urcorner)) \ \ulcorner 2 \urcorner)) \ \ulcorner 1 \urcorner)}) \\
\longmapsto_v & (1^+ \ \underline{((\lambda y.((\lambda z.\ulcorner 1 \urcorner) \ \ulcorner 3 \urcorner)) \ \ulcorner 2 \urcorner)}) \\
\longmapsto_v & (1^+ \ \underline{((\lambda z.\ulcorner 1 \urcorner) \ \ulcorner 3 \urcorner)}) \\
\longmapsto_v & \underline{(1^+ \ \ulcorner 1 \urcorner)} \\
\longmapsto_v & \ulcorner 2 \urcorner .
\end{array}
$$

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 $(1^+ \ [\ \ ])$. But the machine has to recompute the evaluation context at every stage.

3.2.1 The CC Machine

The appropriate method for dealing with this repeated computation is to separate states into the "current subterm of interest" and the "current" evaluation contexts. 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 and the evaluation contexts simply as contexts. The machine is appropriately called CC machine.

In addition to the tasks of the textual machine, the CC machine is responsible for searching the next redex. Initially only the entire program can be the control string; the initial evaluation context must be the empty context. The search instructions implement rules 1 and $2'$ from the beginning of this chapter. That is, if the control string is not an application that consists of values, the leftmost non-value becomes the control string, the rest becomes a part of the evaluation context. For example, if the control string is $((M\ N)\ L)$ and the evaluation context is $E[\ \ ]$, then the machine must search for the redex in $(M\ N)$ and must remember the shape of the rest of the program. Consequently the next state of the machine must have $(M\ N)$ as the control string component and must refine the current context $E[\ \ ]$ to a context of the shape $E[([\ \ ]\ L)]$. Put succinctly, the CC machine must have a state transition of the form

$$\langle (MN), E[\ \ ] \rangle \longmapsto \langle M, E[([\ \ ]\ N)] \rangle \text{ if } M \notin \mathit{Vals}.$$

Search rules for other shapes of applications that do not solely consist of values are needed, too.

Once the control string becomes a redex, the CC machine must naturally simulate the actions of the textual machine to be a faithful implementation. Thus it must have the following two instructions:

$$
\begin{array}{lcll}
\langle (o\ b_1 \ldots b_m), E[\ \ ] \rangle & \longmapsto & \langle V, E[\ \ ] \rangle \text{ where } \delta(o, b_1, \ldots, b_m) = V & (cc.\delta) \\
\langle ((\lambda x.M)V), E[\ \ ] \rangle & \longmapsto & \langle M[x \leftarrow V], E[\ \ ] \rangle & (cc.\beta)
\end{array}
$$

The result of one of these transitions might be a state that pairs a value with an evaluation contexts. In the textual machine this corresponds to a step of the form

$$E[R] \longmapsto_v E[V].$$

Now the textual machine would actually divide the second program into a redex R' and an 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 $R' = ((\lambda x.M)\ V)$, that is, the new redex is formed of the innermost application in $E[\ ]$, and $E'[\ ]$ is the rest of $E[\ ]$.

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

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

Now 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.

Definition 3.2.1 (*CC machine*)

State space:

$$\Lambda^0 \times EConts$$

State transitions:

Redex		*Contractum*	*cc*
$\langle (MN), E[\ ]\rangle$ if $M \notin \mathit{Vals}$	$\longmapsto$	$\langle M, E[([\ ]\ N)]\rangle$	1
$\langle (VM), E[\ ]\rangle$ if $M \notin \mathit{Vals}$	$\longmapsto$	$\langle M, E[(V\ [\ ])]\rangle$	2
$\langle (o\, V_1 \ldots V_i\ M\ N \ldots), E[\ ]\rangle$ if $M \notin \mathit{Vals}$, for all $i \geq 0$	$\longmapsto$	$\langle M, E[(o\, V_1 \ldots V_i\ [\ ]\ N \ldots)]\rangle$	3
$\langle ((\lambda x.M)\ V), E[\ ]\rangle$	$\longmapsto$	$\langle M[x \leftarrow V], E[\ ]\rangle$	β_v
$\langle (o\, b_1 \ldots b_m), E[\ ]\rangle$	$\longmapsto$	$\langle V, E[\ ]\rangle$ where $\delta(o, b_1, \ldots, b_m) = V$	δ_v
$\langle V, E[(U\ [\ ])]\rangle$	$\longmapsto$	$\langle (U\ V), E[\ ]\rangle$	4
$\langle V, E[([\ ]\ N)]\rangle$	$\longmapsto$	$\langle (V\ N), E[\ ]\rangle$	5
$\langle V, E[(o\, V_1 \ldots V_i\ [\ ]\ N \ldots)]\rangle$	$\longmapsto$	$\langle (o\, V_1 \ldots V_i\ V\ N \ldots), E[\ ]\rangle$	6

The evaluation function:

$$eval_v^{cc}(M) = \begin{cases} b & \text{if } \langle M, [\ ]\rangle \longmapsto_{cc}^{*} \langle b, [\ ]\rangle \\ \texttt{closure} & \text{if } \langle M, [\ ]\rangle \longmapsto_{cc}^{*} \langle \lambda x.N, [\ ]\rangle \end{cases}$$

∎

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. The following theorem formulates this idea and proves it.

Theorem 3.2.2 $eval_v^s = eval_v^{cc}$

Proof. We need to show that if $(M, a) \in eval_v^{cc}$ then $(M, a) \in eval_v^s$ and vice versa. By the definition of the two evaluation functions, this require a proof that

$$M \longmapsto_v^* V \text{ if and only if } \langle M, [\ ]\rangle \longmapsto_{cc}^* \langle V, [\ ]\rangle.$$

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 in the context component, we will need a slightly stronger induction hypothesis:

For all evaluation contexts $E[\ ]$, terms M, and values V,

$$E[M] \longmapsto_v^* V \text{ if and only if } \langle M, E[\ ]\rangle \longmapsto_{cc}^* \langle V, [\ ]\rangle.$$

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 $E[M] \longmapsto_v^* V$. If the reduction sequence is empty, the conclusion is immediate, so assume there is at least one step:

$$E[M] \longmapsto_v E'[N] \longmapsto_v^* V$$

where $(L, N) \in \mathbf{v}$ for some L. By inductive hypothesis, $\langle N, E'[\;\;]\rangle \longmapsto_{cc}^* \langle V, [\;\;]\rangle$. To conclude the proof, we must show that

$$\langle M, E[\;\;]\rangle \longmapsto_{cc} \langle N, E'[\;\;]\rangle.$$

Depending on the shape of M and $E[\;\;]$, there are four possibilities:

$M = E''[L]$: Then, $E'[\;\;] = E[E''[\;\;]]$. By Lemma 3.2.3 below, $\langle M, E[\;\;]\rangle \longmapsto_{cc}^* \langle L, E[E''[\;\;]]\rangle$ and, hence, by a $(cc.\delta)$ or $(cc.\beta_v)$ transition, $\langle L, E[E''[\;\;]]\rangle \longmapsto_{cc} \langle N, E[E''[\;\;]]\rangle$.

$E[\;\;] = E'[((\lambda x.N')\ [\;\;])], M \in \mathit{Vals}$: A first step puts the redex into the control position, which enables a $(cc.\beta_v)$ reduction:

$$\langle M, E'[((\lambda x.N')\ [\;\;])]\rangle \longmapsto_{cc} \langle((\lambda x.N')\ M), E'[\;\;]\rangle \longmapsto_{cc} \langle N'[x \leftarrow M], E'[\;\;]\rangle.$$

Since the assumptions imply that $N = N'[x \leftarrow M]$, this proves the conclusion.

$E[\;\;] = E''[([\;\;]\ K)], M \in \mathit{Vals}$: Depending on the nature of K, there are two different scenarios:

1. If $K \in \mathit{Values}$ then $L = (K\ M)$, which implies the conclusion.
2. If $K \notin \mathit{Values}$ then there exists an evaluation context $E'''[\;\;]$ such that $K = E'''[L]$. Hence, $E'[] = E''[(M\ E'''[L])]$ and

$$\begin{array}{rcl}
\langle M, E''[([\;\;]\ K)]\rangle & \longmapsto_{cc} & \langle (M\ K), E''[\;\;]\rangle \\
& \longmapsto_{cc} & \langle K, E''[(M\ [\;\;])]\rangle \\
& \longmapsto_{cc}^* & \langle L, E''[(M\ E''[\;\;])]\rangle \text{ by Lemma 3.2.3} \\
& \longmapsto_{cc}^* & \langle N, E''[(M\ E''[\;\;])]\rangle \\
& = & \langle N, E'[\;\;]\rangle.
\end{array}$$

$E[\;\;] = E'[(o\ b_1 \ldots [\;\;] \ldots\ b_n)], M \in \mathit{BConsts}$: Appropriate adaptations of the proofs of the second and third case prove the conclusion.

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

$$\langle M, E[\ ]\rangle \longmapsto_{cc} \langle N, E'[\ ]\rangle \longmapsto^*_{cc} \langle V, [\ ]\rangle.$$

By inductive hypothesis, $E'[N] \longmapsto^*_v V$. 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 $(cc.1), \ldots, (cc.6)$, then $E[M] = E'[N]$, and the conclusion follows. If it is a $(cc.\beta_v)$ or a $(cc.\delta)$ transition, then $E[M] \longmapsto_v E'[N]$. Put together with the inductive hypothesis,

$$E[M] \longmapsto_v E'[N] \longmapsto^*_v V. \blacksquare$$

Lemma 3.2.3 *For all evaluation contexts $E[\ ]$, if $M = E'[L]$ and L is a* **v***-redex, then*

$$\langle M, E[\ ]\rangle \longmapsto^*_{cc} \langle L, E[E'[\ ]]\rangle.$$

Proof. 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:

$$E'[\ ] = (E''[\ ]\ N), E'[\ ] = (V\ \ E''[\ ]),\ \text{or}\ E'[\ ] = (o\ V_1\ \ldots V_n\ E''[\ ]\ N\ldots).$$

In each case, the machine can move the pieces surrounding $E''[\ ]$ to the context component, *e.g.*,

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

Since $E''[\ ]$ is a component of $E'[\ ]$, the conclusion follows from the inductive hypothesis. ∎

Exercise 3.2.1 Implement the CC-transition function $\longmapsto_{cc}$ that maps CC-states to CC-states. Implement evaluation contexts as an extension of ISWIM expressions. Define eval^{cc}_v based on the transition function. Benchmark the same programs that you used for the test of eval^s_v (see Exercise 1). Explain the timings. ∎

3.2.2 Simplified CC 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:

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

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:

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

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:

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

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

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

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 $cc.2$, which only exists to move arguments from evaluation contexts into the control string position.

start calling things registers

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.

Definition 3.2.4 (*Simplified CC machine*)
State space:

$$\Lambda^0 \times EConts$$

State transitions:

Redex		*Contractum*	*scc*
$\langle (MN), E[\ ]\rangle$	$\longmapsto$	$\langle M, E[([\ ]\ N)]\rangle$	1
$\langle (o\ \ M\ N\ldots), E[\ ]\rangle$	$\longmapsto$	$\langle M, E[(o\ \ [\ \ ]\ N\ldots)]\rangle$	2
$\langle V, E[((\lambda x.M)\ [\ \ ])]\rangle$	$\longmapsto$	$\langle M[x \leftarrow V], E[\ ]\rangle$	3
$\langle V, E[([\ \ ]\ N)]\rangle$	$\longmapsto$	$\langle N, E[(V\ [\ \ ])]\rangle$	4
$\langle b, E[(o\, b_1 \ldots b_i\ [\ \ ])]\rangle$	$\longmapsto$	$\langle V, E[\ ]\rangle$ where $\delta(o, b_1, \ldots, b_i, b) = V$	5
$\langle V, E[(o\, V_1 \ldots V_i\ [\ \ ]\ N L\ldots)]\rangle$	$\longmapsto$	$\langle N, E[(o\, V_1 \ldots V_i\ V\ [\ \ ]\ L\ldots)]\rangle$	6

The evaluation function:

$$eval_v^{scc}(M) = \begin{cases} b & \text{if } \langle M, [\ ]\rangle \longmapsto^*_{scc} \langle b, [\ ]\rangle \\ \texttt{closure} & \text{if } \langle M, [\ ]\rangle \longmapsto^*_{scc} \langle \lambda x.N, [\ ]\rangle \end{cases}$$

∎

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

Theorem 3.2.5 $eval_v^{cc} = eval_v^{scc}$

Proof. The proof of this theorem is analoguous to the proof of Theorem 3.2.2. That is, unfolding the definition of the two evaluation functions leads to the proof obligation that

$$\langle M, [\ ]\rangle \longmapsto^*_{cc} \langle V, [\ ]\rangle \text{ if and only if } \langle M, [\ ]\rangle \longmapsto^*_{scc} \langle V, [\ ]\rangle.$$

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

For all evaluation contexts $E[\ ]$, terms X, and values V,

$$\langle M, E[\ ]\rangle \longmapsto^*_{cc} \langle V, [\ ]\rangle \text{ if and only if } \langle M, E[\ ]\rangle \longmapsto^*_{scc} \langle V, [\ ]\rangle.$$

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

$$\langle M, E[\ ]\rangle \longmapsto^+_{cc} \langle N, E''[]\rangle \text{ and } \langle M, E[\ ]\rangle \longmapsto^+_{scc} \langle N, E''[]\rangle.$$

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:

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

$$\langle E'[L], E[\ ]\rangle \longmapsto^*_{cc} \langle L, E[E'[\ ]]\rangle \longmapsto_{cc} \langle N, E[E'[\ ]]\rangle.$$

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 $\langle b, E[E'[(o\ b_1 \ldots b_i\ [\ ])]]\rangle$ where $L = (o\ b_1 \ldots b_i\ b)$.

$M \in Vals, E[\ ] = E'[((\lambda x.L')\ [\ ])]$: Here the CC-machine creates a redex in the control component and reduces it:

$$\langle M, E'[((\lambda x.L')\ [\ ])]\rangle \longmapsto_{cc} \langle((\lambda x.L')\ M), E'[]\rangle \longmapsto_{cc} \langle L'[x \leftarrow M], E'[]\rangle.$$

The SCC-machine avoids the first step and immediately goes into the same state.

$M \in Vals, E[\ ] = E'[([\ ]\ N)]$: If $N \in Vals$, the proof of this case proceeds along the lines of the second case. Otherwise,

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

Again, the SCC-machine skips the intermediate state.

$M \in Vals, E[\ ] = E'[(o\ V_1 \ldots V_m\ [\ ]\ N_1 \ldots N_n)]$: 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 $\delta(o, V_1, \ldots, V_m, V) = U$ for some value U. It follows that

$$\langle M, E'[(o\ V_1 \ldots V_n\ [\ ])]\rangle \longmapsto_{cc} \langle (o\ V_1 \ldots V_n M), E'[\ ]\rangle \longmapsto_{cc} \langle U, E'[\ ]\rangle.$$

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 $N_1, \ldots N_n \in Vals$, then

$$\begin{array}{rcl} \langle M, E'[(o\ V_1 \ldots V_m\ [\ ]\ N_1 \ldots N_n)]\rangle & \longmapsto_{cc} & \langle (o\ V_1 \ldots V_m\ M\ N_1 \ldots N_n), E'[\ ]\rangle \\ & \longmapsto_{cc} & \langle U, E'[\ ]\rangle. \end{array}$$

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

$$\begin{array}{ll} & \langle M, E'[(o\ V_1 \ldots V_m\ [\ ]\ N_1 \ldots N_n)]\rangle \\ \longmapsto_{scc} & \langle N_1, E'[(o\ V_1 \ldots V_m\ M\ [\ ]\ N_2 \ldots N_n)]\rangle \\ \longmapsto^*_{scc} & \ldots \\ \longmapsto^*_{scc} & \langle N_n, E'[(o\ V_1 \ldots N_{n-1}\ [\ ])]\rangle \\ \longmapsto_{scc} & \langle U, E'[\ ]\rangle. \end{array}$$

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

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

after an appropriate number of steps.

The rest of the proof is a straightforward induction on the number of transition steps. Assume $\langle M, E[\ \ ]\rangle \longmapsto_{cc}^{*} \langle V, [\ \ ]\rangle$. 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 3.2.2 Implement the SCC-transition function $\longmapsto_{cc}$ as a procedure *standard* that maps SCC-states to SCC-states. Implement $eval_v^{scc}$. Check whether the simplified machine is faster or slower on your benchmarks. ∎

3.2.3 CK Machine

Any description 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. When the control string is an application, a new innermost application is created. When the control string turns into a value, the next step is determined by considering this 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:

$$\begin{array}{rcl} K & ::= & K_s^* \\ K_s & ::= & (V\ [\ \ ]) \mid ([\ \ ]\ N) \mid (o\ V \ldots V\ [\ \ ]\ N \ldots) \end{array}$$

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

$$\begin{array}{rcl} K & ::= & \mathsf{mt} \\ & & \mid \langle \mathsf{fun}, V, K\rangle \\ & & \mid \langle \mathsf{arg}, N, K\rangle \\ & & \mid \langle \mathsf{narg}, \langle V, \ldots, V, o\rangle, \langle N, \ldots\rangle, K\rangle \end{array}$$

This data structure is called a continuation code.

Definition 3.2.6 (*CK machine*)
State space:

$$\Lambda^0 \times KCodes$$

where the set *KCodes* is defined by the grammar:

$$\begin{array}{rcl} K & ::= & \mathsf{mt} \\ & & \mid \langle \mathsf{fun}, V, K\rangle \\ & & \mid \langle \mathsf{arg}, N, K\rangle \\ & & \mid \langle \mathsf{narg}, \langle V, \ldots, V, o\rangle, \langle N, \ldots\rangle, K\rangle \end{array}$$

State transitions:

Redex		*Contractum*	*ck*
$\langle (MN), K\rangle$	$\longmapsto$	$\langle M, \langle \mathsf{arg}, N, K\rangle\rangle$	1
$\langle (o\ M\ N \ldots), K\rangle$	$\longmapsto$	$\langle M, \langle \mathsf{narg}, \langle o\rangle, \langle N, \ldots\rangle, K\rangle\rangle$	2
$\langle V, \langle \mathsf{fun}, (\lambda x.M), K\rangle\rangle$	$\longmapsto$	$\langle M[x \leftarrow V], K\rangle$	3
$\langle V, \langle \mathsf{arg}, N, K\rangle\rangle$	$\longmapsto$	$\langle N, \langle \mathsf{fun}, V, K\rangle\rangle$	4
$\langle b, \langle \mathsf{narg}, \langle b_i \ldots b_1\ o\rangle, \langle\rangle, K\rangle\rangle$	$\longmapsto$	$\langle V, K\rangle$ where $\delta(o, b_1, \ldots, b_i, b) = V$	5
$\langle V, \langle \mathsf{narg}, \langle V' \ldots o\rangle, \langle N, L, \ldots\rangle, K\rangle\rangle$	$\longmapsto$	$\langle N, \langle \mathsf{narg}, \langle V,\ V', \ldots\rangle, \langle L, \ldots\rangle, K\rangle\rangle$	6

The evaluation function:

$$eval_v^{ck}(M) = \begin{cases} b & \text{if } \langle M, \mathsf{mt}\rangle \longmapsto_{ck}^* \langle b, \mathsf{mt}\rangle \\ \texttt{closure} & \text{if } \langle M, \mathsf{mt}\rangle \longmapsto_{ck}^* \langle \lambda x.N, \mathsf{mt}\rangle \end{cases}$$

∎

Given an SCC-state it is trivial to find a corresponding CK-state and vice versa based on a bijection between the set of evaluation contexts and the set of continuation codes:

$$\mathcal{KC} : KCodes \longrightarrow EvalConts$$

$$\mathcal{KC}(K) = \begin{cases} [\ ] & \text{if } K = \mathsf{mt} \\ E'[([\ ]\ N)] & \text{if } K = \langle \mathsf{arg}, N, K'\rangle \\ E'[(V\ [\ ])] & \text{if } K = \langle \mathsf{fun}, V, K'\rangle \\ E'[(o\ V_1 \ldots V_i\ [\ ]\ N \ldots)] & \text{if } K = \langle \mathsf{narg}, \langle V_i, \ldots, V_1, o\rangle, \langle N, \ldots\rangle, K'\rangle \\ \quad \text{where } E'[\ ] = \mathcal{KC}(K') & \end{cases}$$

$$\mathcal{CK} : EvalConts \longrightarrow KCodes$$

$$\mathcal{CK}(E[\ ]) = \begin{cases} \mathsf{mt} & \text{if } E[\ ] = [\ ] \\ \langle \mathsf{fun}, N, K'\rangle & \text{if } E[\ ] = E'[([\ ]\ N)] \\ \langle \mathsf{arg}, V, K'\rangle & \text{if } E[\ ] = E'[(V\ [\ ])] \\ \langle \mathsf{narg}, \langle V_i, \ldots, V_1, o\rangle, \langle N, \ldots\rangle, K'\rangle & \text{if } E[\ ] = E'[(o\ V_1 \ldots V_i\ [\ ]\ N \ldots)] \\ \quad \text{where } K' = \mathcal{CK}(E'[\ ]) & \end{cases}$$

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 3.2.7 $eval_v^{scc} = eval_v^{ck}$

Proof. Based on the translations between states, it is easy to check that

1. if $\langle M, E[\ ]\rangle \longmapsto_{scc} \langle N, E'[\ ]\rangle$ then $\langle M, \mathcal{CK}(E[\ ])\rangle \longmapsto_{ck} \langle N, \mathcal{CK}(E'[\ ])\rangle$;
2. if $\langle M, K\rangle \longmapsto_{ck} \langle N, K'\rangle$ then $\langle M, \mathcal{KC}(K)\rangle \longmapsto_{scc} \langle N, \mathcal{KC}(K')\rangle$.

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

$$\langle M, [\ ]\rangle \longmapsto_{scc}^{*} \langle V, [\ ]\rangle \text{ if and only if } \langle M, \mathsf{mt}\rangle \longmapsto_{ck}^{*} \langle V, \mathsf{mt}\rangle,$$

which implies the theorem. ∎

Exercise 3.2.3 Modify the SCC-transition function $\longmapsto_{cc}$ of exercise 2 into an implementation of the CK-transition function. Measure the speed-up of $eval_v^{ck}$ over $eval_v^{scc}$. ∎

3.2.4 CEK Machine

Starting with the textual machine, the reduction of a β_v-redex requires the substitution of a value for all occurrences of an identifier. When the result of the substitution operation is not a value, it becomes the control string until it is reduced to a value. Then the machine traverses many of the applications that the substituition operation just traversed. To avoid this repeition of work, we delay the substitution until needed. That is, 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 is an *environment*. Of course, environments cannot map varibales to expressions because this would re-introduce expressions into the machine, but must map variables to closures.

Definition 3.2.8 (*Environments; Closures*) The sets of environments and closures are defined by mutual induction:

1. An environment is a finite function from variables to value closures. If $x_1, \ldots, x_n$ are distinct variables and $cl_1, \ldots, cl_n$ are value closures (for $n \geq 0$), then

$$\{\langle x_1, cl_1\rangle, \ldots, \langle x_n, cl_n\rangle\}$$

is an *environment*.

Given $E = \{x_1, \ldots, x_n\}$, $E(x_i) = cl_i$ and E's *domain*, written: $Dom(E)$ is $\{x_1, \ldots, x_n\}$.

2. Let M be an ISWIM expression and let E be an environment such that $FV(M) \subseteq Dom(E)$. Then,

$$\langle M, E\rangle$$

is a *closure*. If $M \in \mathit{Values}$, then it is a *value* closure.

If E is an environment and x a variable, then *shadowing* x in E, notation: $E \setminus x$, yields the environment that is like E but lacks a definition for x:

$$E \setminus x \overset{df}{\Leftrightarrow} \{\langle y, E(y)\rangle \mid y \in Dom(E), y \neq x\}.$$

If E is an environment, x a variable, and cl a closure, then the *update* of E at x to cl, notation: $E[x \leftarrow cl]$, yields an environment that now maps x to cl:

$$E[x \leftarrow cl] \overset{df}{\Leftrightarrow} (E \setminus x) \cup \{\langle x, cl\rangle\}.$$

∎

Based on the notions of closures and environments, it is easy to reformulate the CK machine into a machine that works on control strings with environments and continuation codes: the CEK machine.

Definition 3.2.9 (*The CEK machine*)
State space:

$$\mathit{Closures} \times \mathit{EKCodes}$$

where the set *EKCodes* of CEK continuation codes is defined by the grammar:

$$\begin{array}{rcl} K & ::= & \mathsf{mt} \\ & & \mid \langle \mathsf{fun}, vcl, K\rangle \\ & & \mid \langle \mathsf{arg}, cl, K\rangle \\ & & \mid \langle \mathsf{narg}, \langle v, \ldots, v, o\rangle, \langle cl, \ldots\rangle, K\rangle \end{array}$$

with vcl ranging over value closures and cl over closures.

State transitions:

Redex		*Contractum*	*cek*
$\langle\langle(MN), E\rangle, K\rangle$	$\longmapsto$	$\langle\langle M, E\rangle, \langle \mathsf{arg}, \langle N, E\rangle, K\rangle\rangle$	1
$\langle\langle(o\ M\ N\ldots), E\rangle, K\rangle$	$\longmapsto$	$\langle\langle M, E\rangle, \langle \mathsf{narg}, \langle o\rangle, \langle\langle N, E\rangle, \ldots\rangle, K\rangle\rangle$	2
$\langle\langle V, E\rangle, \langle \mathsf{fun}, \langle(\lambda x.M), E'\rangle, K\rangle\rangle$ if $V \notin Vars$	$\longmapsto$	$\langle\langle M, E'[x \leftarrow \langle V, E\rangle]\rangle, K\rangle$	3
$\langle\langle V, E\rangle, \langle \mathsf{arg}, N, K\rangle\rangle$ if $V \notin Vars$	$\longmapsto$	$\langle N, \langle \mathsf{fun}, \langle V, E\rangle, K\rangle\rangle$	4
$\langle\langle b, E\rangle, \langle \mathsf{narg}, \langle\langle b_i, E_i\rangle \ldots \langle b_1, E_1\rangle\ o\rangle, \langle\rangle\rangle, K\rangle$	$\longmapsto$	$\langle V, K\rangle$ where $\delta(o, b_1, \ldots, b_i, b) = V$	5
$\langle\langle V, E\rangle, \langle \mathsf{narg}, \langle cl', \ldots\rangle, \langle\langle N, E'\rangle, cl, \ldots\rangle, K\rangle\rangle$ if $V \notin Vars$	$\longmapsto$	$\langle\langle N, E'\rangle, \langle \mathsf{narg}, \langle\langle V, E\rangle, cl', \ldots\rangle, \langle cl, \ldots\rangle, K\rangle\rangle$	6
$\langle\langle x, E\rangle, K\rangle$	$\longmapsto$	$\langle cl, K\rangle$ where $E(x) = cl$	7

The evaluation function:

$$eval_v^{cek}(M) = \begin{cases} b & \text{if } \langle M, \mathsf{mt}\rangle \longmapsto^*_{ck} \langle b, \mathsf{mt}\rangle \\ \texttt{closure} & \text{if } \langle M, \mathsf{mt}\rangle \longmapsto^*_{ck} \langle\langle \lambda x.N, E\rangle, \mathsf{mt}\rangle \end{cases}$$

∎

Given a CEK-state it is again straightforward to find a corresponding CK-state. The basic idea is to replace every closure in the state by the closed term that it represents:

$$\begin{aligned} &unload : Closures \longrightarrow \Lambda^0 \\ &unload(\langle M, E\rangle) = M[x_1 \leftarrow unload(cl_1)] \ldots [x_n \leftarrow unload(cl_n)] \\ &\qquad \text{where } E = \{\langle x_1, cl_1\rangle, \ldots, \langle x_n, cl_n\rangle\} \end{aligned}$$

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:

$$\mathcal{KK} : KCodes \longrightarrow EKCodes$$

$$\mathcal{KK}(K) = \begin{cases} [\ ] & \text{if } K = \mathsf{mt} \\ \langle \mathsf{arg}, unload(cl), K^*\rangle & \text{if } K = \langle \mathsf{arg}, cl, K'\rangle \\ \langle \mathsf{fun}, unload(cl), K^*\rangle & \text{if } K = \langle \mathsf{fun}, cl, K'\rangle \\ \langle \mathsf{narg}, \langle unload(cl'), \ldots o\rangle, \langle unload(cl), \ldots\rangle, K^*\rangle & \\ \qquad \text{if } K = \langle \mathsf{narg}, \langle cl', \ldots o\rangle, \langle cl, \ldots\rangle, K'\rangle & \end{cases}$$
$$\text{where } K^* = \mathcal{KK}(K')$$

Unfortunately it is impossible to define inverses of these 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 subsections, 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 3.2.10 $eval_v^{ck} = eval_v^{cek}$

Proof. Given the translation from CEK-states it is easy to check that for any intermediate state $\langle cl, K\rangle$ in the sequence of CEK-states, there is a state $\langle cl', K'\rangle$ such that

$$\langle cl, K\rangle \longmapsto_{ck}^{*} \langle cl', K'\rangle$$

and

$$\langle unload(cl), \mathcal{KK}(K)\rangle \longmapsto_{ck} \langle unload(cl', \mathcal{KK}(K'))\rangle.$$

Hence, $\langle\langle M, \emptyset\rangle, \mathsf{mt}\rangle \longmapsto_{ck}^{*} \langle\langle V, E\rangle, \mathsf{mt}\rangle$ implies $\langle M, \mathsf{mt}\rangle \longmapsto_{ck} \langle unload(\langle V, E\rangle), \mathsf{mt})\rangle$, which proves the right-to-left direction.

The left-to-right direction requires a stronger induction hypothesis than the right-to-left 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 $\langle M_1, K_1\rangle$ and CEK-state $\langle\langle M_1', E\rangle, K_1'\rangle$ such that $M_1 = unload(\langle M_1', E\rangle)$ and $K_1 = \mathcal{KK}(K_1')$, if
>
> $$\langle M_1, K_1\rangle \longmapsto_{ck}^{*} \langle V, \mathsf{mt}\rangle$$
>
> then for some closure cl with $unload(cl) = V$,
>
> $$\langle\langle M_1', E\rangle, K_1'\rangle \longmapsto_{ck}^{*} \langle cl, \mathsf{mt}\rangle.$$

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

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:

$M_1 = (L\ N)$**:** The CK-machine performs a (ck.1) instruction:

$$\langle (L\ N), K_1\rangle \longmapsto_{ck} \langle L, \langle \mathsf{arg}, N, K_1\rangle\rangle.$$

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

$$\langle\langle (L'\ N'), E\rangle, K_1'\rangle \longmapsto_{ck} \langle\langle L', E\rangle, \langle \mathsf{arg}, \langle N', E\rangle, K_1'\rangle\rangle.$$

Clearly, $\mathcal{KK}(\langle \mathsf{arg}, \langle N', E\rangle, K_1'\rangle) = \langle \mathsf{arg}, N, K_1\rangle$. Thus the inductive hypothesis applies and proves the claim for this case.

$M_1 \in \mathit{Values}, K_1 = \langle \mathsf{fun}, (\lambda x.N), K_2 \rangle$: Here the CK-transition sequence begins as follows:

$$\langle M_1, \langle \mathsf{fun}, (\lambda x.N), K_2 \rangle \rangle \longmapsto_{ck} \langle N[x \leftarrow M_1], K_2 \rangle.$$

Given the definition of *unload*, there are two kinds of CEK-states that can map to the given CK-state:

$M_1' \notin \mathit{Vars}$: Here the CEK-step directly corresponds to the CK-step:

$$\langle \langle M_1', E \rangle, \langle \mathsf{fun}, \langle (\lambda x.N), E' \rangle, K_2 \rangle \rangle \longmapsto_{ck} \langle \langle N, E'[x \leftarrow \langle M_1', E \rangle] \rangle, K_2 \rangle.$$

$M_1' \in \mathit{Vars}, \mathit{unload}(E(M_1')) = M_1$: 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:

$$\begin{array}{rcl} \langle \langle M_1', E \rangle, \langle \mathsf{fun}, \langle (\lambda x.N'), E' \rangle, K_2 \rangle \rangle & \longmapsto_{ck} & E(M_1'), \langle \mathsf{fun}, \langle (\lambda x.N), E' \rangle, K_2 \rangle \\ & \longmapsto_{ck} & \langle \langle N, E'[x \leftarrow E(M_1')] \rangle, K_2 \rangle. \end{array}$$

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

$$\mathit{unload}(\langle N', E'[x \leftarrow cl] \rangle) = N[x \leftarrow \mathit{unload}(cl)]$$

where $cl = \langle M_1', E \rangle$. From the definition of *unload*, if $E = \{\langle x_1, cl_1 \rangle, \ldots, \langle x_1, cl_1 \rangle\}$,

$$\mathit{unload}(\langle \lambda x.N', E' \rangle) = \lambda x.N'[x_1 \leftarrow \mathit{unload}(cl_1)] \ldots [x_n \leftarrow \mathit{unload}(cl_n)] = \lambda x.N,$$

which implies

$$N'[x_1 \leftarrow \mathit{unload}(cl_1)] \ldots [x_n \leftarrow \mathit{unload}(cl_n)] = N.$$

By the Substitution Lemma, subst lemma

$$\begin{array}{rcl} N[x \leftarrow \mathit{unload}(cl)] & = & N'[x_1 \leftarrow \mathit{unload}(cl_1)] \ldots [x_n \leftarrow \mathit{unload}(cl_n)][x \leftarrow \mathit{unload}(cl)] \\ & = & \mathit{unload}(\langle N', E[x \leftarrow cl] \rangle). \end{array}$$

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

The proof of the other cases are analoguous to preceding one. ∎

Exercise 3.2.4 Implement the CEK-transition function and the CEK-evaluator. Measure the speed-up of $eval_v^{cek}$ over $eval_v^{ck}$. ∎

Exercise 3.2.5 Choose a concrete representation for the environment component, *e.g.*, a stack, and introduce instructions for extending environments and looking up values. ∎

3.3 The Behavior of Programs

The Standard Reduction Theorem is also useful for the analysis of the behavior of programs.

3.3.1 Observational Equivalence

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

Lemma 3.3.1 (Activity Lemma) *Let M be a closed expression and let C be a context. If $eval_v(C[M])$ exists, then*

(i) either $eval_v(C[M'])$ 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

$$C[M'] \longmapsto_v^* E[M'];$$

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

$$C[M'] \longmapsto_v^* E[(M'\ U)];$$

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

$$C[M'] \longmapsto_v^* E[(o\ b_1\ \ldots\ M'\ \ldots\ b_m)].$$

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

$$C[M] \longmapsto_v^* V.$$

However, because M may be duplicated during β_v-reductions, the induction hypothesis needs to be strengtened. 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.

Definition 3.3.2 (*ISWIM Multi-Hole Contexts*) Let $[\ ]_i$ for all $i \in \mathbb{N}$ be new members of the alphabet distinct from variables, constants, λ, ., and (,).

The set of multi-hole *contexts* is a set of ISWIM expressions with some holes:

$$\begin{array}{rcl} C & ::= & [\ ]_1 \mid \ldots \mid [\ ]_i \mid \ldots \\ & & \mid M \\ & & \mid (C\ C) \mid (\lambda x.C) \mid (o^n\ C \ldots C) \end{array}$$

where $M \in \Lambda$, $x \in$ *Vars*. C, C', and similar letters range over multi-hole contexts.

If there is no need to index the holes individually, we use the notation $C[\cdot, \ldots, \cdot]$ to indicate that C ranges over context with m holes.; otherwise, we write $C[\ ]_1 \ldots [\ ]_m$. Filling the holes of a multi-hole context is like filling the hole of a single-hole context. ∎

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

Lemma 3.3.3 *Let $M_1, \ldots, M_m$ be closed expressions and let C be a context with m holes. If $eval_v(C[M_1, \ldots, M_m])$ exists, then*

(*i*) *either $eval_v(C[M'_1, \ldots, M'_m])$ 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, $1 \leq i \leq m$,*

(*a*) *if M_i is an application, then*

$$C[M'_1, \ldots, M'_m] \longmapsto_v^* E[M'_i];$$

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

$$C[M'_1, \ldots, M'_m] \longmapsto_v^* E[(M'_i\ U)];$$

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

$$C[M'_1, \ldots, M'_m] \longmapsto_v^* E[(o\ b_1\ \ldots\ M'\ \ldots\ b_m)].$$

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

$$C[M_1, \ldots, M_m] \longmapsto_v K \longmapsto_v^* V$$

for some expression K and value V. By Lemma 3.1.2, the program can be partitioned into an evaluation context E' and a redex $(P\ Q)$ such that

$$E'[(P\ Q)] = C[M_1, \ldots, M_m],$$

or into an evaluation context E' and a redex $(o\ c_1 \ldots c_{m+1})$ such that

$$E'[(o\ c_1 \ldots c_{m+1})] = C[M_1, \ldots, M_m].$$

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

how about the second case?

1. Some expression M_i contains, or is equal to, the redex $(P\ Q)$. Since $(P\ Q)$ 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,

$$E[\ ] = C[M'_1]_1, \ldots, [\ ]_i, \ldots, [M'_m]_m$$

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 P. Now the second clause of part (ii) holds: for all closed expressions $M'_1, \ldots, M'_m$,

   $$E[\ ] = C[M'_1, \ldots, [\ ]_i, \ldots M'_m]$$

   and U differs from Q only in occurrences of holes of 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 $Q = M_j$, 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 P is a λ-abstraction, $\lambda x.P'$, that may contain some of the expressions $M_1, \ldots, M_m$. Now:

   $$C[M_1, \ldots, M_m] = E'[((\lambda x.P')\ Q)] \longmapsto_v E'[P'[x \leftarrow Q]]$$

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

   $$C[M'_1, \ldots, M'_m] = E''[((\lambda x.P'')\ Q')] \longmapsto_v E''[P''[x \leftarrow Q']]$$

   for some evaluation context E'', expression P'', and value Q' that differ accordingly from E', P', and Q respectively. Since the expressions $M_1, \ldots, M_m$ are closed, there is some context C' of n holes and a sequence of indices $1 \leq j_1, \ldots, j_n \leq m$ such that

   $$E'[P'[x \leftarrow Q]] = C'[M_{j_1}, \ldots, M_{j_n}]$$

   and

   $$E''[P''[x \leftarrow Q']] = C'[M'_{j_1}, \ldots, M'_{j_n}].$$

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

$$C'[M_{j_1}, \ldots, M_{j_n}] \longmapsto_v^* V$$

is shorter, and therefore the induction hypothesis applies. ∎

Equipped with the Activity Lemma, we are ready to finish Theorem 2.6.4 with the following lemma. It shows the observational equivalence of two expressions that are not provably equal.

Lemma 3.3.4 $\Omega \simeq_v (\Omega\ (\lambda x.x))$

Proof. Let $C[\ ]$ be a context such that $C[\Omega]$ and $C[(\Omega\ (\lambda x.x))]$ are programs. Assume that $eval_v(C[\Omega])$ exists. By the Activity Lemma, it could be the case that for all closed expressions M there is some evaluation context such that

$$C[M] \longmapsto_v^* E[M].$$

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

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

$$eval_v(C[(\Omega\ (\lambda x.x))]) = eval_v(C[\Omega]).$$

This proves that $\Omega \simeq_v (\Omega\ (\lambda x.x))$ because C is arbitrary. ∎

some more interesting observationl equivalences?

$M \simeq \Omega$ if M is stuck

3.3.2 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:

$$M_1 \longmapsto_v M_2 \longmapsto_v \ldots \longmapsto_v M_m$$

such that $\longmapsto_v$ 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.*,

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

Adopting a machine-oriented view, programs for which $\longmapsto_v$ 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.

Definition 3.3.5 (*Stuck Applications; Stuck Programs*) An application M is *stuck* if

1. $M = (o^m\ b_1 \ldots b_m)$ iff $\delta(o^m, b_1, \ldots, b_m)$ is undefined; or
2. $M = (o^m\ b_1 \ldots (\lambda x.N) \ldots b_m)$; or
3. $M = (b\ V)$ (for $b \in BConsts$).

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

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

Lemma 3.3.6 *If M is a program (closed expression), then either*

1. *$M \longmapsto_v^* V$ for some value V;*
2. *$M \longmapsto_v^* N$ for some stuck program N;*
3. *for all N such that $M \longmapsto_v^* N$ there exists an L such that $N \longmapsto_v L$.*

Proof. Assume there exists an N such that $M \longmapsto_v^* N$ and $\longmapsto_v$ is undefined for N. That is,

$$M_1 \longmapsto_v M_2 \longmapsto_v M_3 \longmapsto_v \ldots \longmapsto_v M_m = N.$$

By the Unique Evaluation Context Lemma (Lemma 3.1.2), N is either a value, or there is an evaluation context $E[\ ]$, possibly a primitive o, 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. ∎

Need a lemma that says beta/delta do not introduce free variables

The following chapter will address a uniform, language-based treatment of stuck states.

Notes

The general idea for such a theorem is due to Curry and Feys [*ref:* Curry&Feys], who proved it for Church's pure λ-calculus.

Chapter 4

Functional Extensions and Alternatives

4.1 Complex Data in ISWIM

The set of Cons ISWIM expressions, called Λ, contains the following expressions:

$$\begin{array}{rcl} M & ::= & x \mid (\lambda x.M) \mid (M\ M) \\ & & \mid b \mid (o^n\ M \ldots M) \\ & & \mid (\mathsf{cons}\ M\ M) \mid \mathsf{car} M \mid \mathsf{cdr} M \end{array}$$

The set of Cons ISWIM values, $Vals_c \subseteq \Lambda$, is the collection of basic constants ($b \in BConsts$), variables ($x \in Vars$), λ-abstractions in Λ, and pairs of values:

$$V ::= b \mid x \mid \lambda x.M \mid (\mathsf{cons}\ V\ V).$$

In addition to V, U, V, W, X also range over values.
Also, $Vals^0 = \{V \in Vals \mid FV(V) = \emptyset\}$.

$$\begin{array}{lll} (\mathsf{car}\ (\mathsf{cons}\ V\ U)) & \mathbf{car} & V \\ (\mathsf{cdr}\ (\mathsf{cons}\ V\ U)) & \mathbf{cdr} & U \end{array}$$

Exercise: vectors of values,
Exercise: other constructors
Question: facilities for defining new constructors?

4.2 Complex Data and Procedures as Recognizable Data

Definition 4.2.1 (*Extended δ-function*) A δ-function f for the set of function symbols $\bigcup_{n \in \mathbb{N}} FConsts^n$ is a family of partial $(n+1)$-ary functions, f^n, for each n where

$FConsts^n \neq \emptyset$, such that f^n maps an n-ary functional constant and n basic constants to a closed value:

$$f^n : FConsts^n \times \underbrace{Vals^0 \times \ldots \times Vals^0}_{n} \longrightarrow Vals^0.$$

If $f(o, V_1, \ldots, V_n)$ does not exist, we also say o is undefined for $b_1, \ldots, b_n$. ∎

Now definable:

$$\begin{aligned} \mathsf{number?} &: Vals \longrightarrow Vals \\ \mathsf{procedure?} &: Vals \longrightarrow Vals \\ \mathsf{car} &: Vals \longrightarrow Vals \\ \ldots & \end{aligned}$$

4.3 Call-By-Name ISWIM and Lazy Constructors

4.3.1 Call-By-Name Procedures

$$((\lambda x.M)\ N)\beta M[x \leftarrow N]$$

Otherwise the song remains the same
show that the observational theory is not a sub-theory of the strict theory

4.3.2 Lazy Constructors

The set of Cons ISWIM values, $Vals_c \subseteq \Lambda$, is the collection of basic constants ($b \in BConsts$), variables ($x \in Vars$), λ-abstractions in Λ, and pairs of values:

$$V ::= b \mid x \mid \lambda x.M \mid (\mathsf{cons}\ M\ M).$$

In addition to V, U, V, W, X also range over values.
Also, $Vals^0 = \{V \in Vals \mid FV(V) = \emptyset\}$.

$$\begin{array}{lll} (\mathsf{car}\ (\mathsf{cons}\ M\ N)) & \mathbf{car} & M \\ (\mathsf{cdr}\ (\mathsf{cons}\ M\ N)) & \mathbf{cdr} & N \end{array}$$

Chapter 5

Simple Control Operations

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 the 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 "error" to the set of answers and to add a single clause to the definition of the evaluator:

$$eval(M) = \begin{cases} b & \text{if } M \longmapsto_v^* b \\ \texttt{closure} & \text{if } M \longmapsto_v^* \lambda x.N \\ \mathsf{error} & \text{if } M \longmapsto_v^* N \text{ and } N \text{ is stuck.} \end{cases}$$

By Lemma 3.3.6, 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 $(/\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner)$. What is needed is 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 "error" with the intention that the evaluation of error terminates the execution of the program. We treat this extended version of ISWIM in the first section of this chapter.

Once 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 constructs are highly desirable in practice and are provided by many programming languages. Indeed, these constructs are often used to exit from loops once the result has been detected or to implement some form of backtracking. After examining errors and error handlers, we take a brief look at simple control constructs like try and fail and Lisp's catch and throw.

5.1 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.

Definition 5.1.1 (*Error ISWIM: Syntax*) Let *Errors* be a set of terminals that does not overlap with *Vars*, *BConsts*, or *FConsts*, the sets of variables, basic and function constant symbols. The symbol error ranges over the elements of *Errors* but it is also used as if it were an element of *Errors*.

The set of Error ISWIM expressions, Λ_e, is the extension of Λ with the elements of *Error*:

$$M ::= x \mid (\lambda x.M) \mid (M\ M) \mid b \mid (o^n\ M \ldots M) \mid \mathsf{error} \qquad \text{for } \mathsf{error} \in \mathit{Errors}$$

The notions of *context*, *syntactic compatibility* (see Definitions 2.1.2 and 2.1.3, and Proposition 2.1.4) and *syntactic equality* are adapted *mutatis mutandis* from Λ. ∎

5.1.1 Calculating with Error ISWIM

Roughly speaking, if a sub-expression raises an error signal, the error should eliminate any computation that would have beeen executed if the evaluation of the sub-expression had not raised an error signal. A precise specification of the behavior of error in terms of calculation rules requires an analysis of how error interacts with other syntactic constructions. Thus, take application as an example. If a function is applied to error, the appliction must be terminated. We can express this with the following informal notions of reduction

$$((\lambda x.M)\ \mathsf{error}) \longrightarrow \mathsf{error}; \qquad (o\ \mathsf{error}) \longrightarrow \mathsf{error}.$$

The rule also clarifies that error is *not* an ordinary value; otherwise applications like $((\lambda x.\ulcorner 5 \urcorner)\ \mathsf{error})$ would reduce to $\ulcorner 5 \urcorner$. The case for error in function position is similarly straightforward. When error occurs in function position, there is no function to be applied so the application also reduces to error:

$$(\mathsf{error}\ M) \longrightarrow \mathsf{error}.$$

Next we need to consider the situation when error occurs as the body of a procedure body. Since a procedure like $\lambda x.\mathsf{error}$ may never be invoked, the error should not propagate through the abstraction. The hidden error should only become visible when the λ-abstraction is applied to a value. Technically speaking, $\lambda x.\mathsf{error}$ is irreducible.

Given that the evaluation context surrounding a redex represents the rest of the computation, a natural formalization of the behavior of error says that it erases evaluation contexts. Thus, we introduce the following error reduction:

$$E[\mathsf{error}] \ \longrightarrow\ \mathsf{error} \qquad (\mathsf{error})$$

In addition to occurrences of error 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:

$$(b\ V) \longrightarrow \mathsf{error}_b.$$

The question is whether the generated error should also depend on the nature of V. Suppose an application of b to U for $V \neq U$ reduces to a distinct error, say, error'_b. Then, if it is also the case that $V \longrightarrow\!\!\!\!\!\!\rightarrow U$, the calculus proves

$$\mathsf{error}_V = (b\ V) = (b\ U) = \mathsf{error}_U.$$

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: After all, the argument may actually be the intended one.

If the set of errors is rich enough, we can reserve one error per basic constant to signal this kind of error.

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

$$(+\ (\lambda xy.x)\ 0) \longrightarrow \mathsf{error}_+$$

and

$$(+\ (\lambda xy.y)\ 0) \longrightarrow \mathsf{error}_+,$$

independent of the argument.

Finally, we need to 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 demand that the δ functions return appropriate error messages for bad inputs of a function. Thus, the interpretation of good and bad data is combined in a single definition. We can now give a formal definition of the calculus and the evaluator for *Error ISWIM*.

Definition 5.1.2 (*The Error ISWIM Calculus and Evaluator*) The set of Error ISWIM values, $Vals_e$, is the collection of basic constants ($b \in BConsts$), variables ($x \in Vars$), and λ-abstractions:

$$V ::= b \mid x \mid \lambda x.M.$$

The set of *evaluation contexts* for Error ISWIM is the following set of contexts:

$$E ::= [\ ] \mid (V\ E) \mid (E\ M) \mid (o\ V \ldots V\ E\ M \ldots M)$$

where V is a value and M is an arbitrary expression.

For Error ISWIM a δ-function f is a family of total $(n+1)$-ary functions, f^n, for each n where $FConsts^n \neq \emptyset$, such that f^n maps an n-ary functional constant and n basic constants to a closed value or an error element:

$$f^n : FConsts^n \times \underbrace{BConsts \times \ldots \times BConsts}_{n} \longrightarrow Vals^0 \cup Errors.$$

An application M is *faulty* if

1. $M = (o^m\ b_1\ \ldots\ b_m)$ and $f^m(o^m, b_1, \ldots, b_m) \in Errors$; or
2. $M = (o^m\ V_1 \ldots\ (\lambda x.N)\ \ldots V_m)$; or
3. $M = (b\ V)$ (for $b \in BConsts$).

The basic notions of reduction for Error ISWIM are:

δ : $(o^m\ b_1\ \ldots\ b_m) \delta V$ iff $f^m(o^m, b_1, \ldots, b_m) = V : Vals^0$

β_v : $((\lambda x.M)\ V)\quad \beta_v \quad M[x \leftarrow V]$

δ_{error} : Fix an error element error_b for each basic constant b and an error constant error_o for each primitive operation o. Then, $M \delta_{error} \mathsf{error}$ iff one of the following three conditions holds:

- $(b\ V) \delta_{error} \mathsf{error}_b$;
- $(o^m\ V_1 \ldots\ (\lambda x.N)\ \ldots V_m) \delta_{error} \mathsf{error}_o$;
- $(o^m\ b_1\ \ldots\ b_m) \delta_{error} \mathsf{error}$ iff $f^m(o^m, b_1, \ldots, b_m) = \mathsf{error}$.

error : $E[\mathsf{error}]$ **error** error

The complete notion of reduction is the union of these relations:

$$\mathbf{e} \;=\; \beta_v \cup \delta \cup \mathbf{error} \cup \delta_{error} \tag{e}$$

The *one-step* **e**-*reduction* $\longrightarrow_e$ is the compatible closure of **e**. Next, $\twoheadrightarrow_e$ is the **e**-*reduction*; it is the reflexive, transitive closure of $\longrightarrow_e$. Finally, $=_e$ is the smallest equivalence relation generated by $\longrightarrow_e$. If $M_1 =_e M_2$, we also write $\boldsymbol{\lambda_v}\text{-}\mathbf{e} \vdash M_1 = M_2$ to emphasize the fact that the calculus is an equational proof system.

The set of Error ISWIM answers is the set of basic constants plus the symbols `closure` and error:

$$A = BConsts \cup Errors \cup \{\texttt{closure}\}.$$

The partial function $eval_e : \Lambda_e^0 \longrightarrow A$ is defined as follows:

$$eval_e(M) = \begin{cases} b & \text{if } M =_e b \\ \texttt{closure} & \text{if } M =_e \lambda x.N \\ \mathsf{error} & \text{if } M =_e \mathsf{error} \end{cases}$$

∎

Next we need to check that the addition of error does not destroy the key properties of the evaluator. To begin with, we would like to know that the evaluator and thus the programming language are still functional. That is, our goal is to establish a consistency theorem for Error ISWIM.

Theorem 5.1.3 (Consistency) *The relation $eval_e$ is a partial function.*

The proof of this theorem has the same structure as the proof of Theorem 2.5.1, the corresponding theorem for $eval_v$. We first prove a Church-Rosser property, that is, we showq 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 error. 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 **e**.

We first deal with the extension of the ISWIM calculus to the larger syntax. Let $\mathbf{w} = \delta \cup \beta_v$, *i.e.*, the extension of **v** to Λ_e, the full set of Error ISWIM expressions. The notation $\longrightarrow_w$ and $\twoheadrightarrow_w$ denote the one-step reduction and its transitive-reflexive closure, respectively.

Lemma 5.1.4 (Diamond Property for $\twoheadrightarrow_w$) *Let L, M, and N be Error ISWIM expressions. If $L \twoheadrightarrow_w M$ and $L \twoheadrightarrow_w N$ then there exists an expression K such that $M \twoheadrightarrow_w K$ and $N \twoheadrightarrow_w K$.*

Next we turn to the analysis of the two new notions of reduction. Let

$$\mathbf{f} = \mathbf{error} \cup \delta_{error}.$$

Unlike $\mathbf{v}$ or $\mathbf{w}$, this notion of reduction never duplicates expressions or creates new opportunities to reduce sub-expressions. We should therefore expect that the one-step relation satisfies the diamond property, though unfortunately, it does not hold. Consider the reduction from $E[\mathsf{error}]$ to error. If $E[\mathsf{error}]$ also reduces to $E'[\mathsf{error}]$ for some evaluation context $E'[\ ]$, then $E'[\mathsf{error}]$ directly reduces to error. Hence it is not the one-step reduction $\longrightarrow_f$ but its reflexive closure $\longrightarrow_f^0$ that satisfies the diamond property.

Lemma 5.1.5 (Diamond Property for $\longrightarrow\!\!\!\!\!\rightarrow_f$) *Let L, M, and N be Error ISWIM expressions. If $L \longrightarrow\!\!\!\!\!\rightarrow_f M$ and $L \longrightarrow\!\!\!\!\!\rightarrow_f N$ then there exists an expression K such that $M \longrightarrow\!\!\!\!\!\rightarrow_f K$ and $N \longrightarrow\!\!\!\!\!\rightarrow_f K$.*

Proof. Following the discussion leading up to this lemma, we first prove the claim for a subset of the relation: If $L \longrightarrow_f^0 M$ and $L \longrightarrow_f^0 N$ then there exists an expression K such that $M \longrightarrow_f^0 K$ and $N \longrightarrow_f^0 K$. The lemma clearly follows from this claim. The proof proceeds by case analysis of the reduction from L to M:

L is faulty; $M = \mathsf{error}$: Clearly, if L is faulty and $L \longrightarrow_f N$ then N is faulty. Hence, $K = \mathsf{error}$ is the desired fourth expression.

$L = E[\mathsf{error}]; M = \mathsf{error}$: It is easy to check that any $\mathbf{f}$-reduction applied to $E[\mathsf{error}]$ yields an expression of the shape $E'[\mathsf{error}]$ for some evaluation context $E'[\ ]$, which implies that the reduction **error** applies to N.

$L = M$: Set $K = N$.

$L = C[L'], M = C[\mathsf{error}], (L', \mathsf{error}) \in \mathbf{f}$: If $N = C'[L']$ for some context $C'[\ ]$, then $K = C'[\mathsf{error}]$ because $C[L'] \longrightarrow_f C'[L']$ implies that for all N', $C[N'] \longrightarrow_f C'[N']$. Otherwise $N = C'[\mathsf{error}]$ by the definition of compatibility, in which case $K = M = N$. ∎

Exercise 5.1.1 Prove: If L is faulty and $L \longrightarrow_f N$ then N is faulty and reduces to the same error element. ∎

Exercise 5.1.2 Prove: If $L = E[\mathsf{error}]$ and $L \longrightarrow_f M$ then there exists an evaluation context $E'[\ ]$ such that $M = E'[\mathsf{error}]$. ∎

After establishing that each of the notions of reduction $\mathbf{w}$ and $\mathbf{f}$ on Λ_e satisfy the diamond property, we need to show that these results can be combined. Intuitively, the combination of $\mathbf{w}$ and $\mathbf{f}$ 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.

Definition 5.1.6 (*Commuting Reductions*) Let $\longrightarrow\!\!\!\!\!\rightarrow_r$ and $\longrightarrow\!\!\!\!\!\rightarrow_s$ be relations on some set A. The relations commute if for all a, b, and c,

$$a \longrightarrow\!\!\!\!\!\rightarrow_r b \quad \text{and} \quad a \longrightarrow\!\!\!\!\!\rightarrow_s c$$

implies the existence of d such that

$$b \longrightarrow\!\!\!\!\!\rightarrow_s d \quad \text{and} \quad c \longrightarrow\!\!\!\!\!\rightarrow_r d.$$

∎

If we can show that the one-step relations based on **w** and **f** commute, the commutation for their transitive closures clealry follows. However, the one-step relations do not commute. Consider the expression

$$((\lambda f.(f\ (f\ \ulcorner 0 \urcorner)))\ (\lambda x.\underline{(1^+\ (\lambda x.x))})).$$

Reducing the underlined δ_{error}-redex first yields

$$((\lambda f.(f\ (f\ \ulcorner 0 \urcorner)))\ (\lambda x.\mathsf{error}_{1^+})) \longrightarrow_e ((\lambda x.\mathsf{error}_{1^+})\ ((\lambda x.\mathsf{error}_{1^+})\ \ulcorner 0 \urcorner));$$

but reducing the β_v-redex first requires *two* δ_{error}-reductions to reach the same state:

$$\begin{aligned} ((\lambda x.\underline{(1^+\ (\lambda x.x))})\ ((\lambda x.\underline{(1^+\ (\lambda x.x))})\ \ulcorner 0 \urcorner)) &\longrightarrow_e ((\lambda x.\mathsf{error}_{1^+})\ ((\lambda x.\underline{(1^+\ (\lambda x.x))})\ \ulcorner 0 \urcorner)) \\ &\longrightarrow_e ((\lambda x.\mathsf{error}_{1^+})\ ((\lambda x.\mathsf{error}_{1^+})\ \ulcorner 0 \urcorner)). \end{aligned}$$

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 5.1.7 *The reductions* $\longrightarrow\!\!\!\!\!\rightarrow_w$ *and* $\longrightarrow\!\!\!\!\!\rightarrow_f$ *commute.*

Proof. The first step of the proof shows that for all $L, M, N \in \Lambda_e$ such that $L \longrightarrow_w M$ and $L \longrightarrow_f N$ there exists a $K \in \Lambda_e$ such that

$$M \longrightarrow\!\!\!\!\!\rightarrow_f K \text{ and } N \longrightarrow_w^0 K.$$

The proof of the claim is an induction on the derivation of $L \longrightarrow_w M$:

$L = ((\lambda x.L')\ V), M = L'[x \leftarrow V]$: Only two subcases are possible because a value cannot reduce to error:

$N = ((\lambda x.L'')\ V), L' \longrightarrow_f L''$: If we can show that $M[x \leftarrow V] \longrightarrow_f M'[x \leftarrow V]$ if $M \longrightarrow_f M'$, then we can take $K = L''[x \leftarrow V]$.

$N = ((\lambda x.L')\ V'), V \longrightarrow_f V'$: If we can show that $M[x \leftarrow V] \twoheadrightarrow_f M[x \leftarrow V']$ if $V \longrightarrow_f V'$, then we can take $K = L'[x \leftarrow V']$.

$L \delta M$: Then, by definition of δ and δ_{error}, it is impossible that $L \longrightarrow_f N$.

$L = C[L'], M = C[M'], (L', M') \in \mathbf{w}$: If $C[L'] \longrightarrow_f C[L'']$, the claim follows from the inductive hypothesis. Otherwise, $C[L'] \longrightarrow_f C'[L']$ 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 5.1.8 (Diamond Property for $\twoheadrightarrow_e$) *Let L, M, and N be Error ISWIM expressions. If $L \twoheadrightarrow_e M$ and $L \twoheadrightarrow_e N$ then there exists an expression K such that $M \twoheadrightarrow_e K$ and $N \twoheadrightarrow_e K$.*

Proof. For the cases where $L = M$ or $L = N$, the lemma obviously holds. Thus assume $L \neq M, L \neq N$. 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:

$L \twoheadrightarrow_f M_1, L \twoheadrightarrow_w N_1$: Here the lemma follows by the commutation property for the two reductions.

$L \twoheadrightarrow_w M_1, L \twoheadrightarrow_f N_1$: Again, the lemma is a consequence of the commutation property.

$L \twoheadrightarrow_f M_1, L \twoheadrightarrow_f N_1$: The dimaond property for $\twoheadrightarrow_f$ implies the lemma.

$L \twoheadrightarrow_w M_1, L \twoheadrightarrow_w N_1$: This case holds due to the dimaond property for $\twoheadrightarrow_w$. ∎

5.1.2 Standard Reduction for Error ISWIM

As discussed in Chapter 3, a specification of the evaluator based on a calculus does not lend itself 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, the deterministic ISWIM evaluator partitions the program into an evaluation context E and a **v**-redex M. If N is the contractum of M, the next state in the evaluation sequence is $E[N]$, *i.e.*,

$$E[M] \longmapsto_v E[N].$$

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 **v**-redex.

Our goal in this section is to prove that the ISWIM strategy also works for Error ISWIM. By a straightforward adaptation, the four kinds of standard reduction steps should be

$$\begin{array}{rcl} E[(o^m\ b_1 \ldots b_m)] & \longmapsto_e & E[V] \quad \text{if } f^m(o^m, b_1, \ldots, b_m) = V \\ E[((\lambda x.M)\ V)] & \longmapsto_e & E[M[x \leftarrow V]] \\ E[M] & \longmapsto_e & E[\mathsf{error}] \quad \text{if } M\,\delta_{error}\,\mathsf{error} \\ E[E'[\mathsf{error}]] & \longmapsto_e & E[\mathsf{error}] \end{array}$$

The last standard reduction transition in this set obviously introduces a redundant transition into the textual machine. The resulting program is immediately the redex for the same kind of standard reduction step. It is therefore better to simplify this transition so that these two steps are merged into one:

$$E[\mathsf{error}] \quad \longmapsto_e \quad \mathsf{error}$$

Definition 5.1.9 (*Standard Reduction Function for Error ISWIM*) The *standard reduction function* for Error ISWIM maps closed non-values to expressions:

$$\begin{array}{ll} M \longmapsto_e N \text{ iff for some } E, & M \equiv E[M'], N \equiv E[N'], \text{ and } (M', N') \in \delta \cup \beta_v \cup \delta_{error} \\ & M \equiv E[\mathsf{error}], \text{ and } N = \mathsf{error} \end{array}$$

The partial function $eval_e^s : \Lambda_e^0 \longrightarrow A$ is defined as follows:

$$eval_e^s(M) = \begin{cases} b & \text{if } M \longmapsto_e^* b \\ \texttt{closure} & \text{if } M \longmapsto_e^* \lambda x.N \\ \mathsf{error} & \text{if } M \longmapsto_e^* \mathsf{error} \end{cases}$$

∎

As we know from the previous chapters, the definition specifies a function only if the partitioning of a program into an evaluation context and a redex is unique. For the literal meaning of redex, namely, **e**-redex, this property no longer holds because $E[\mathsf{error}]$ is a redex for every evaluation context $E[\ ]$. However, if, in this context only, redex means error or one of the other redexes, we can prove the desired property.

Definition 5.1.10 (*Standard Redex*) The set of Error ISWIM *standard redexes* is generated by the grammar

$$R ::= (V_1\ V_2) \mid (o^m\ V_1\ \ldots\ V_m) \mid \mathsf{error}$$

where $V_1, \ldots, V_n$ are values and o^m is a primitive operator. ∎

Lemma 5.1.11 (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 R such that $M = E[R]$.*

Proof. The proof is similar to that of Lemma 3.1.2. ∎

The lemma validates that the new specification of the evaluator, $eval^s_e$, 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 5.1.12 $eval_e = eval^s_e$

Exercise 5.1.3 Adapt the proof of Theorem 3.1.8 to prove Theorem 5.1.12. ∎

Exercise 5.1.4 Extend the implementation of Exercise 1 to Error ISWIM. ∎

Exercise 5.1.5 If *Errors* contains only one element, a feasible alternative for propagating error through applications is the following notion of reduction:

$$A[\mathsf{error}] \ \ \mathbf{error} \ \ \mathsf{error}.$$

Here A ranges over the set of *application* contexts:

$$A ::= [\ ] \mid (M\ A) \mid (A\ M) \mid (o^m\ M_1 \ldots A \ldots M_m).$$

Check that the theorems in the preceding two subsections still hold. What is unusual about the standard reduction function? What breaks when *Errors* contains at least two elements? ∎

5.1.3 The relationship between Iswim and Error ISWIM

upward compatible?

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 programs may terminate with an error signal.

Theorem 5.1.13 *Let M be an ISWIM program. Then,*

1. *$eval_v(M) = a$ implies that $eval_e(M) = a$;*
2. *$eval_e(M) = a$ and $a \neq \mathsf{error}$ implies that $eval_v(M) = a$;*
3. *$eval_e(M) \in Errors$ implies that $eval_v(M)$ diverges; and*

4. *if $eval_e(M)$ is undefined, then $eval_v(M)$ is undefined.*

Proof. For the proof of the first claim assume $\boldsymbol{\lambda_v} \vdash M = V$. Since $\mathbf{v}$ interpreted on Λ_e is a strict subset of $\mathbf{e}$, it is clearly true that $\boldsymbol{\lambda_v}$-$\mathbf{e} \vdash M = V$. For the proof of the second claim, recall that by Theorem 5.1.12, $eval_e = eval_e^s$. Hence, assume $M \longmapsto_e^* V$. The assumption implies $V \neq \mathsf{error}$. But then none of the standard reduction steps from M to V can be a step based on δ_{error} or **error**. We prove this statement by induction on the length of the standard reduction sequence from M to V. If $M = V$, we are done. Thus assume

$$M \longmapsto_e M_1 \longmapsto_e^* V.$$

We know that $M = E[M']$ for some evaluation context E and an $\mathbf{e}$-redex M'. If $M' \delta_{error} \mathsf{error}$ or $M' = \mathsf{error}$, then $M \longmapsto_e \mathsf{error}$, contradicting $V \neq \mathsf{error}$. Hence, $M \longmapsto_v M_1$ and by inductive hypothesis $M_1 \longmapsto_v^* V$. In summary,

$$eval_v(M) = a.$$

The last two claims follow from the observation that if for some ISWIM program M, $M \longmapsto_e^* N$ then $M \longmapsto_v^* N$ and N is in ISWIM if the standard transition reduces a β_v or a δ redex. Hence, if $M \longmapsto_e^* \mathsf{error}$ then $M \longmapsto_v^* E[N] \longmapsto_e E[\mathsf{error}] \longmapsto_e \mathsf{error}$ for some evaluation context E and a faulty application N. Hence, $eval_v(M)$ is undefined. Finally, if $M \longmapsto_e^* N$ implies that $N \longmapsto_e L$, then all transitions are according to δ or β_v, and thus $eval_v(M)$ 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 error 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.

Definition 5.1.14 (*Observational Equivalence for Error ISWIM*) Two Error ISWIM expressions, M and M', are *observationally equivalent*, written: $M \simeq_e M'$, if and only if

$$eval_e(C[M]) = eval_e(C[M'])$$

in all contexts C for which $C[M]$ and $C[M']$ are programs. ∎

The following theorem formalizes the discussed relationship between the observational equivalence relations of ISWIM and Error ISWIM.

Theorem 5.1.15 *Let M and N be ISWIM expressions. Then,*

1. *$M \simeq_e N$ implies $M \simeq_v N$; but*

2. *$M \simeq_v N$ does not imply $M \simeq_e N$.*

Proof. 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 Proposition **??**, which showed

$$(\lambda fx.((f\ x)\ \Omega)) \simeq_v (\lambda fx.\Omega).$$

But in Error ISWIM the two expressions are clearly distinguishable with

$$C = ([\ \ ]\ (\lambda x.\mathsf{error})\ (\lambda x.x)).$$

Whereas $eval_e(C[(\lambda fx.((f\ x)\ \Omega))]) = \mathsf{error}$, $eval_e(C[(\lambda fx.\Omega)])$ 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 5.1.16 *If $M, N \in \Lambda$ are such that $M \simeq_v N$ and $M \not\simeq_e N$, then there exists a context C over Error ISWIM such that $C[M]$ and CN are programs and one of the following conditions hold:*

1. *there are errors error_M and error_N with $\mathsf{error}_M \neq \mathsf{error}_N$ and*

$$eval(C[M]) = \mathsf{error}_M \text{ and } eval(C[N]) = \mathsf{error}_N;$$

2. *there is an error error_M and*

$$eval(C[M]) = \mathsf{error}_M \text{ and } eval(C[N]) \text{ diverges;}$$

3. *there is an error error_N and*

$$eval(C[M]) \text{ diverges and } eval(C[N]) = \mathsf{error}_N.$$

Proof. Let $M, N \in \Lambda$ be such that $M \simeq_v N$ and $M \not\simeq_e N$. 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,

$$eval(C[M]) = b_M, eval(C[N]) = b_N, \text{ and } b_M \neq b_N.$$

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 $\mathsf{error} \in Errors$, the answer must be

error. Thus, let C' be like C except that all occurrences of error elements are replaced by Ω. But then, by the following lemma,

$$eval(C[M]) = eval(C'[M]) = b_M$$

and

$$eval(C[N]) = eval(C'[N]) = b_N,$$

which means that some ISWIM context can already observe differences between M and N: $M \not\simeq_v N$. This contradiction to the assumptions of the lemma proves that the context C cannot exist. ∎

Exercise 5.1.6 Check the following case in the proof of Theorem 5.1.16. Assume C is an Error ISWIM context such that $eval(C[M]) = b_M$ for some basic constant b_M and $eval(C[N]) = \mathsf{error}_N$. ∎

We are 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 3.3.1) for Error ISWIM.

Lemma 5.1.17 *Let C be an m-hole context over Error ISWIM. If $eval(C[\mathsf{error}_1, \ldots, \mathsf{error}_m]) = a$ and $a \neq \mathsf{error}$, then $eval(C[\Omega, \ldots, \Omega]) = a$.*

Proof. By assumption, $C[\mathsf{error}_1, \ldots, \mathsf{error}_m] \longmapsto_e^* V$ for some value $V \neq \mathsf{error}$. 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,

$$C[\mathsf{error}_1, \ldots, \mathsf{error}_m] \longmapsto_e M \longmapsto_e^* V.$$

By the definition of standard reduction, there must be some evaluation context E and standard redex R such that

$$C[\mathsf{error}_1, \ldots, \mathsf{error}_m] = E[R].$$

The standard redex R cannot be an error element and it cannot be a δ_{error} redex because both would contradict the assumption. But then R reduces to some contractum N such that for some n-ary context C',

$$M = C'[\mathsf{error}_{i_1}, \ldots, \mathsf{error}_{i_n}]$$

where $i_1, \ldots, i_n$ is a (possibly repetitive) permutation of $1, \ldots, m$. This clearly means that

$$C[\Omega, \ldots, \Omega] \longmapsto_e C'[\Omega, \ldots, \Omega]$$

and, by inductive hypothesis,

$$C[\Omega, \ldots, \Omega] \longmapsto_e V'$$

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,

$$eval(C[\Omega, \ldots, \Omega]) = a. \quad \blacksquare$$

Exercise 5.1.7 Use the proof of Theorem 5.1.15 to show that Error ISWIM can reveal the order in which a procedure invokes its parameters that are procedures. Hint: Separate the cases where $Errors = \{\mathsf{error}\}$ and $Errors \supseteq \mathsf{error}_1, \mathsf{error}_2$ with $\mathsf{error}_1 \neq \mathsf{error}_2$. ∎

Exercise 5.1.8 Use Theorem 5.1.3 to prove that for any ISWIM expressions M and N, $\boldsymbol{\lambda_v} \vdash M = N$ implies $\boldsymbol{\lambda_v}\text{-}\mathsf{e} \vdash M = N$.

Conclude that for all $M, N \in \Lambda$, $\boldsymbol{\lambda_v}\text{-}\mathsf{e} \vdash M = N$ implies $M \simeq_v N$. Hint: Use the Church-Rosser Theorem (2.5.2) for the λ_v-calculus.

Also prove that $\boldsymbol{\lambda_v}\text{-}\mathsf{e} \vdash M = N$ implies $M \simeq_e N$. Hint: Compare Theorem 2.6.4. ∎

stuck states in ISWIM are equivalent to Omega (prove in 3!)

5.2 Error Handlers: Exceptions

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

$$(\mathsf{handle}\ B\ \mathsf{with}\ H)$$

where the expression B is the (proteceted) *body* and H 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.

Error 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 error, a program can attempt to compute the answer with the first and primary method but in such a way that when an error occurs, control is transferred to the secondary method of computation.

Provided error's come with additional information, the error handler's action may also depend on what kind of error occured. Indeed, this option increases the usefulness of error handlers dramatically because the handler not only knows that an error occurred but also what kind of error occurred. One easy way to implement this idea is to assume that errors are like primtive operations that are applied to a value. Here we assume that the errors are applied to basic constants, and that the basic constant in an error application is the information that a handler receives.

Definition 5.2.1 (*Handler ISWIM: Syntax*) Let error, handle, and with be new terminal tokens, distinct from all elements in *Vars*, *FConsts*, and *BConsts*.

The set of Handler ISWIM expressions, Λ_x, is:

$$\begin{array}{rcl} M & ::= & x \mid (\lambda x.M) \mid (M\ M) \\ & & \mid b \mid (o^n\ M \ldots M) \\ & & \mid (\mathsf{error}\ b) \\ & & \mid (\mathsf{handle}\ M\ \mathsf{with}\ \lambda x.M) \end{array}$$

The set of contexts for Handler ISWIM contains the following expressions with holes:

$$\begin{array}{rcl} C & ::= & [\ ] \mid (M\ C) \mid (C\ M) \mid (\lambda x.C) \\ & & \mid (o^n\ M_1\ \ldots\ M_i\ C\ M_{i+1}\ \ldots\ M_n) \text{ for } 0 \leq i \leq n \\ & & \mid (\mathsf{handle}\ C\ \mathsf{with}\ \lambda x.M) \mid (\mathsf{handle}\ M\ \mathsf{with}\ \lambda x.C) \end{array}$$

∎

For an illustration of the versatility of error handlers, we consider a procedure that multiplies the elements of a list of numbers. A naïve version of the procedure traverses the list and multiplies the elements one by one:

$$\begin{array}{lll} \Pi = \mathsf{Y}_v(\lambda\pi.\ \lambda l. & (\mathsf{if0} & (\mathsf{null?}\ l)\ \ulcorner 1 \urcorner \\ & & (*\ (\mathsf{car}\ l)\ (\pi\ (\mathsf{cdr}\ l))))) \end{array}$$

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 is going to be $\ulcorner 0 \urcorner$. Exiting can be implemented by raising an error that is handled by a surrounding handler:

$$\Pi^0 = \lambda l.(\mathsf{handle}\ (\Pi'\ l)\ \mathsf{with}\ \lambda x.x)$$

where

$$\begin{array}{llll} \Pi' = \mathsf{Y}_v(\lambda\pi.\ \lambda l. & (\mathsf{if0} & (\mathsf{null?}\ l) & \ulcorner 1 \urcorner \\ & (\mathsf{if0} & (\mathsf{car}\ l) & (\mathsf{error}\ 0) \\ & & (*\ (\mathsf{car}\ l)\ (\pi\ (\mathsf{cdr}\ l))))) & \end{array}$$

In this example, signaling an error indicates that the proper result was found and thus speeds up the ordinary computation.

5.2.1 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 now depends on the behavior of handle 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 handle expression. Othwerwise, if the second evaluation signals an error, the error is not propagated but 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:

$$(\mathsf{handle}\ U\ \mathsf{with}\ \lambda x.M) \longrightarrow U.$$

A notion of reduction that describes the error handling component of the behavior says that if the first sub-expression turns into an error element, its corresponding basic constant becomes the argument of the handler:

$$(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ \lambda x.M) \longrightarrow ((\lambda x.M)\ b).$$

No other reductions dealing with handle 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 handle expression: Otherwise error propagation through evaluation contexts would circumvent error handlers.

The notions of reduction for error elements have the same shape as the ones for Error ISWIM. The following definition introduces the calculus for Error ISWIM. It should be understood as an extension of Definition 5.1.2

Definition 5.2.2 (*The Handler ISWIM Calculus and Evaluator*) The set of values in Handler ISWIM and the set of evaluation contexts are as follows:

$$V \quad ::= \quad b \mid x \mid \lambda x.M$$

$$E \quad ::= \quad [\ ] \mid (V\ E) \mid (E\ M) \mid (o\,V \ldots V\ E\ M \ldots M)$$

Given a total δ-function f whose range includes expressions of the shape $\mathsf{error}\ b$, the basic notions of reduction are δ, β_v, δ_{error}, and **error** plus the relations that deal with handle expressions:

$\delta : (o^m\ b_1\ \ldots\ b_m)\delta V$ iff $f^m(o^m, b_1, \ldots, b_m) = V : \mathit{Vals}^0$

β_v : $((\lambda x.M)\ V) \quad \beta_v \quad M[x \leftarrow V]$

δ_{error} : Fix an error element error_b for each basic constant b and an error constant error_o for each primitive operation o. Then, $M \delta_{error} \mathsf{error}$ iff one of the following three conditions holds:

- $(b\ V)\delta_{error}\mathsf{error}\ \mathsf{error}_b$;
- $(o^m\ V_1 \ldots\ (\lambda x.N)\ \ldots V_m)\delta_{error}\mathsf{error}\ \mathsf{error}_o$;
- $(o^m\ b_1\ \ldots\ b_m)\delta_{error}\mathsf{error}\ b$ iff $f^m(o^m, b_1, \ldots, b_m) = \mathsf{error}\ b$.

error : $E[\mathsf{error}\ b] \quad \mathbf{error} \quad \mathsf{error}\ b$

return : $(\mathsf{handle}\ V\ \mathsf{with}\ \lambda x.M) \quad \mathbf{return} \quad V$

handle : $(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ \lambda x.M) \quad \mathbf{handle} \quad ((\lambda x.M)\ b)$.

The complete notion of reduction is the union of these relations:

$$\mathbf{h} \ = \ \delta \cup \beta_v \cup \mathbf{error} \cup \delta_{error} \cup \mathbf{handle} \cup \mathbf{return} \qquad (\mathbf{h})$$

All other concepts of the Error ISWIM calculus, *e.g.*, reduction and provable equality, are adapted *mutatis mutandis*.

The evaluator maps programs to basic constants, basic constants tagged with `error`, and `closure`:

$$eval_h(M) = \begin{cases} b & \text{if } M =_h b \\ \texttt{closure} & \text{if } M =_h \lambda x.N \\ \texttt{error}\ b & \text{if } M =_h \mathsf{error}\ b \end{cases}$$

∎

It is easy to show that the calculus defines an evaluator function.

Theorem 5.2.3 *The relation $eval_h$ is a partial function.*

Proof. The relations **handle** and **return** trivially satisfy the diamond property on Λ_h. It is also straightforward to show that the proof of the dimaond property for the notion of reduction **e** extends to the full syntax. If the reduction based on **e** commutes with the one based on **handle** and **return**, we have the diamond property for the full language. To prove the commutativity of **handle** and **return** with the old reductions, assume

$C[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ \lambda x.M)] \longrightarrow C[((\lambda x.M)\ b)]$: Then, a reduction step according to one of $\delta, \beta_v, \delta_{error}$, or **error** either modifies the context C or the handler V:

$C[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ \lambda x.M)] \longrightarrow C'[(\mathsf{handle}\ \mathsf{error}\ b\ \mathsf{with}\ \lambda x.M)]$: The common contractum for $C'[(\mathsf{handle}\ \mathsf{error}\ b\ \mathsf{with}\ \lambda x.M)]$ and $C[((\lambda x.M)\ b)]$ is $C'[((\lambda x.M)\ b)]$.

$C[(\textsf{handle}\ (\textsf{error}\ b)\ \textsf{with}\ \lambda x.M)] \longrightarrow C[(\textsf{handle}\ (\textsf{error}\ b)\ \textsf{with}\ \lambda x.M')]$: Here the expressions reduce to $C[(\textsf{handle}\ (\textsf{error}\ b)\ \textsf{with}\ \lambda x.M')]$.

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.

$C[(\textsf{handle}\ U\ \textsf{with}\ \lambda x.M)] \longrightarrow C[U]$: 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 $eval_h$ directly follow from the diamond property according to the familiar proof strategy. ∎

5.2.2 A Standard Reduction Function

The definition of a deterministic evaluation function for Handler ISWIM can no longer rely on the simple algorithm for ISWIM and Error ISWIM. If the leftmost-outermost redex of a program occurs inside of the body of a **handle** expression, the program cannot be the result of filling an evaluation context with the redex because evaluation contexts do not contain the bodies of handlers. For example, there is no evaluation context E such that the expression

$$(\textsf{handle}\ \underline{((\lambda x.x)\ \ulcorner 0 \urcorner)}\ \textsf{with}\ \lambda x.x)$$

is the result of filling E with the underlined redex.

Unlike in ISWIM or Error ISWIM, evaluation contexts in Handler ISWIM no longer represent the entire rest of a computation relative to a redex but only the rest of the computation up to the closest **handle** expression. A representation of the entire rest of the computation must include the pending **handle** expressions as well. We call this extension of the set of evaluation contexts the set of *standard contexts*.

Definition 5.2.4 (*Standard Contexts and Redexes*) The set of standard contexts for Error ISWIM is defined by the following grammar (for $1 \le i \le m$):

$$\begin{array}{rcl} S & ::= & [\ ] \mid (SM) \mid (VS) \mid (o^m\ V_1\ \ldots V_{i-1}\ S\ M_{i+1}\ \ldots\ M_m) \\ & & \mid (\textsf{handle}\ S\ \textsf{with}\ V) \end{array}$$

where M is arbitrary, $V, V_1, \ldots, V_n$ are values, and o^m is a primitive operator. ∎

Based on the notion of a standard context, the new standard reduction algorithm could proceed almost like the one for ISWIM and Error ISWIM:

1. Unless the program is already a value or an **error** expression, partitioning a program into a standard context S and an **h**-redex R.

2. Reduce R to C and construct the program $S[R]$.

3. Start over.

Put more formally,

$$M \longmapsto_h N \text{ iff for some } S, M \equiv S[M'], N \equiv S[N'], \text{ and } (M', N') \in \mathbf{h}.$$

Unfortunately, this definition does not yield a function. For example, if E and E' are evaluation contexts and S is a standard context, then

$$S[E[E'[\mathsf{error}\ b]]] \longmapsto_h S[E[\mathsf{error}\ b]]$$

as well as

$$S[E[E'[\mathsf{error}\ b]]] \longmapsto_h S[\mathsf{error}\ b].$$

Indeed, the definition of a standard reduction function for Error ISWIM involved the same problem, but we solved it without much ado by demanding that the standard reduction function eliminate the entire program if the standard redex is error. For Handler ISWIM this simplistic solution does not work. Instead of eliminating all of the standard context, an error expression only erases as much of its surrounding evaluation context as possible without deleting a handle expression. Thus, if the entire standard context is an evaluation context, an error expression terminates the evaluation; otherwise it only removes the evaluation context in the body of the closest handle expression:

$$\begin{array}{rcl} E[\mathsf{error}\ b] & \longmapsto_h & (\mathsf{error}\ b) \\ S[(\mathsf{handle}\ E[\mathsf{error}\ b]\ \mathsf{with}\ V)] & \longmapsto_h & S[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ V)] \end{array}$$

The complete definition follows.

Definition 5.2.5 (*Standard Reduction Function*) The set of Handler ISWIM *standard redexes* is defined as

$$\begin{array}{rcl} R & ::= & (V_1\ V_2) \mid (o^n\ V_1\ \ldots\ V_n) \mid (\mathsf{error}\ b) \\ & & \mid (\mathsf{handle}\ V\ \mathsf{with}\ \lambda x.M) \mid (\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ \lambda x.M) \end{array}$$

Given a δ-function f, the *standard reduction function* maps closed non-values to expressions:

$$\begin{array}{rcl} S[(o^m\ b_1 \ldots b_m)] & \longmapsto_h & S[V] \quad \text{if } f^m(o^m, b_1, \ldots, b_m) = V \\ S[((\lambda x.M)\ V)] & \longmapsto_h & S[M[x \leftarrow V]] \\ E[\mathsf{error}\ b] & \longmapsto_h & \mathsf{error}\ b \\ S[M] & \longmapsto_h & S[\mathsf{error}\ b] \quad \text{if } M \delta_{error} \mathsf{error}\ b \\ S[(\mathsf{handle}\ U\ \mathsf{with}\ V)] & \longmapsto_h & S[U] \\ S[(\mathsf{handle}\ E[\mathsf{error}\ b]\ \mathsf{with}\ V)] & \longmapsto_h & S[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ V)] \\ S[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ V)] & \longmapsto_h & S[(V\ b)] \end{array}$$

The partial function $eval_h^s : \Lambda_e^0 \longrightarrow A$ is defined as follows:

$$eval_e^s(M) = \begin{cases} b & \text{if } M \longmapsto_e^* b \\ \texttt{closure} & \text{if } M \longmapsto_e^* \lambda x.N \\ \texttt{error}\ b & \text{if } M \longmapsto_e^* (\mathsf{error}\ b) \end{cases}$$

∎

The correctness proof of this new specification of the evaluator consists of the usual steps. First, we need to show that it is a partial function. This follows from the usual "Unique Partitioning Lemma".

Lemma 5.2.6 (Unique Standard Contexts) *Every closed Handler ISWIM expression M is either a value, or there exists a unique standard context S and a standard redex R such that $M = S[R]$.*

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 5.2.7 $eval_h = eval_h^s$

is it interesting? **Proof.** ∎

5.2.3 Observational Equivalence of Handler ISWIM

Observational equivalence of Handler ISWIM is defined as usual.

Definition 5.2.8 (*Observational Equivalence for Handler ISWIM*) Two Handler ISWIM expressions, M and M', are *observationally equivalent*, written: $M \simeq_h M'$, if and only if

$$eval_h(C[M]) = eval_h(C[M'])$$

for all contexts C such that $C[M]$ and $C[M']$ are programs. ∎

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 5.2.9 *For $M, N \in \Lambda$, $M \simeq_e N$ implies $M \simeq_h N$. Moreover, Theorem 5.1.16 does not hold for Handler ISWIM.*

Proof. The first part is trivial. For the second part, assume that C is a distinguishing context in Error ISWIM for M and N. Take

$$C' = (\mathsf{handle}\ C\ \mathsf{with}\ (\lambda x.x))$$

as the distinguishing context in Handler ISWIM. ∎

Handlers are useless unless an error occurs. Some use of the activity lemma can probably do this!

Conjecture I: For $M, N \in \Lambda_e$, $M \simeq_e N$ iff $M \simeq_h N$.

Conjecture II: Let

$$\begin{array}{rcl} J_e & = & \{(M,N) \mid M,N \in \Lambda, M \simeq_v N, M \not\simeq_e N\} \\ J_h & = & \{(M,N) \mid M,N \in \Lambda, M \simeq_v N, M \not\simeq_h N\} \end{array}$$

Then, $J_e = J_h$. Note: Conjecture I implies Conjecture II.

Exercise 5.2.1 Here is an alternative evaluator of Handler ISWIM:

$$eval'_h(M) = \begin{cases} b & \text{if } M =_h b \\ \mathtt{closure} & \text{if } M =_h \lambda x.N \\ b & \text{if } M =_h \mathsf{error}\ b \end{cases}$$

It differs from the original evaluator in that it maps all programs to basic constants or `closure`. In particular, the modified evaluator ignores the difference between errors and basic constants.

Check that the modified evaluator is still a function. Show that the two observational equivalence relations, defined by $eval_h$ and $eval'_h$, respectively, differ. ∎

5.3 Tagged Handlers

The use of error handlers for signaling the end of a sub-computation and circumventing the ordinary flow of evaluation raises the question whether this programming methodology is safe. Since error handlers accept all kind of errors and pass the associated information to their handler, it may happen that an error element intended for one handler is intercepted by a different handler. And indeed, this is easily possible in Handler ISWIM.

Suppose a procedure f accepts another procedure g as an argument. Furthermore, assume that g's implementor uses $\mathsf{error}\ 0$ to exit whenever the result is known but an evaluation according to ordinary rules would be much slower. A typical call pattern may be:

$$(\mathsf{handle}\ (f\ (\lambda x. \cdots \mathsf{error}\ 0 \cdots))\ \mathsf{with}\ (\lambda x.x)).$$

But the implementor of f, perhaps a different person than the implementor of g, may want to protect the algorithm in f against errors that can occur in sub-computations:

$$f = \lambda g. \cdots (\mathsf{handle}\ (/\ \ulcorner 1 \urcorner\ (g\ x))\ \mathsf{with}\ (\lambda x.\ulcorner 10000000000 \urcorner)).$$

Now if the application of g to x signals an error to speed up the execution of the program, the error will not reach the error handler at the application site of f but only the error handler at the application site of g.

A first step towards eliminating this interference problem is the separation of the information that reports what error occurred and the information that connects an error with an error handler. In other words, a handler expression requires the specification of an additional value, called a *label*, and each error element is associated with one of these labels. For the moment, we assume that such a label is an ordinary ISWIM value. If the label on an error and on an error handler are equal, the two are associated, *i.e.*, if this error is the result of the protected body, then the error handler will be applied to the basic constant associated with the error.

Since error elements now carry two pieces of information, a label and a basic constant, and since the label is used first, we switch to the following syntax for errors:

$$(\mathsf{error}\ L\ b)$$

where L is a value, which we call a label. The construct for installing a handler also requires a small modification:

$$(\mathsf{handle}\ L:\ M\ \mathsf{with}\ \lambda x.N).$$

The value L is also called a *label*, M is the body, and $\lambda x.N$ is the error handler.

Adapting the basic notions of reduction to this modified syntax and the behavior is easy. The relation **return** stays the same.

$$(\mathsf{handle}\ L:\ V\ \mathsf{with}\ \lambda x.N) \longrightarrow V.$$

When the body of a handler expression does not raise any error signals, there is no need to handle an error. However, the relation for modeling error handling changes. Before an error handler is applied to the basic constant that comes with an error, the labels on the error and the error handler must be compared. We emphasize this division of concerns by separating the notion of reduction for handling errors into two pieces:

$$\begin{array}{lcll}(\mathsf{handle}\ L:\ (\mathsf{error}\ L'\ b)\ \mathsf{with}\ \lambda x.N) & \longrightarrow & ((\lambda x.N)\ b) & \text{if } L = L' \\ (\mathsf{handle}\ L:\ (\mathsf{error}\ L'\ b)\ \mathsf{with}\ \lambda x.N) & \longrightarrow & (\mathsf{error}\ L'\ b) & \text{if } L \neq L'\end{array}$$

The formulation of these relations looks simple but appearances are deceiving. Both relations rely on a comparison of arbitrary values, which is clearly a problem in the presence of procedures as values. There are two obvious, vastly different methods for comparing procedures. The first method would check whether they are the same mathematical function. Due to the infinite domains and ranges of these functions, this check is clearly not computable, which means that the semantics cannot be implemented. The second method would check whether the procedures are textually identical. Even though this approach could in principle be implemented, it is expensive to traverse a large piece of program text, or a closure of an appropriately modified CEK-machine, just to determine whether an error handler should be invoked or not.

A simple solution of the comparison problem is to restrict the comparison of labels to the comparison of basic constants and to ignore procedures as labels. That is, we demand that error handlers are used only if the labels are both basic constants and identical. The introduction of this restriction is a typical example for the effects of translating informal language definitions into formal specifications. The translation reveals vagueness and forces clarifications. The following definition summarizes the modified language developed so far; the definition is sketchy because the language is not quite the desired one and because the final language specification is beyond the scope of the current chapter.

Definition 5.3.1 (*TH ISWIM: Syntax and Calculus*) The sets of expression, values, and evaluation contexts for TH ISWIM are:

$$
\begin{array}{lcl}
M & ::= & x \mid (\lambda x.M) \mid (M\ M) \\
 & & \mid b \mid (o^n\ M \ldots M) \\
 & & \mid (\mathsf{error}\ V\ b) \\
 & & \mid (\mathsf{handle}\ V:\ M\ \mathsf{with}\ \lambda x.M) \\
 & & \\
V & ::= & x \mid (\lambda x.M) \mid b \\
 & & \\
E & ::= & [\ ] \mid (V\ E) \mid (E\ M) \mid (o\, V \ldots V\ E\ M \ldots M).
\end{array}
$$

The set of contexts is similar to the set of Handler ISWIM contexts.

In addition to δ, β_v, δ_{error}, and **error**, the calculus for TH ISWIM is based on the following two notions of reduction:

return: $(\mathsf{handle}\ L:\ V\ \mathsf{with}\ \lambda x.N)$ **return** V;

handle: $(\mathsf{handle}\ c:\ (\mathsf{error}\ c'\ b)\ \mathsf{with}\ \lambda x.N)$ **handle** $((\lambda x.N)\ b)$ if $c = c'$ for $c, c' \in$ *BConsts*;

propagate: $(\mathsf{handle}\ L:\ (\mathsf{error}\ L'\ b)\ \mathsf{with}\ \lambda x.N)$ **propagate** $(\mathsf{error}\ L'\ b)$ if $L \notin$ *BConsts*, $L' \notin$ *BConsts*, or $L \neq L'$.

∎

If the underlying language of primitive data contains numerals and lists, the above example of a function that multiplies the elements of a list of numbers has now the following shape:

$$\Pi^0 = \lambda l.(\mathsf{handle}\ \ulcorner 13 \urcorner:\ (\Pi'\ l)\ \mathsf{with}\ \lambda x.x)$$

where

$$\Pi' = \mathsf{Y}_v(\lambda\pi.\ \lambda l.\ \begin{array}{lll}(\mathsf{if0} & (\mathsf{null?}\ l) & \ulcorner 1 \urcorner \\ (\mathsf{if0} & (\mathsf{car}\ l) & (\mathsf{error}\ \ulcorner 13 \urcorner\ \ulcorner 0 \urcorner) \\ & (*\ (\mathsf{car}\ l)\ (\pi\ (\mathsf{cdr}\ l))))) & \end{array}$$

The numeral $\ulcorner 13 \urcorner$ serves as the label for the exit point. The equality predicate in the tagged handler is numerical equality.

Exercise 5.3.1 The evaluator for the extension of Error ISWIM with tagged handlers is defined in the usual manner:

$$eval_{th}(M) = \begin{cases} b & \text{if } M =_{th} b \\ \mathtt{closure} & \text{if } M =_{th} \lambda x.N \\ \mathsf{error}_b & \text{if } M =_{th} \mathsf{error}_b \end{cases}$$

where $=_{th}$ is the calculus for tagged error handlers. Prove that $eval_{th}$ is a function. ∎

too difficult! **Exercise 5.3.2** The addition of tagged error handlers to Handler ISWIM is superfluous if the set of errors is finite and if a general equality predicate for basic constants exists. Assume that $Errors = \{b, b_1, \ldots, b_e\}$. Show that replacing every instance of

$$(\mathsf{handle}\ b:\ M\ \mathsf{with}\ N)$$

with

$$\begin{array}{ll}\mathsf{handle}\ M\ \mathsf{with} & ((\lambda h.\lambda x. \\ & \quad (\mathsf{if0}\ \ (=\ b\ x)\ (h\ b) \\ & \quad (\mathsf{if0}\ \ (=\ b_1\ x)\ \mathsf{error}_{b_1} \\ & \quad \ldots \\ & \quad (\mathsf{if0}\ \ (=\ b_n\ x)\ \mathsf{error}_{b_e})\ldots)))) \\ & N)\end{array}$$

in a prorgam does not affect the final result of the program. ∎

Unfortunately, tagged handlers only reduce the risk of interference but do not eliminate it. Different programmers may still accidentally pick the same label for two different error handlers, which may result in one catching errors intended for the other. What Handler ISWIM needs in addition to tagged handlers is a facility to define *unique* values. Since the problem of creating unique values is closely related to the problem of allocating reference cells with an indefinite life-time, we defer the problem of defining an interference-free error handler until we have designed a calculus of reference cells in the next chapter.

Exercise 5.3.3 Catch and Throw Once an error construct contains two pieces of information, it is really a control construct for circumventing the conventional flow of control. The developers of Lisp realized this confusion between errors and control constructs at an early stage and introduced the control constructs catch and throw. For circumventing the pending computations a program *throws* a value to a destination with a certain *label*. Dually, a program can *catch* the *thrown* value of a sub-expression if the labels agree. The concrete syntax for the catching construct is

$$(\mathsf{catch}\ L\ M)$$

where L is a label value and M, which is the body of the construct; for the former, we use

$$(\mathsf{throw}\ L\ M)$$

where L also evaluates to a label and M evaluates to the value to be thrown.

Define a *Catch* and *Throw* calculus for each of the following choices:

1. The label sub-expressions of catch and throw are values.

2. The label sub-expressions are arbitrary expressions.

In each case, the comparison of labels may ignore variables and procedures. Find examples that distinguish the two choices.

The intention is that a throw expression, whose sub-expressions are values, eliminates pending computations; a catch expression whose body is a throw expressions may stop the throw. ∎

5.4 Control Machines

As we could see in Chapter 2, 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. 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.

5.4.1 The Extended CC Machine

Recall the seven basic standard transitions:

$$\begin{array}{rcl}
S[(o^m\ b_1 \ldots b_m)] & \longmapsto_h & S[V] \quad \text{if } f^m(o^m, b_1, \ldots, b_m) = V \\
S[((\lambda x.M)\ V)] & \longmapsto_h & S[M[x \leftarrow V]] \\
E[\mathsf{error}\ b] & \longmapsto_h & \mathsf{error}\ b \\
S[M] & \longmapsto_h & S[\mathsf{error}\ b] \quad \text{if } M \delta_{error} \mathsf{error}\ b \\
S[(\mathsf{handle}\ U\ \mathsf{with}\ V)] & \longmapsto_h & S[U] \\
S[(\mathsf{handle}\ E[\mathsf{error}\ b]\ \mathsf{with}\ V)] & \longmapsto_h & S[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ V)] \\
S[(\mathsf{handle}\ (\mathsf{error}\ b)\ \mathsf{with}\ V)] & \longmapsto_h & S[(V\ b)]
\end{array}$$

The basic difference between this standard reduction function and the one for ISWIM is that the latter distinguishes two notions of the rest of a computation. In Handler ISWIM an evaluation context represents the rest of the computation up to the closest handle expression and a standard redex denotes the rest of the entire computation relative to some standard redex.

A naïve adaptation of the strategy of Chapter 3 suggests a separation of the standard context from the control string to avoid the repeated partitioning task. Machine states for the revised CC-machine are 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_{error}-redex, the machine places an appropriate error expression in its control register. Finally, if the control string is an error 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 definition extends Definition 3.2.1.

Definition 5.4.1 (*CC machine for Handler ISWIM*)
State space:

$$\Lambda^0 \times SConts$$

State transitions:

Redex		*Contractum*	*cc*
$\langle M, S[\;]\rangle$	$\longmapsto$	$\langle \mathsf{error}\ b, S[\;]\rangle$ provided $M\,\delta_{error}\,(\mathsf{error}\ b)$	**h**1
$\langle (\mathsf{error}\ b), E[\;]\rangle$	$\longmapsto$	$\langle (\mathsf{error}\ b), [\;]\rangle$	**h**2
$\langle (\mathsf{handle}\ M\ \mathsf{with}\ \lambda x.N), S[\;]\rangle$	$\longmapsto$	$\langle M, S[(\mathsf{handle}\ [\;]\ \mathsf{with}\ \lambda x.N)]\rangle$	**h**3
$\langle V, S[(\mathsf{handle}\ [\;]\ \mathsf{with}\ U)]\rangle$	$\longmapsto$	$\langle V, S[\;]\rangle$	**h**4
$\langle (\mathsf{error}\ b), S[(\mathsf{handle}\ E\ \mathsf{with}\ V)]\rangle$	$\longmapsto$	$\langle (\mathsf{error}\ b), S[(\mathsf{handle}\ [\;]\ \mathsf{with}\ V)]\rangle$	**h**5
$\langle (\mathsf{error}\ b), S[(\mathsf{handle}\ [\;]\ \mathsf{with}\ V)]\rangle$	$\longmapsto$	$\langle (V\ b), S[\;]\rangle$	**h**6

The evaluation function:

$$eval_h^{cc}(M) = \begin{cases} b & \text{if } \langle M, [\;]\rangle \longmapsto_{cc}^{*} \langle b, [\;]\rangle \\ \mathtt{closure} & \text{if } \langle M, [\;]\rangle \longmapsto_{cc}^{*} \langle \lambda x.N, [\;]\rangle \\ \mathtt{error}\ b & \text{if } \langle M, [\;]\rangle \longmapsto_{cc}^{*} \langle \mathsf{error}\ b, [\;]\rangle \end{cases}$$

∎

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 5.4.2 $eval_h^s = eval_h^{cc}$

Exercise 5.4.1 Prove Theorem 5.4.2. ∎

Exercise 5.4.2 Prove that it is possible to eliminate $\mathbf{h}_2$ by wraping the initial program in a handler ∎ error x necessary!

5.4.2 The CCH Machine

One major bottleneck of the naïve CC-machine for Handler ISWIM is obvious. Even though 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 teh 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 handle expressions. In order to handle errors correctly, the evaluation contexts should be paired with their respective handlers. When the machine encounters a handle expression, it pushes the current evaluation context and the new handler on the stack:

$$\langle(\mathsf{handle}\ M\ \mathsf{with}\ \lambda x.N), E, H\rangle \longmapsto \langle M, [\ ], \langle\lambda x.N, E\rangle H\rangle.$$

When the body of a handle expression is eventually reduced to a value, the machine pops the stack

$$\langle V, [\ ], \langle V, E\rangle H\rangle \longmapsto \langle V, E, H\rangle;$$

that is, it throws away the current handler and reinstalls the surrounding evaluation context. The instruction for handling errors is analoguous.

The CC instructions for plain ISWIM expressions basically stay the same. They do not use the additional register in any way and do not alter it. We call the new machine the CCH machine, where the H stands for handler stack.

Definition 5.4.3 (*CCH machine for Handler ISWIM*)
State space:

$$\Lambda^0 \times \mathit{EConts} \times (\mathit{Vals}^0 \times \mathit{EConts})^*$$

State transitions:

Redex		*Contractum*	*cch*
$\langle (MN), E[\ ], H\rangle$ if $M \notin \mathit{Vals}$	$\longmapsto$	$\langle M, E[([\ ]\ N)], H\rangle$	1
$\langle (VM), E[\ ], H\rangle$ if $M \notin \mathit{Vals}$	$\longmapsto$	$\langle M, E[(V\ [\ ])], H\rangle$	2
$\langle (o\ V_1 \ldots V_i\ M\ N \ldots), E[\ ], H\rangle$ if $M \notin \mathit{Vals}$, for all $i \geq 0$	$\longmapsto$	$\langle M, E[(o\ V_1 \ldots V_i\ [\ ]\ N \ldots)], H\rangle$	3
$\langle ((\lambda x.M)\ V), E[\ ], H\rangle$	$\longmapsto$	$\langle M[x \leftarrow V], E[\ ], H\rangle$	β_v
$\langle (o\ b_1 \ldots b_m), E[\ ], H\rangle$	$\longmapsto$	$\langle V, E[\ ], H\rangle$ where $\delta(o, b_1, \ldots, b_m) = V$	δ_v
$\langle V, E[(U\ [\ ])], H\rangle$	$\longmapsto$	$\langle (U\ V), E[\ ], H\rangle$	4
$\langle V, E[([\ ]\ N)], H\rangle$	$\longmapsto$	$\langle (V\ N), E[\ ], H\rangle$	5
$\langle V, E[(o\ V_1 \ldots V_i\ [\ ]\ N \ldots)], H\rangle$	$\longmapsto$	$\langle (o\ V_1 \ldots V_i\ V\ N \ldots), E[\ ], H\rangle$	6
$\langle M, E, H\rangle$ if $M \delta_{error} (\mathsf{error}\ b)$	$\longmapsto$	$\langle \mathsf{error}\ b, E, H\rangle$	**h**1
$\langle (\mathsf{error}\ b), E, H\rangle$	$\longmapsto$	$\langle (\mathsf{error}\ b), [\ ], H\rangle$	**h**2
$\langle (\mathsf{handle}\ M\ \mathsf{with}\ \lambda x.N), E, H\rangle$	$\longmapsto$	$\langle M, [\ ], \langle \lambda x.N, E\rangle H\rangle$	**h**3
$\langle V, [\ ], \langle U, E\rangle H]\rangle$	$\longmapsto$	$\langle V, E, H\rangle$	**h**4
$\langle (\mathsf{error}\ b), E, H\rangle$	$\longmapsto$	$\langle (\mathsf{error}\ b), [\ ], H\rangle$	**h**5
$\langle (\mathsf{error}\ b), [\ ], \langle U, E\rangle H\rangle$	$\longmapsto$	$\langle (U\ b), E, H\rangle$	**h**6

The evaluation function:

$$eval_h^{cch}(M) = \begin{cases} b & \text{if } \langle M, [\ ]\rangle \longmapsto^*_{cch} \langle b, [\ ]\rangle \\ \texttt{closure} & \text{if } \langle M, [\ ]\rangle \longmapsto^*_{cch} \langle \lambda x.N, [\ ]\rangle \\ \texttt{error}\ b & \text{if } \langle M, [\ ]\rangle \longmapsto^*_{cch} \langle \mathsf{error}\ b, [\ ]\rangle \end{cases}$$

∎

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 5.4.4 $eval_h^{cch} = eval_h^{cc}$

Exercise 5.4.3 Prove Theorem 5.4.4. ∎

Exercise 5.4.4 Implement the extended CC and CCH machines. Measure the improvement. ∎

Exercise 5.4.5 Derive extended CK and CEK machines based on the CCH machine. ∎

Exercise 5.4.6 Design a modified CCH machine that reserves another register for the current error handler. Prove its equivalence with the above CCH machine. ∎

History

pragmatic development

calculus of control constructs: Felleisen & Friedman, ... & Hieb ... & Wright [1986a, 1986b, 1987, 1992, 1995]; Cartwright & Felleisen [1992]

simple versions: known but not published

machines: Talcott, Felleisen & Friedman

proof techniques: Barendregt Chapter 3.2 and 3.3, especially the commuting CR problem. See historical references there.

errors and errset: Lisp 1.5, but also see ML and other languages for error and exception handlers.

catch & throw: Lisp

Chapter 6

Imperative Assignment

Most programming languages include assignment statements for changing the value of identifiers or operators for mutating the state of data structures. Indeed, the core of early high-level programming languages was a set of assignment facilities plus some primitive functional capabilities like **while**- and **for**-loops. Adding assignments or mutation operators to languages with higher-order procedures and complex data structures provides a whole new set of interesting programming paradigms. In such a language, programs can create, manipulate, and alter cyclic data structures and recursive nests of procedures. Similarly, they compose an opaque bundle consisting of data structures and procedures for changing the data structures such that other parts of the program can only inspect or change the state of this bundle according to some strictly enforced, yet programmer-specified protocol. This paradigm is the basis of object-oriented programming.

unique objects exception handling a la ML

In the first section, we treat assignable variables as they are found in Scheme [*ref:* rrrs]. In the second section, we collect garbage. In the third section, we look at Scheme- and ML-style reference cells and briefly address objects. In the last section, we discuss parameter-passing: assignments add an extra dimension to this aspect of a language.

6.1 Assignable Variables

When a language provides assignment statements for identifiers, the nature of identifier no longer resembles that of mathematical variables. While variables always represent some fixed value, 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*.[1] The rest of this section introduces a syntax for *State* ISWIM, a language with assignment statements for ordinary program identifiers. The language also includes syntax to express the association between identifiers and values explicitly. Afterwards we gradually introduce the semantics of

[1]Reflects the major use: to model the changing state of an object in the real world.

the language, define a calculus for *State* ISWIM, and analyze its relationship to the λ_v-calculus.

6.1.1 Syntax

To add assignable identifiers and assignments to ISWIM, we extend the syntax with two classes of expressions. The first one is the assignment statement

$$(:= x\ M),$$

where the variable x is called the *assigned* identifier and M is called the right-hand side. An assignment evaluates the right-hand side and changes the value that the left-hand side identifier represents: from now on until the next assignment or until the end of the program evaluation, the identifier represents the new value. Any reference to this identifier where the program expects a value will produce this new value. The value of the right-hand side of an assignment is also the result of the entire assignment.

The purpose of the second new facility is to represent a set of associations between identifiers and values. These expressions are similar to **begin**-blocks in Algol 60 and **letrec**-expressions in Scheme. We call them ρ-expressions [*ref:* TCSr]. A ρ-expression has the shape

$$\rho\{(x_1, V_1); \ldots; (x_n, V_n)\}.M$$

where $x_1, \ldots, x_n$ are variables, $V_1, \ldots, V_n$ are values, and M is an arbitrary expression. It binds the variables $x_1, \ldots, x_n$ in both M and $V_1, \ldots, V_n$, that is, the bindings are mutually recursive.

A ρ-expression provides a tool for representing in a syntactic manner what value some given identifier stands for in some expression. Formally, a ρ-expression is most easily explained as an abbreviation of an ISWIM expression with assignments. Specifically, it is equivalent to the application of a procedure with parameters x_1 through x_n to some arbitrary initial values (I). The procedure first assigns values V_i to the identifiers x_i, respectively, and then evaluates M:

$$((\lambda x_1, \ldots, x_n.((\lambda d_1, \ldots, d_n, d.d)\ (:= x_1\ V_1) \ldots (:= x_n\ V_n)\ M))I \ldots I).$$

We add ρ-expressions for convenience but also to express the fact that in languages with assignments (and assignable operators) the association of identifiers to values constitutes a primitive action that deserves a syntactic counterpart.

The extension of ISWIM with assignments and ρ-expressions is called *State ISWIM*.

Definition 6.1.1 (*State ISWIM*) Let $:=$, ρ, $\{$, and $\}$, be new terminal symbols. Then, *State* ISWIM is the following extension of ISWIM:

$$\begin{aligned} M &::= V \mid x \mid (M\ M) \mid (:=\ x\ M) \mid \rho\{\theta\}.M \\ V &::= b \mid f \mid \lambda x.M \\ \theta &::= \epsilon \mid \theta(x, V) \end{aligned}$$

where $x \in Vars$, $b \in BConsts$ and $f \in FConsts$ as usual.

In an assignment, $(:=\ x\ M)$, x is the left-hand side and M is the right-hand side.

In a ρ-expression, $\rho\theta.M$, the list of identifier-value pairs, θ, is the *declaration*, M is the body. The list of identifiers $x_1, \ldots, x_n$ in a declaration $\theta = (x_1, V_1) \ldots (x_n, V_n)$ are called the domain of the declaration, the values $V_1, \ldots, V_n$ are its range.

For a ρ-expression to be a legal syntactic expression, a variable may occur at most once in its domain. ∎

Free and Bound Variables, ρ-Sets, Assigned Variables: Since *State* ISWIM introduces a new binding construct, ρ, we need to reconsider the conventions about free and bound variables. First, we extend Barendregt's conventions for free and bound variables. That is, we identify all ρ-expressions that differ only through a systematic renaming of the bound variables:

$$\begin{aligned}
&\rho\{(x_1, V_1) \ldots (x_n, V_n)\}.M \\
&\quad \equiv \rho\{(y_1, V_1[x_1 \leftarrow y_1] \ldots [x_n \leftarrow y_n]) \ldots (y_n, V_n[x_1 \leftarrow y_1] \ldots [x_n \leftarrow y_n])\}. \\
&\qquad\qquad M[x_1 \leftarrow y_1] \ldots [x_n \leftarrow y_n]
\end{aligned}$$

where $y_1, \ldots, y_n$ do not occur in $V_1, \ldots, V_n$ and M.

Second, we also identify all ρ-expressions that differ only by the ordering of their identifier-value pairs:

$$\begin{aligned}
\rho\{(x_1, V_1) \ldots (x_n, V_n)\}.M &\equiv \rho\{(x_{i_1}, V_{i_1}) \ldots (x_{i_n}, V_{i_n})\}.M, \\
&\quad \text{for all permutations } i_1, \ldots, i_n \text{ of } 1, \ldots, n.
\end{aligned}$$

Put differently, we treat ρ-declarations as *sets* of pairs and sometimes even as *finite functions*. Thus, when we write

$$\rho\theta \cup \{(x, V)\}.M$$

we imply that $\theta \cup \{(x, V)\}$ is a valid set of declarations, i.e., that x does not occur in the domain of θ.

Taking into account the above conventions, *State* ISWIM's syntax can be specified in an alternative way using extended BNF notation:

$$\begin{aligned}
M &::= x \mid V \mid (M\ M) \mid \rho\{(x_1, V_1) \ldots (x_n, V_n)\}.M \\
V &::= b \mid f \mid \lambda x.M.
\end{aligned}$$

We use set notation for declarations to emphasize that they are basically finite functions.

The set of *assigned variables* in an expression M, denoted $AV(M)$, is the set of *free* variables in M that occur on the left-hand side of an assignment.

6.1.2 Reductions for *Tiny*

Clearly, a calculus for *State* ISWIM cannot be a simple extension of the λ_v-calculus, the calculus for ISWIM. In particular, β_v, the central axiom of the ISWIM calculus, requires the substitution of a procedure parameter by the argument value. But a naïve usage of this rule for *State* ISWIM would result in *syntactically* incorrect programs because left-hand sides of assignments would be replaced by values. Indeed, while it is almost obvious how assignments work on concrete computers, it is not at all clear how reductions, that is, changes of program text, can explain the effect of assignments at all.

To develop some intuition into the problem, we temporarily ignore λ-abstractions and concentrate on a sub-language, *Tiny*, of *State* ISWIM, which is a simple, imperative language based on assignment, blocks, and statement sequences. In addition, the body of an *Tiny* ρ-expression may consist of a sequence of expressions. An evaluation of such a sequence proceeds from left to right, discarding all values except for the last one, which is the result of the entire ρ-expression. Finally, for the following examples, the language contains some arithmetic sub-language.

A typical expression in *Tiny* would be

$$\begin{array}{rl} M_1 = & \rho\{(x, \ulcorner 1\urcorner)\ (y, \ulcorner 2\urcorner)\ (z, \ulcorner 3\urcorner)\}. \\ & \quad (:=\ x\ \ulcorner 4\urcorner) \\ & \quad (:=\ y\ x) \\ & \quad (:=\ x\ (+\ z\ y))\ . \end{array}$$

The first assignment in the body is supposed to change the association between x and $\ulcorner 1\urcorner$ such that x stands for $\ulcorner 4\urcorner$. On an actual machine, such an assignment would alter the contents of a cell in the memory. In this example, we could consider the declarations of the ρ-expression as the relevant slice of the memory of such a machine. Then, one obvious way to model the assignment is to change the declaration and to replace the first assignment with its value, which in turn disappears according to the informal description above:

$$\begin{array}{rlcl} M_2 = & \rho\{(x, \ulcorner 4\urcorner)\ (y, \ulcorner 2\urcorner)\ (z, \ulcorner 3\urcorner)\}. & \mathit{red} & \rho\{(x, \ulcorner 4\urcorner)\ (y, \ulcorner 2\urcorner)\ (z, \ulcorner 3\urcorner)\}. \\ & \quad \ulcorner 4\urcorner & & \quad (:=\ y\ x) \\ & \quad (:=\ y\ x) & & \quad (:=\ x\ (+\ z\ y))\ . \\ & \quad (:=\ x\ (+\ z\ y))\ . & & \end{array}$$

The next assignment does not have a value on the right-hand side but the expression x. Hence, it is necessary to determine the value of x, which is currently $\ulcorner 4\urcorner$ as the declaration indicates. Since no other assignment can affect the association of x and $\ulcorner 4\urcorner$ before we need the value of x, it is safe to replace x with $\ulcorner 4\urcorner$ in this assignment:

$$\begin{array}{rl} M_3 = & \rho\{(x, \ulcorner 4\urcorner)\ (y, \ulcorner 2\urcorner)\ (z, \ulcorner 3\urcorner)\}. \\ & \quad (:=\ y\ \ulcorner 4\urcorner) \\ & \quad (:=\ x\ (+\ z\ y))\ . \end{array}$$

Now the first assignment expression in M_3 has a value on the right-hand side and we can proceed as for M_1:

$$\begin{array}{l}\rho\{(x, \ulcorner 4 \urcorner)\ (y, \ulcorner 4 \urcorner)\ (z, \ulcorner 3 \urcorner)\}. \\ \quad (:=\ x\ (+\ z\ y))\ .\end{array}$$

The final four steps are clear: to evaluate the expression $(+\ z\ y)$ we replace y with $\ulcorner 4 \urcorner$ and z with $\ulcorner 3 \urcorner$, add the two numerals, and reduce the assignment expression. The result is

$$\begin{array}{l}\rho\{(x, 7)\ (y, \ulcorner 4 \urcorner)\ (z, \ulcorner 3 \urcorner)\}. \\ \quad 7\ .\end{array}$$

Since the body of the block has been reduced to a simple value, we can say that the program evaluated to 7.

In summary, when assignment are in a "good" position with respect to the declaration of the assigned variable, and their right-hand side is a value, a reduction can take place that changes the declaration and eliminates the assignment expression. Similarly, when a ρ-declared identifier is in a "good" position, then a reduction can replace the identifier with its actual value. To formalize these first two notions of reduction, we must only fix what it means for a position to be "good."

In an ordinary machine, a "good position" for an assignment or an identifier lookup means that the respective expression is the next expression to be executed. Now recall that in our reduction-based "machine" for ISWIM the next expression to be reduced is in the hole of an evaluation context. Hence, the following definition of "good position" suggests itself naturally: An assignment or an identifier are in "good position" relative to some surrounding ρ-expression if the body is equivalent to an evaluation context with the assignment or identifier in its hole. Since a partitioning of expressions into evaluation contexts and subexpressions is unique, adopting this definition is feasible: at most one assignment in the body of a particular block can affect the declarations and at most one lookup can exploit the declaration, no imperative actions in a block can happen simultaneously.

Formally, the two reduction rules for Imperative ISWIM are:

$$\rho\theta \cup \{(x, V)\}.E[x] \ \ \mathit{red} \ \ \rho\theta \cup \{(x, V)\}.E[V] \qquad (D)$$

$$\rho\theta \cup \{(x, U)\}.E[(:= x\ V)] \ \ \mathit{red} \ \ \rho\theta \cup \{(x, V)\}.E[V] \qquad (\sigma)$$

where E ranges over evaluation contexts. For *Tiny*, evaluation contexts contain their hole in the first expression of the body of a ρ-expression. Moreover, since identifiers and even assignments can occur deeply nested in arithmetic expressions, evaluation contexts have approximately the same shape as in ISWIM. Their hole can occur in the function position of any application or in the argument position of applications whose function position is a value. Thus, for the moment, we can take the following set of evaluation contexts:

$$E \ ::= \ \rho\theta.E'\ M_2 \ldots M_m$$

$$E' \quad ::= \quad [\ ] \mid (:=\ x\ E') \mid (c\ E) \mid (E\ M).$$

To illustrate the generality of the definitions, we reduce the program

$$\rho\{(x, 0)\}.(\underline{(:=\ x\ 1^+)}\ (x\ \ulcorner 1 \urcorner))$$

to an answer:

$$\begin{array}{lll} \ldots & \mathit{red} & \rho\{(x, 1^+)\}.(1^+\ (\underline{x}\ \ulcorner 1 \urcorner)) \\ \ldots & \mathit{red} & \rho\{(x, 1^+)\}.(1^+\ (1^+\ \ulcorner 1 \urcorner)) \\ \ldots & \mathit{red} & \rho\{(x, 1^+)\}.(1^+\ \ulcorner 2 \urcorner) \\ \ldots & \mathit{red} & \rho\{(x, 1^+)\}.\ulcorner 3 \urcorner. \end{array}$$

The underlined sub-expressions above indicate the contents of the hole of the respective evaluation contexts.

While σ and D clearly form the core of a reduction system for *Tiny*, they do not suffice for the full sublanguage, which contains nested ρ-expressions. Consider the following example:

$$\begin{array}{l} \rho\{(x, \ulcorner 1 \urcorner)\ (y, \ulcorner 2 \urcorner)\ (z, \ulcorner 3 \urcorner)\}. \\ \quad (:=\ x\ \ulcorner 4 \urcorner) \\ \quad \rho\{(x, \ulcorner 3 \urcorner)\}. \\ \qquad (+\ (:=\ y\ x)\ y)\ . \end{array}$$

The first assignment is reducible according to σ and results in the expression

$$\begin{array}{l} \rho\{(x, \ulcorner 4 \urcorner)\ (y, \ulcorner 2 \urcorner)\ (z, \ulcorner 3 \urcorner)\}. \\ \quad \rho\{(x, \ulcorner 3 \urcorner)\}. \\ \qquad (+\ (:=\ y\ x)\ y)\ . \end{array}$$

For the second assignment to become a redex, x must become a value, which a D reduction accomplishes:

$$\begin{array}{l} \rho\{(x, \ulcorner 4 \urcorner)\ (y, \ulcorner 2 \urcorner)\ (z, \ulcorner 3 \urcorner)\}. \\ \quad \rho\{(x, \ulcorner 3 \urcorner)\}. \\ \qquad (+\ (:=\ y\ x)\ y)\ . \end{array}$$

The D reduction took the value from the *closest* ρ-declaration because this is where the identifier x in the original expression was bound. To finish the evaluation, the assignment to y must affect the declaration in the outermost ρ-expression, but, given the preliminary definition of evaluation contexts, the redex is not a σ-redex.

Two strategies suggest themselves naturally. First, we could extend the set of evaluation contexts such that the above expression contains a σ-redex. Second, we could add a notion of reduction that *merges* two ρ-expression when one is in a "good position" of the body of the other. In anticipation of the extension of Imperative ISWIM to *State* ISWIM, we choose the second solution and define two notions of reduction, $\rho_\cup$ and ρ_{lift}, pronounced ρ-merge and ρ-lift, respectively:

$$\rho\theta.\rho\theta'.M \ \ red \ \ \rho\theta\cup\theta'.M \qquad (\rho_\cup)$$

$$E[\rho\theta.M] \ \ red \ \ \rho\theta.E[M] \qquad (\rho_{lift})$$

Both relations rely on the naming conventions for variables. If E contains free variables from the domain of θ', they are automatically α-substituted appropriately. Moreover, if the domains of the two declarations overlap, the variable convention again demands that variables are renamed correctly.

To resume the reduction of our running example, we first rewrite the expression as

$$\begin{array}{l}\rho\{(x,\ulcorner 4\urcorner)\ (y,\ulcorner 2\urcorner)\ (z,\ulcorner 3\urcorner)\}.\\ \quad\rho\{(w,\ulcorner 3\urcorner)\}.\\ \qquad(:=\ y\ \ulcorner 3\urcorner)\ .\end{array}$$

Then, it reduces to

$$\begin{array}{l}\rho\{(x,\ulcorner 4\urcorner)\ (y,\ulcorner 2\urcorner)\ (z,\ulcorner 3\urcorner)\ (w,\ulcorner 3\urcorner)\}.\\ \quad(:=\ y\ \ulcorner 3\urcorner)\ .\end{array}$$

Now, the assignment, together with the evaluation context, forms a σ-redex and the body of the resulting expression is a value:

$$\begin{array}{l}\rho\{(x,\ulcorner 4\urcorner)\ (y,\ulcorner 3\urcorner)\ (z,\ulcorner 3\urcorner)\ (w,\ulcorner 3\urcorner)\}.\\ \quad\ulcorner 3\urcorner\ .\end{array}$$

As above, we say that the result of the program is $\ulcorner 3\urcorner$.

6.1.3 Reductions for *State* ISWIM

Equipped with reductions for the *Tiny*, we can return to the question of reductions for full *State* ISWIM. To do so, we must adapt and extend the calculus for *Tiny* in two regards. First, the above definitions rely on the definition of a set of evaluation contexts. Is it possible to extend this set and to adapt the notions of reductions *mutatis mutandis*? Second, full *State* ISWIM has procedures and requires a replacement for β_v, the central reduction rule for the functional core.

The adaptation of the set of evaluation contexts to the larger language is easy. Since λ-abstractions denote procedures that force the evaluation of their arguments, they deserve the same treatment as functional constants. Technically speaking, there must be evaluation contexts of the form

$$((\lambda x.M)\ E)$$

such that arguments to abstractions can be evaluated. Moreover, in the absence of multiple-bodied ρ-expressions it is unnecessary to a clause for specifying evaluation contexts in the single body of a ρ-expression. Thus, the set of evaluation contexts for *State* ISWIM is

$$E ::= [\,] \mid (V\ E) \mid (E\ M) \mid (:=\ x\ E).$$

Comparing the new definition of evaluation contexts to the one for pure ISWIM is illuminating. For the functional core language evaluation contexts only specify how to evaluate applications, which are the only tool to specify computations. *Tiny* adds assignment and hence a new syntactic form that requires a non-trivial evaluation rule. Given that assignments evaluate their right-hand side, it is natural to expect that the set of evaluation contexts contains additional contexts of the form

$$(:=\ x\ E).$$

However, *State* ISWIM also contains ρ-expressions, whose evaluation apparently requires membering the context. As indicated in the preceding subsection, our answer to this question is the ρ-lift rule, which moves ρ-expression out of the way when necessary.

To illustrate the ρ-lift rule and its effect, and to prepare the introduction of the final reduction rule, we discuss a series of examples. The first example shows how to
Machinesection think of ρ-expressions with λ-bodies as "closures" (see) that are about to be applied. Take the expression

$$((\rho\{(x, (\lambda y.x))\}.(\lambda z.x))\ \ulcorner 3 \urcorner).$$

By lifting the ρ-expression above the application, we get an ordinary application of a λ-expression to a value:

$$((\rho\{(x, (\lambda y.x))\}.(\lambda z.x)\ \ulcorner 3 \urcorner)).$$

Applying β_v yields

$$((\rho\{(x, (\lambda y.x))\}.x,$$

and the rest is an application of the dereference rule:

$$((\rho\{(x, (\lambda y.x))\}.(\lambda y.x).$$

The result is another "closure". The example shows how the ρ-declaration that surrounds the λ-expression may be perceived as a representation of its environment. To apply such an object it suffices to apply the body of the ρ-expression to the argument and to evaluate "under" the ρ-declaration.

To take this analogy further, we next consider an example that applies a procedure to a ρ-expression with a λ-body:

$$((\lambda x.x)\ (\rho\{(x, (\lambda y.x))\}.(\lambda z.x))).$$

Again, lifting the declaration over the application and β_v-reducing the application yields the result:

$$\begin{aligned} \ldots\ \ & \mathit{red} \quad (\rho\{(x, (\lambda y.x))\}.((\lambda x.x)\ (\lambda z.x))) \\ & \mathit{red} \quad (\rho\{(x, (\lambda y.x))\}.(\lambda z.x)) \end{aligned}$$

The outcome is a "closure" whose body refers to free variables in the original "closure".

Now we are finally ready to return to the question of how to adapt the reduction rules of ISWIM to a language with assignment statements. As we discussed above, the question arises because the presence of assignments to parameter identifiers implies a changing association between identifier and value. At this point the role of ρ-expressions should be clear. Instead of substituting a value for a parameter, the reduction of an application creates a syntactic association between the parameter and the value and then continues to evaluate the body of the λ-expression under this new declaration. This reasoning suggests the following rule:

$$((\lambda x.M)\ V)\ \ \mathit{red}\ \ \rho\{(x, V)\}.M\ . \tag{β_σ}$$

If the reduction of M leads to a basic constant, the answer is obvious. If it becomes an abstraction, the ρ-lift rule realizes the correct behavior of the combined object as we have seen in the preceding examples. In summary, δ, ρ_{lift}, $\rho_{\cup}$, D, and β_σ constitute a complete evaluation system for *State* ISWIM programs.

Example 1: Just to show how β_σ can do almost everything

$$\begin{array}{rl}
 & ((\lambda x.((\lambda d.(1^- x))\ (:=\ x\ (1^+ x))))\ulcorner 0\urcorner) \\
\mathit{red} & (\rho\{(x, \ulcorner 0\urcorner)\}.((\lambda d.(1^- x))\ (:=\ x\ (1^+ x)))) \\
\ldots\ \mathit{red} & (\rho\{(x, \ulcorner 0\urcorner)\}.((\lambda d.(1^- x))\ (:=\ x\ \ulcorner 1\urcorner))) \\
\mathit{red} & (\rho\{(x, \ulcorner 1\urcorner)\}.((\lambda d.(1^- x))\ \ulcorner 1\urcorner)) \\
\ldots\ \mathit{red} & (\rho\{(x, \ulcorner 1\urcorner)\ (d, \ulcorner 1\urcorner)\}.(1^- x)) \\
\ldots\ \mathit{red} & (\rho\{(x, \ulcorner 1\urcorner)\ (d, \ulcorner 1\urcorner)\}.\ulcorner 0\urcorner)
\end{array}$$

Ideally we would like to show eventually that this is the identity function on numbers. It makse sense to discard the surrounding rho.

Example 2: set!'s create recursive procedures

$$\begin{array}{rl}
 & ((\lambda f.((\lambda d.f)\ (:=\ f\ (\lambda y.fy))))(\lambda y.y)) \\
\mathit{red} & (\rho\{(f, (\lambda y.y))\}.((\lambda d.f)\ (:=\ f\ (\lambda y.fy)))) \\
\ldots\ \mathit{red} & (\rho\{(f, (\lambda y.fy))\}.((\lambda d.f)\ (\lambda y.fy))) \\
\mathit{red} & (\rho\{(f, (\lambda y.fy))\ (d, (\lambda y.fy)\}.f) \\
\mathit{red} & (\rho\{(f, (\lambda y.fy))\ (d, (\lambda y.fy)\}.(\lambda y.fy))
\end{array}$$

The result cannot survive w/o the rho.

Example 3: Recursion:

The final example of this subsection demonstrates how imperative higher-order programs reduce to values in this calculus and shows the need for recursive ρ-declarations. The original program is the expression:

$$\begin{array}{l}((\lambda f.\ ((\lambda d.(f\ \ulcorner 8 \urcorner)) \\ \qquad (:=\ f\ (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner)))))) \\ \ \ulcorner 0 \urcorner)\end{array}$$

$$\begin{array}{ll} & ((\lambda f.\ ((\lambda d.(f\ \ulcorner 8 \urcorner)) \\ & \qquad (:=\ f\ (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner)))))) \\ & \ \ulcorner 0 \urcorner) \\ \mathit{red} & (\rho\{(f, \ulcorner 0 \urcorner)\}. \\ & \qquad ((\lambda d.(f\ \ulcorner 8 \urcorner)) \\ & \qquad (:=\ f\ (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner)))))) \\ \mathit{red} & (\rho\{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))\}. \\ & \qquad ((\lambda d.(f\ \ulcorner 8 \urcorner)) \\ & \qquad (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))) \\ \mathit{red} & (\rho\{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))\}. \\ & \qquad (\rho\{(d, (\lambda x.(\mathsf{if0}\ x\ 0\ (f\ \ulcorner 0 \urcorner))))\}. \\ & \qquad\quad (f\ \ulcorner 8 \urcorner))) \\ \mathit{red} & (\rho\{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))(d, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))\}. \\ & \quad (f\ \ulcorner 8 \urcorner))\end{array}$$

The recursive function is set up. Now we can apply it

$$\begin{array}{ll}\mathit{red} & (\rho\{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))(d, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))\}. \\ & \quad ((\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner)))\ \ulcorner 8 \urcorner))\end{array}$$

The call to the recursive function sets up things in the env:

$$\begin{array}{ll}\mathit{red}^+ & (\rho\{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))(d, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))(x, \ulcorner 8 \urcorner)\}. \\ & \quad (\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))) \\ \mathit{red}^+ & (\rho\{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))(d, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner))))(x, \ulcorner 8 \urcorner)\}. \\ & \quad (f\ \ulcorner 0 \urcorner)) \\ \mathit{red}^+ & (\rho\{\ \{(f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner)))\ \}. \\ & \quad (d, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0 \urcorner)))) \\ & \quad (x, \ulcorner 8 \urcorner) \\ & \quad (x', \ulcorner 0 \urcorner) \\ & \quad (\mathsf{if0}\ x'\ x'\ (f\ \ulcorner 0 \urcorner)))\end{array}$$

$$
\begin{array}{ll}
\mathit{red}^{+} & (\rho\{\ (f, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0\urcorner))))\ \}. \\
 & \quad (d, (\lambda x.(\mathsf{if0}\ x\ x\ (f\ \ulcorner 0\urcorner)))) \\
 & \quad (x, \ulcorner 8\urcorner) \\
 & \quad (x', \ulcorner 0\urcorner) \\
 & \quad \ulcorner 0\urcorner)
\end{array}
$$

Summary: rho expressions are closures; lift and union rules plus beta-v show this for funcitonal programs. with beta-sigma, we get to do imperative things too. Next section this suffices!

6.1.4 Ooops

It turns out that replacing β_v with β_σ suffices to define a calculus for *Tiny* with the usual properties. That is, the calculus defines an evaluation function according to the usual definition and, according to this evaluator, a program either diverges or produces a value or reduces to a δ-stuck state (which corresponds to an error). δ-stuck state

define eval. prove uniform evaluation lemma?

Evaluation Contexts:

$$E ::= [\,] \mid (V\ E) \mid (E\ M) \mid (:=x\ E)$$

Notions of Reductions:

$$
\begin{array}{rcll}
fv & \mathit{red} & \delta(f,v) \text{ if } \delta(f,v) \text{ is defined} & (\delta) \\
((\lambda x.M)\ V) & \mathit{red} & \rho\{(x,V)\}.M & (\beta_\sigma) \\
\rho\theta \cup \{(x,V)\}.E[x] & \mathit{red} & \rho\theta \cup \{(x,V)\}.E[V] & (D) \\
\rho\theta \cup \{(x,U)\}.E[(:=x\ V)] & \mathit{red} & \rho\theta \cup \{(x,V)\}.E[V] & (\sigma) \\
\rho\theta.E[\rho\theta'.M] & \mathit{red} & \rho\theta \cup \theta'.E[M] & (\rho_\cup)
\end{array}
$$

Figure 6.1: Reductions for *State* ISWIM

The semantics of the call-by-value/pass-by-worth language is the partial function, $eval_v w$, from programs to answers, where an answer is either a value, or a ρ-expression whose body is a value. As discussed in the previous section, the program rewriting function first partitions a program into an evaluation context and a redex and then transforms it to a new program. A *call-by-value* evaluation context specifies the eager evaluation strategy:

$$E_v ::= [\,] \mid (v\ E_v) \mid (E_v\ e) \mid (:=x\ E_v)$$

The call-by-value/pass-by-worth language redexes are the following expressions: fv, $(\lambda x.e)v$, x, $(:=x\ v)$, and $\rho\theta.e$.

The program transformation function, *red*, and the semantics, $eval_v w$, for the call-by-value/pass-by-worth language, are defined in the second part of Figure 6.1. The first two rules, $E.\delta$ and $E.\beta_v$, provide the call-by-value semantics for Λ; the others specify the effects of imperative constructs.

6.1.5 Church-Rosser and Standardization

6.1.6 λ_v-S-calculus: Relating λ_v-calculus to *State* ISWIM

$$
\begin{array}{rcll}
(f\ v) & red & \delta(f,v) \text{ if } \delta(f,v) \text{ defined.} & (E.\delta) \\
(\lambda x.M)V & red & \rho\{(x,V)\}.M & (E.\beta_\sigma) \\
\rho\theta \cup \{(x,V)\}.E_v[x] & red & \rho\theta \cup \{(x,v)\}.E_v[V] & (E.D_v) \\
\rho\theta \cup \{(x,U)\}.E_v[(:=x\ V)] & red & \rho\theta \cup \{(x,V)\}.E_v[V] & (\sigma_v) \\
E_v[\rho\theta.M] & red & \rho\theta.E_v[M] & (\rho_{lift}) \\
\rho\theta\rho\theta'.M] & red & \rho\theta \cup \theta'.M & (\rho_\cup)
\end{array}
$$

ρ is Superfluous

6.2 Garbage

The machine rewrites the program until it produces an answer and then removes all unneeded ρ-bindings by applying "garbage collection" reductions to the answer. More technically, the garbage collection notion of reduction **gc** is the union of the following relations:

$$
\begin{array}{rcl}
\rho\theta_0 \cup \theta_1.e & red & \rho\theta_1.e \\
 & & \quad \text{if } \theta_0 \neq \emptyset \text{ and} \\
 & & \quad FV(\rho\theta_1.e) \cap \sqsupseteq (\theta_0) = \emptyset \\
\rho\emptyset.e & red & e
\end{array}
$$

It is easy to verify that the notion of reduction **gc** is strongly normalizing and Church-Rosser. Therefore, all expressions have a unique **gc**-*normal form* (**gc**-nf). Furthermore, the **gc**-nf of an answer is also an answer.

6.3 Boxes

6.3.1 Objects

6.4 Parameter Passing Techniques

6.4.1 Call-By-Value/Pass-By-Reference

6.4.2 Copy-In/Copy-Out

6.4.3 Call-By-name

6.4.4 Call-By-Need for Call-By-Name

can we prove equivalence of the two implementations?

Chapter 7

Complex Control

J, call/cc, we do: C, F
exercises: call/cc semantics

7.1 Continuations and Control Delimiter

7.2 Fancy Control Constructs

F; prompt with handlers; examples: programming rewriting systems

Chapter 8

Types and Type Soundess

types for simple functional language; typed programs don't go wrong
types for imperative language: typed programs still don't go wrong
poly types