Parameter-passing and the lambda calculus

Crank, Erik T., M.S.

Rice University, 1991

RICE UNIVERSITY

Parameter-Passing and the Lambda Calculus

by

Erik Crank

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

[illegible]

Matthias Fellesien, Director
Assistant Professor of Computer Science

Robert Cartwright

Robert Cartwright
Professor of Computer Science

[illegible]

Joe Warren
Assistant Professor of Computer Science

[illegible]

Bruce Duba
Research Scientist in Computer Science

Houston, Texas

August, 1990

Parameter-Passing and the Lambda Calculus

Erik Crank

Abstract

The choice of a parameter-passing technique is an important decision in the design of a high-level programming language. To clarify some of the semantic aspects of the decision, we develop, analyze, and compare modifications of the λ-calculus for the most common parameter-passing techniques. More specifically, for each parameter-passing technique we provide

1. a program rewriting semantics for a language with side-effects and first-class procedures based on the respective parameter-passing technique;
2. an equational theory derived from the rewriting semantics;
3. a formal analysis of the correspondence between the calculus and the semantics; and
4. a strong normalization theorem for the largest possible imperative fragment of the theory.

A comparison of the various systems reveals that Algol's call-by-name indeed satisfies the well-known β rule of the original λ-calculus, but at the cost of complicated axioms for the imperative part of the theory. The simplest and most appealing axiom system appears to be the one for a call-by-value language with reference cells as first-class values.

Acknowledgments

First and foremost I wish to thank my advisor, teacher and friend Matthias Felleisen for introducing me to the *essence* of computer science. Without his constant enthusiasm and encourgement I could not have completed this work.

Thanks are also due to the members of my thesis committee, Corky Cartwright, Joe Warren and Bruce Duba for their suggestions for improvements to this research. In addition, my fellow students here are Rice have provided me with much assistance, advice and support throughout these last two years.

Finally, I am indebted to my dear wife Laura for her unfailing love and support during this difficult time. Although thousands of miles away, she has comforted me, encouraged me, and given me the confidence I needed to finish this thesis.

Contents

Chapter 1

Introduction

The choice of a parameter-passing technique is an important element in the design of a high-level programming language. The wide variety of techniques in modern languages, e.g., call-by-value, call-by-name, pass-by-reference, suggests a lack of a consensus about the advantages and disadvantages of the various techniques. In this thesis we analyze the most common techniques by studying and comparing equational theories for each of them.

The first to consider equational theories for the analysis of parameter-passing techniques was Plotkin [22]. Starting from the folklore that Church's λ-calculus captures the essence of call-by-name in a *functional* language, he developed a variant of the calculus, the λ_v-calculus, to formalize Landin's [17] notion of call-by-value in a simple framework comparable to the λ-calculus. More importantly, he used these two examples, call-by-name and call-by-value, to analyze the formal relationship between programming languages and calculi. Both the λ-calculus and the λ_v-calculus satisfy general correspondence conditions with respect to the appropriate semantics: (1) the calculi are sufficiently strong to evaluate a program to its answer and (2) they are sound in the sense that the equality of terms in the calculus implies their interchangeablility in programs.

Recently, Felleisen et al [8, 9, 10] extended Plotkin's work to call-by-value programming languages with *imperative* constructs like assignments and jumps. Their result shows that like functional constructs, imperative constructs have a simple operational semantics, and that there are conservative extensions of the λ_v-calculus for reasoning about them.

The extension of Plotkin's work to imperative languages is important because it enables us to consider a broader spectrum of parameter-passing techniques. Specifically, it is only through the addition of imperative constructs that the design options for alternative parameter-passing techniques become interesting. Whereas in a functional setting the only observable differences between the call-by-value and call-by-name versions of a program is termination behavior, the two versions of the same program can produce different results in imperative languages. Moreover, in the presence of assignments, it is also possible to distinguish parameter-passing protocols that bind parameters to values from those that bind parameters to "references" to values.

Based on these observations, we classify parameter-passing techniques according to their respective *evaluation* and *binding* strategies. The evaluation strategy determines when to evaluate the procedure arguments, while the binding strategy determines the correspondence between formal parameters and arguments. The two prevailing evaluation strategies are *eager* evaluation (call-by-value), which evaluates the argument to a procedure call *before* binding the parameter, and *delayed* evaluation (call-by-name), which does not evaluate the argument until the value of the parameter is needed. As for binding strategies we consider *pass-by-worth*, which binds the parameter to the *value*, i.e., "worth," of the argument, and *pass-by-reference*, which binds the parameter to a variable, i.e., a "reference" to a value. A third strategy, *pass-by-value-result*, has properties of both pass-by-worth and pass-by-reference.

In this thesis we analyze the parameter-passing techniques produced by the various combinations of evaluation and binding strategies. In addition, we examine the *reference cell*, a language construct that provides an alternative to the pass-by-reference technique. For each technique we develop an operational semantics and a calculus for a higher-order programming language that uses the technique. The calculi satisfy variants of Plotkin's correspondence criteria. In addition, we show that the imperative fragments satisfy strong normalization theorems. A comparison of the calculi reveals some aspects of the relative semantic complexity of the parameter-passing techniques. In particular, this study verifies the folklore that Algol's call-by-name parameter-passing corresponds to a calculus with the famous β rule in its full generality, even in the presence of side-effects. However, this correspondence comes at the expense of complicated axioms for the imperative fragment of the calculus. The simplest system is the one for the call-by-value/pass-by-worth language with reference cells as values. It is a conservative extension of the λ_v-calculus, satisfies a correspondence theorem and has a decidable imperative fragment. Furthermore, reference cells in the call-by-value/pass-by-worth language provide some of the capabilities of call-by-value/pass-by-reference.

The next two chapters of this thesis examine different combinations of binding and evaluation strategies. Chapter 2 deals with techniques that use the eager evaluation strategy while Chapter 3 examines those using delayed evaluation. Also included in Chapter 2 is a discussion of the reference cell. Each section defines an operational semantics and a corresponding calculus for a particular parameter-passing technique. Chapter 4 discusses languages with multiple parameter-passing techniques. Finally, we study some simple examples, discuss related work and form conclusions in the last chapter.

Chapter 2

Eager Evaluation

The evaluation strategy of a language dictates when to evaluate the argument in a procedure application. In the eager strategy, also known as *strict* or *call-by-value*, the evaluation of the argument expression occurs before the evaluation of the procedure body. In this chapter, we consider three binding strategies for call-by-value languages: pass-by-worth, pass-by-reference and pass-by-value-result. The last section discusses the reference cell, a language construct that provides an alternative to certain parameter-passing techniques.

2.1 Call-by-value/Pass-by-worth

Many common programming languages, e.g., C, Fortran, ML, Pascal and Scheme, employ the *call-by-value/pass-by-worth* parameter-passing technique. In these languages, a procedure application evaluates the argument expression *before* binding the formal parameter to the *value* of the argument. This section, which serves as the basis for the rest of the thesis, presents the call-by-value/pass-by-worth language and a calculus for reasoning about the language. Both the language and calculus are variants of the work by Felleisen and Hieb [9].

Syntax

The call-by-value term language, *Idealized Scheme* or IS_v, extends the language of the λ-calculus with two new expressions. First, there is an assignment statement, (set! x e), which *assigns* to the variable x the *value* of the expression e. The result of an assignment expression is the value that is assigned to the variable. Second, there is a ρ-expression, which is similar to a block in Algol and the **letrec** expression in Scheme. It contains a sequence of variable-value pairs and a subexpression. The ρ-expression establishes mutually recursive bindings of the variables to their associated values and returns the value of the subexpression.

Definition 2.1.1. (*Call-by-value Term Language IS_v*) *Vars* denotes the set of variables. *Consts* is an unspecified set of constants with a subset of functional constants, *FConsts*. Let x range over *Vars*, c over *Consts* and f over *FConsts*. The set of expressions in IS_v consists of values, variables, applications, assignments and ρ-expressions. Values are constants and λ-abstractions:

$$\begin{array}{rcll} e & ::= & v \mid x \mid (e\ e) \mid (\texttt{set!}\ x\ e) \mid \rho\theta.e & \quad (\textit{Expressions}) \\ v & ::= & c \mid \lambda x.e. & \quad (\textit{Values}) \end{array}$$

A ρ-list is a sequence of variable-value pairs in which the variables are pairwise distinct:

$$\theta \ ::= \ \epsilon \mid \theta(x, v), \text{ where } (x, v') \notin \theta. \qquad (\rho\textit{-lists})$$

The subexpression e is the *body* of the λ-abstraction $\lambda x.e$, and of the ρ-expression $\rho\theta.e$. The *variable position* of a `set!` expression is the leftmost subexpression.

There are two basic differences between this language and the language of Felleisen and Hieb [9]. First, we use the assignment statement `set!` instead of the *sigma capability* introduced by Felleisen. Although the sigma capability is a powerful programming construct, most languages implement weaker constructs such as the `set!` expression. Second, like in Scheme, all variables are assignable, that is, they may occur in the variable position of a `set!` expression. Therefore, unlike in the λ_v-calculus [22] and the λ_v-S-calculus [9], variables in our language are expressions, not values.[1]

The λ-abstractions and ρ-expressions are binding expressions. The λ-abstraction $\lambda x.e$ binds the variable x in the body e. The ρ-expression $\rho(x_1, v_1)\ldots(x_n, v_n).e$ binds the variables $x_1, \ldots, x_n$ in the body e and in each of the values $v_1, \ldots, v_n$. The set of free variables in an expression e is denoted $FV(e)$. An expression is *closed* if it contains no free variables. Following Barendregt's [1] conventions, we assume that the bound variables are distinct from the free variables in all expressions, and we identify expressions that differ only by a renaming of the bound variables. Similarly, we identify ρ-lists that differ only by the ordering of their pairs:

$$(x_1, v_1)\ldots(x_n, v_n) \equiv (x_{i_1}, v_{i_1})\ldots(x_{i_n}, v_{i_n}), \text{for all permutations } i_1, \ldots, i_n \text{ of } 1, \ldots, n.$$

[1]Although variables are values in the λ_v-calculus, the addition of an assignment statement to the language requires that assignable variables not be values. Felleisen et al [8, 9] conservatively extend the λ_v-calculus by enlarging the set of variables with a set of assignable variables. The original unassignable variables of the λ_v-calculus are *values*, whereas the additional assignable variables are not. To simplify matters, we abandon this distinction.

Put differently, we treat ρ-lists as *sets* of pairs and in some circumstances as *finite functions.* Accordingly, we refer to a ρ-list as a *ρ-set* and use standard set operations on these sets.

The expression $e[x \leftarrow e']$ denotes the expression resulting from the substitution of all free occurrences of the variable x in e with the expression e'. The set of *assigned variables* in an expression e, denoted $AV(e)$, is the set of *free* variables in e that occur in the variable position of a **set!** expression.

A *context* is a term with a "hole" ([ ]) in the place of a subexpression. Formally, the set of contexts over IS_v is defined as follows:

$$C ::= [\,] \mid (e\ C) \mid (C\ e) \mid (\textbf{set!}\ x\ C) \mid \lambda x.C \mid \rho\theta \cup \{(x, C)\}.e \mid \rho\theta.C$$

If C is a context, then $C[e]$ is the expression produced by replacing the hole in C with the expression e.

The set of constants, *Consts*, is not specified, but is intended to represent basic objects such as primitive functions, numbers and booleans. A partial function δ, from functional constants and closed values ($Values^0$) to closed values, provides the interpretation of functional constants:

$$\delta : FConsts \times Values^0 \longrightarrow Values^0$$

We restrict the interpretation function so that functional constants cannot distinguish λ-abstractions. That is, if $\delta(f, v)$ is defined for some λ-abstraction v, then for all λ-abstractions v', $\delta(f, v) = \delta(f, v')$. This restriction on functional constants is reasonable in the context of most programming languages[2]. Essentially, it prohibits from the language only those functions that can examine the text of procedure bodies, but permits functions that can distinguish constants from procedures, such as the Scheme predicates int? and proc?.

Semantics

The semantics of the call-by-value/pass-by-worth language is a partial function, $eval_{vw}$, from *programs* to *answers*, where a program is a closed expression and an answer is either a value, or a rho expression whose body is a value:

$$a \ ::= \ v \mid \rho\theta.v \qquad (Answers)$$

[2]For reflective languages, in which programs have knowledge about program text, this restriction is too strong. Our work does not address such languages [19].

We specify this function via an abstract machine that rewrites programs according to a *program transformation function*. More precisely, the machine partitions the program into an *evaluation context* and a *redex*. The evaluation context is a special kind of context that specifies the evaluation order of the subexpressions in a compound expression. Intuitively, the hole in an evaluation context points to the next subexpression to be evaluated. A *call-by-value* evaluation context specifies the eager evaluation strategy:

$$E_v ::= [\,] \mid (v\ E_v) \mid (E_v\ e) \mid (\text{set! } x\ E_v)$$

A redex is an expression that specifies how the transformation function rewrites the program. The call-by-value/pass-by-worth language redexes are the following expressions: fv, $(\lambda x.e)v$, x, $(\text{set! } x\ v)$, and $\rho\theta.e$. According to the type of redex, the transformation function rewrites the entire program to a new program. The machine continues to apply the transformation function until it produces an answer. Some programs do not produce answers, either because the rewriting process never terminates or because the transformation function is not defined on the programs. For these programs, $eval_{vw}$ is undefined. Finally, the machine 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 ($\mathbf{gc} = gc \cup elim$):

$$\rho\theta_0 \cup \theta_1.e \longrightarrow \rho\theta_1.e, \text{ if } \theta_0 \neq \varnothing \text{ and } FV(\rho\theta_1.e) \cap Dom(\theta_0) = \varnothing \qquad (gc)$$
$$\rho\varnothing.e \longrightarrow e \qquad (elim)$$

It is easy to verify that the notion of reduction **gc** is strongly normalizing and Church-Rosser. Therefore, all expressions in IS_v have a unique **gc**-*normal form* (**gc**-nf). Furthermore, the **gc**-nf of an answer is also an answer.

Definition 2.1.2. ($eval_{vw}$) The transformation function for the call-by-value/pass-by-worth language, $\triangleright_{vw}$, is a partial function from IS_v-programs to IS_v-programs defined by the transition rules given in Figure 2.1. The transitive closure of $\triangleright_{vw}$ determines the call-by-value/pass-by-worth semantics of IS_v:

$$eval_{vw}(e) = a \text{ if } \rho\varnothing.e \triangleright^*_{vw} \rho\theta.v, \text{ and } a \text{ is the } \mathbf{gc}\text{-nf of } \rho\theta.v.$$

If $eval_{vw}(e) = a$, then we say the evaluation of e *terminates* with answer a. If $eval(e)$ is undefined, we say e *diverges*. If $\rho\varnothing.e \triangleright^*_{vw} e'$, then e *evaluates* to e'.

$$\begin{array}{rcll}
\rho\theta.E_v[fv] & \triangleright_{vw} & \rho\theta.E_v[\delta(f,v)], \text{ if } \delta(f,v) \text{ defined.} & (E.\delta) \\
\rho\theta.E_v[(\lambda x.e)v] & \triangleright_{vw} & \rho\theta.E_v[e[x \leftarrow v]], \text{ if } x \notin AV(e) & (E.\beta_v) \\
\rho\theta.E_v[(\lambda x.e)v] & \triangleright_{vw} & \rho\theta.E_v[\rho\{(x,v)\}.e], \text{ if } x \in AV(e) & (E.\beta_{v\sigma}) \\
\rho\theta \cup \{(x,v)\}.E_v[x] & \triangleright_{vw} & \rho\theta \cup \{(x,v)\}.E_v[v] & (D_v) \\
\rho\theta \cup \{(x,u)\}.E_v[(\text{set! } x\ v)] & \triangleright_{vw} & \rho\theta \cup \{(x,v)\}.E_v[v] & (\sigma_v) \\
\rho\theta.E_v[\rho\theta'.e] & \triangleright_{vw} & \rho\theta \cup \theta'.E_v[e] & (\rho_\cup)
\end{array}$$

Figure 2.1 Call-by-value/Pass-by-worth Transformation Function

The first two rules in Figure 2.1 specify the semantics of the functional subset of IS_v. The third rule defines the behavior of λ-abstractions in applications when the bound variable is assigned within the body. This rule creates a ρ-set that maps the bound variable to the argument. The fourth and fifth rules give the semantics for a reference to a variable and for an assignment to a variable. A variable reference is replaced by its corresponding value from the ρ-set and an assignment changes the mapping of the variable in the ρ-set. Finally, the last rule joins two ρ-sets when a ρ-expression occurs in the hole of the evaluation context.

To verify that $eval_{vw}$ is a *function*, we need to show that $\triangleright_{vw}$ is a function. The transition rules that define $\triangleright_{vw}$ partition a program into a surrounding ρ-expression, an evaluation context, and a redex. It is easy to show that the definitions for evaluation contexts and redexes ensure that this partition (if it exists) is unique, and therefore at most one transition rule can be applied to any program. Furthermore, since the final answer is the unique **gc-nf** of the result obtained by the transition function, $eval_{vw}$ is a function.

Although $eval_{vw}$ provides a means for determining whether two *programs* have the same behavior, programmers often need to know whether one *subexpression* can be replaced by another without affecting the overall behavior of a program. This notion of "equivalent expressions" is captured by the *observational equivalence* relation. Intuitively, two terms are observationally equivalent if they are interchangeable within all programs without affecting the *observable* behavior of the programs.[3] Two programs have the same observable behavior if either they both diverge or they both converge, and if one evaluates to a basic constant then the other evaluates to the same constant.

Definition 2.1.3. ($\simeq_{vw}$) Two terms are observationally equivalent, $e \simeq_{vw} e'$, if for all contexts C such that both $C[e]$ and $C[e']$ are programs,

- $eval_{vw}(C[e])$ is defined iff $eval_{vw}(C[e'])$ is defined, and
- $eval_{vw}(C[e]) = c$ iff $eval_{vw}(C[e']) = c$, for some basic constant c.

Calculus

Proving the observational equivalence of two terms is a difficult task. A proof must show that for *all* program contexts, the two terms are interchangeable without affecting the program behavior. Our goal is to develop a *simple* equational theory that can prove terms observationally equivalent. A first step is the λ_v-W-calculus, an equational theory that closely resembles the state calculus of Felleisen and Hieb [9]. The definition of the λ_v-W-calculus is given in Definition 2.1.4.

The calculus is derived from the semantic program transformation function. The axioms δ, β_v and $\beta_{v\sigma}$ are simplifications of the corresponding relations $E.\delta$, $E.\beta_v$, $E.\beta_{v\sigma}$. The ρ_{lift} axiom lifts a ρ-expression from within an evaluation context. The gc reductions equate terms that differ only by unneeded items in ρ-sets. The other axioms are identical to their corresponding rules in the transformation function. In fact, we could construct the same calculus directly from the transformation function by taking $\triangleright_{vw} \cup$ gc as the set of axioms instead of vw. However, for proving meta-theorems about the calculus the above definition is more appropriate.

We defined a semantics for a language and then derived an equational calculus from the semantics. Before proving that the calculus corresponds to the semantics, we must first give a formal definition of "correspondence." Plotkin [22] gave two criteria for correspondence between a calculus and a semantics. First, the calculus must be sufficiently strong to evaluate programs. In particular, if the semantics evaluates a

[3]For historical reasons, some authors call this relation *operational* equivalence. To avoid confusion, we use the term observational because the relation is based upon observable characteristics, independent of an operational semantics.

Definition 2.1.4. ($\boldsymbol{\lambda}_v$-W) The theory $\boldsymbol{\lambda}_v$-**W** is based upon a set of axioms that includes the relations D_v, σ_v, $\rho_\cup$ in Figure 2.1, the notion of reduction gc and the following axioms:

$$\begin{array}{rcll} fv &=& \delta(f,v) & (\delta)\\ (\lambda x.e)v &=& e[x \leftarrow v],\ \text{if } x \notin AV(e) & (\beta_v)\\ (\lambda x.e)v &=& \rho\{(x,v)\}.e,\ \text{if } x \in AV(e) & (\beta_{v\sigma})\\ E_v[\rho\mathit{0}.e] &=& \rho\mathit{0}.E_v[e],\ \text{if } E_v \neq [\,] & (\rho_{lift}) \end{array}$$

The complete set of axioms is: $\mathbf{vw} = \delta \cup \beta_v \cup \beta_{v\sigma} \cup D_v \cup \sigma_v \cup \rho_\cup \cup \rho_{lift} \cup$ **gc**. The theory $\boldsymbol{\lambda}_v$-**W**, also called the λ_v-W-calculus, is the set of formulas $e_1 =_{vw} e_2$, where $e_1, e_2 \in IS_v$ and $=_{vw}$ is the least equivalence relation generated by the compatible closure of **vw**:

$$\begin{array}{rcll} (e_1,e_2) \in \mathbf{vw} &\Rightarrow& e_1 =_{vw} e_2 & (\mathit{Axioms})\\ e_1 =_{vw} e_2 &\Rightarrow& C[e_1] =_{vw} C[e_2],\ \text{for context } C & (\mathit{Compatible})\\ && e_1 =_{vw} e_1 & (\mathit{Reflexive})\\ e_1 =_{vw} e_2 &\Rightarrow& e_2 =_{vw} e_1 & (\mathit{Symmetric})\\ e_1 =_{vw} e_2, e_2 =_{vw} e_3 &\Rightarrow& e_1 =_{vw} e_3 & (\mathit{Transitive}) \end{array}$$

We write $\boldsymbol{\lambda}_v$-**W** $\vdash e_1 = e_2$ if $e_1 =_{vw} e_2$.

program e to a, then there should be a proof in the calculus $\boldsymbol{\lambda}_v$-**W** $\vdash e = a$. Similarly, if the calculus proves that a program is equivalent to an answer, then the program should evaluate to an answer. Second, the calculus must be sound with respect to the observational equivalence relation. In other words, if two terms are equal in the calculus, they are observationally equivalent with respect to $eval_{vw}$.

The Correspondence Theorem states that the λ_v-W-calculus corresponds to the call-by-value/pass-by-worth language.

Theorem 2.1.5 (Correspondence) *The λ_v-W-calculus corresponds to the call-by-value/pass-by-worth semantics of IS_v, $eval_{vw}$. In particular,*

1. $\boldsymbol{\lambda}_v$-**W** *is adequate:*

   *(a) if $eval_{vw}(e) = a$, then $\boldsymbol{\lambda}_v$-**W** $\vdash e = a$,*

   *(b) if $\boldsymbol{\lambda}_v$-**W** $\vdash e = a$, then $eval_{vw}(e)$ is defined; and*

2. $\boldsymbol{\lambda}_v$-**W** *is sound with respect to $\simeq_{vw}$:*

$$\boldsymbol{\lambda}_v\text{-}\mathbf{W} \vdash e_1 = e_2 \textit{ implies } e_1 \simeq_{vw} e_2.$$

Proof. Adequacy follows from the fact that $\triangleright_{vw}$ is a subset of $=_{vw}$. For soundness, we interpret the axioms from left to right to define a *notion of reduction* **vw**. The reduction satisfies Church-Rosser and Curry-Feys Standardization lemmas that imply the soundness of the calculus.

The proof is an adaptation of the proofs for the calculi of Plotkin [22] and Felleisen et al [8, 9, 10] and are long and tedious. The complete proof is given in Appendix A.1. ∎

A secondary interesting question about a calculus is whether it is decidable. In our case, the imperative subtheory ρ_v, that is, the theory based upon the axioms in **vw** that do *not* deal with procedure applications, is decidable. Specifically, the notion of reduction consisting of these axioms is strongly normalizing. This is a new result, which does *not* hold for the imperative fragment of the λ_v-S-calculus. It is motivated in part by the completeness theorem for a non-recursive fragments of first-order Lisp in Mason and Talcott [18].

Theorem 2.1.6 (Strong Normalization) *Let δ_c be the restriction of the relation δ as follows:*

$$(f\ v) = \delta(f, v), \ \textit{where } \delta(f, v) \in \textit{Consts} \qquad (\delta_c)$$

Let $\mathbf{s}_v = \delta_c \cup D_v \cup \sigma_v \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}$. *The notion of reduction* $\mathbf{s}_v$ *is strongly normalizing.*

Proof. The proof uses a size argument. We define a "potential" function, P : *Expressions* $\longrightarrow \mathsf{N} \times \mathsf{N}$, such that $P(e) \succ P(e')$, by lexicographical ordering, if $e \longrightarrow_{s_v} e'$. This function counts the number of redexes in a term, and also takes into account potential redexes that may be introduced in further reductions. See Appendix A.1 for the complete proof. ∎

In subsequent sections, if a theory **th** satisfies the two correspondence criteria with respect to an evaluation function *eval*, we write **th** $\models$ Corr(*eval*). Similarly, we write **r** $\models$ SN when the notion of reduction **r** is strongly normalizing.

2.2 Call-by-value/Pass-by-reference

Some programming languages such as Pascal and Fortran allow *pass-by-reference* procedure parameters in addition to pass-by-worth parameters. Generally, these languages require that the actual argument for a pass-by-reference parameter be a variable. A procedure application binds the parameter to the variable so that during the evaluation of the procedure body, references and assignments to the formal parameter are equivalent to references and assignments to the actual argument.

The motivation behind pass-by-reference parameter-passing is the following. Suppose there are several subexpressions of a program that are identical except for some differing variable names. It would seem natural to write a procedure that abstracts over the different variables in the subexpressions and to replace each occurrence of the expression in the program with a procedure call. This does not work in the pass-by-worth semantics, however, because if this procedure contains an assignment to a parameter, then by definition, the assignment will not affect the argument of the procedure call, and the resulting program may not have the same meaning as the original. Pass-by-reference parameters overcome this deficiency, since all references to the parameter in the body become references to the argument.

Semantics

Unlike most languages, our semantics generalizes the notion of pass-by-reference to allow arbitrary expressions as arguments to procedures.[4] For the case in which the argument is not a variable, the transformation function evaluates the argument until it becomes a variable before the procedure call. If the argument does not evaluate to a variable, then the meaning of the program is undefined.

The term language for the pass-by-reference semantics is IS_v, except that we interpret $\lambda x.e$ as a pass-by-reference procedure. Two changes to the pass-by-worth program transformation function in Figure 2.1 are necessary to obtain the transformation function for the pass-by-reference semantics:

- When a variable occurs as an argument to a λ-abstraction, the transformation function substitutes the argument variable for the bound variable within the body of a λ-abstraction:

  $$\rho\theta.E_v[(\lambda x.e)y] \rhd \rho\theta.E_v[e[x \leftarrow y]] \qquad (E.\beta_r)$$

  The rule $E.\beta_r$ replaces the two rules ($E.\beta_v$, $E.\beta_{v\sigma}$) for procedure application.

- The transformation function does not dereference a variable when it occurs as an argument to a lambda abstraction:

  $$\rho\theta \cup \{(x,v)\}.E_v[x] \rhd \rho\theta \cup \{(x,v)\}.E_v[v], \text{ if } E_v \neq E'_v[(\lambda y.e)[\,]] \qquad (E.D'_v)$$

[4]Although some versions of Fortran permit arguments other than variables, the semantics of these cases is defined by the *pass-by-worth* rule $E.\beta_{v\sigma}$. In this section we restrict our attention to pass-by-reference.

This dereference rule $E.D'_v$ replaces D_v.

The call-by-value/pass-by-reference transformation function, $\triangleright_{vr}$, is the union of the two new rules and the previously defined rules in Figure 2.1:

$$\triangleright_{vr} = E.\delta \cup E.\beta_r \cup E.D'_v \cup \sigma_v \cup \rho_\cup$$

As in the pass-by-worth semantics, the transformation function and the notion of reduction **gc** determine the semantics:

$$eval_{vr}(e) = a \text{ if } \rho\emptyset.e \triangleright^*_{vr} \rho\theta.v, \text{ and } a \text{ is the } \mathbf{gc}\text{-nf of } \rho\theta.v$$

The observational equivalence relation, $\simeq_{vr}$, for this language is based upon $eval_{vr}$.

In order for $eval_{vr}$ to be a function, the transformation relation $\triangleright_{vr}$ must also be a function. The $\triangleright_{vr}$ relation is a function because only one transition rule can apply to a given program. Although there may not be a unique partition of a program into an evaluation context and a redex as in the call-by-value/pass-by-worth language, the added condition on the $E.D'_v$ rule eliminates the one case in which two rules could have applied to a program.

Calculus

Only two simple modifications to the pass-by-worth transformation function were needed to define a pass-by-reference semantics. Presumably, since the pass-by-worth calculus was derived from the transformation function, only simple modifications are necessary to define an equational theory for the pass-by-reference semantics. Our approach is to make changes to the pass-by-worth calculus that correspond to the two changes needed for the transformation function. First, a new axiom β_r replaces β_v and β_σ:

$$(\lambda x.e)y = e[x \leftarrow y] \qquad (\beta_r)$$

Second, the $E.D'_v$ relation *could* serve as the variable dereference axiom instead of D_v. Unfortunately, the $E.D'_v$ is an *unsound* axiom for a variable dereference in the empty context. For example, with $E.D'_v$ it is possible to prove the following equivalence:

$$(\lambda y.c)\rho\{(x,v)\}.x = (\lambda y.c)\rho\{(x,v)\}.v$$

These two programs are not observationally equivalent, however, because the meaning of the left program is c while the meaning of the right is undefined. Fortunately,

$E.D'_v$ is unsound only for this case. Adding an appropriate restriction to D_v solves the problem:

$$\rho\theta \cup \{(x,v)\}.E_v[x] \;=\; \rho\theta \cup \{(x,v)\}.E_v[v], \qquad (D'_v)$$
$$\text{if } E_v \neq E'_v[(\lambda y.e)[\,]] \text{ and } E_v \neq [\,]$$

This rule together with β_r and the others from the pass-by-worth calculus form the basis of the λ_v-R-calculus. The set of axioms is:

$$\mathbf{vr} = \delta \cup \beta_r \cup D'_v \cup \sigma_v \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}.$$

As before, the theory $\boldsymbol{\lambda}_v$-**R** consists of the equivalences $e_1 =_{vr} e_2$, where $=_{vr}$ is the equivalence relation generated by the compatible closure of the axioms in **vr**. We write $\boldsymbol{\lambda}_v$-**R** $\vdash e_1 = e_2$ if $e_1 =_{vr} e_2$.

Because of the soundness problem encountered with the D'_v rule, the calculus does *not* exactly correspond to the semantics by Plotkin's criteria. The condition on the D'_v reduction relation restricts the ability of the calculus to evaluate programs. Specifically, for some programs the calculus can only evaluate them to an expression that is "one step away" from the answer by the transformation function. Based on this observation, we prove a *Weak* Correspondence Theorem.[5]

Theorem 2.2.1 (Weak Correspondence) *The λ_v-R-calculus* weakly *corresponds to the call-by-value/pass-by-reference semantics of IS_v, $eval_{vr}$. In particular,*

1. $\boldsymbol{\lambda}_v$-**R** *is "almost" adequate:*

   (a) *if $eval_{vr}(e) = a$ then either*

   - $\boldsymbol{\lambda}_v$-**R** $\vdash e = a$*, or*
   - $\boldsymbol{\lambda}_v$-**R** $\vdash e = \rho\theta \cup \{(x,v)\}.x$ *and* $\boldsymbol{\lambda}_v$-**R** $\vdash \rho\theta \cup \{(x,v)\}.v = a$,

   (b) *if* $\boldsymbol{\lambda}_v$-**R** $\vdash e = a$*, then $eval_{vr}(e)$ is defined; and*

2. $\boldsymbol{\lambda}_v$-**R** *is sound with respect to $\simeq_{vr}$:*

$$\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e_1 = e_2 \textit{ implies } e_1 \simeq_{vr} e_2.$$

[5]An alternative to formulating a weak correspondence relationship would be to redefine the semantics to correspond to the calculus. In particular, if we treat the expression $\rho\theta.x$ as an *answer* in the semantics, then the calculus satisfies the correspondence criteria. However, programming languages do not use this notion of call-by-value/pass-by-reference semantics.

Proof. The proof has the same structure as the previous correspondence theorem and is given in Appendix A.2. ∎

If a theory **th** satisfies a weak correspondence theorem with respect to an evaluation function *eval*, we write $\mathbf{th} \models \text{WCorr}(eval)$.

The imperative fragment of the pass-by-reference calculus, ρ'_v, differs from ρ_v only with respect to the dereference axioms D_v and D'_v. Not surprisingly, ρ'_v is also decidable. In addition, the procedure call reduction, β_r is strongly normalizing.

Theorem 2.2.2 (Strong Normalization) *Let* $\mathbf{s}'_v = \delta_c \cup D'_v \cup \sigma_v \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}$. *Then,* $\mathbf{s}'_v \models \text{SN}$ *and* $\beta_r \models \text{SN}$.

Proof. The proof for $\mathbf{s}'_v$ is a straightforward adaptation of the proof for pass-by-worth. For β_r, a simple size argument suffices. ∎

2.3 Call-by-value-result

The programming languages Ada [26] and Algol W [27] specify a parameter-passing technique known as *call-by-value-result*, or *copy-in/copy-out*.[6] This parameter-passing technique uses the eager evaluation strategy and is similar to both the pass-by-worth and pass-by-reference binding techniques. Like pass-by-reference, arguments to procedures must be variables. Like pass-by-worth, the procedure applications bind the formal parameter to the *value* of the argument variable. However, after evaluation of the procedure body, the argument variable receives the value of the formal parameter. In other words, the value of the argument is "copied in" to the parameter before the procedure call and the value of the parameter is "copied out" to the argument after the call.[7] As a result, assignments to the formal parameter affect the argument, but only *after* the evaluation of the procedure body.

This technique addresses a problem that occurs with aliasing parameters in pass-by-reference. Specifically, in pass-by-reference if two formal parameters refer to the same variable during evaluation of the procedure body, then assignments to one parameter also affect the other parameter. This does not occur in call-by-value-result because formal parameters are bound to the value of the argument and the assignments to the formals do not affect the argument variable until *after* the evaluation of

[6]Technically, the term "call-by-value/pass-by-value-result" is the name consistent with our terminology. We use the more popular name *call-by-value-result* for simplicity.

[7]Some languages also specify "call-by-result" parameter-passing in which only the copy-out is performed. This method is useful only in passing information out of a procedure and is a special case of copy-in/copy-out.

the procedure body. In programs that contain no aliased parameters, the semantics of call-by-value-result is the same as call-by-value/pass-by-reference [5].

Syntax

The sequential nature of the call-by-value-result parameter-passing technique requires a new language construct for the specification of the semantics.[8] We extend the set of expressions in IS_v with a *sequence* statement:

$$e ::= \ldots \mid \langle e; e \rangle$$

The expression $\langle e_1; e_2; \ldots e_n \rangle$ abbreviates $\langle e_1; \langle e_2; \ldots e_n \rangle\rangle$. The rest of IS_v remains the same, except that $\lambda x.e$ stands for a call-by-value-result procedure.

Semantics

The semantics of the sequence expression is straightforward: the left sub-expression is evaluated first followed by the right. The value of the whole expression is the value of the second sub-expression. An extended notion of evaluation context specifies the order of evaluation:

$$E_v ::= \ldots \mid \langle E_v; e \rangle$$

The program transformation rule for sequence statements discards the value obtained from evaluating the left sub-expression:

$$\rho\theta.E_v[\langle v; e \rangle] \rhd \rho\theta.E_v[e] \qquad (E.seq)$$

A procedure application requires four steps. First, the application binds the procedure parameter to the *value* of its argument. Second, evaluation continues with the procedure body. Third, the value bound to the parameter is "copied" back to the argument. Finally, the value of the procedure body is returned as the result of the application. The sequence expression specifies this series of events as follows:

$$\rho\theta.E_v[(\lambda x.e)y] \rhd \rho\theta.E_v[\rho\{(x,c),(r,c)\}.\langle(\mathbf{set!}\ x\ y);(\mathbf{set!}\ r\ e);(\mathbf{set!}\ y\ x);r\rangle] \quad (E.\beta_c)$$

The variable r is a new variable that holds the return value of the procedure body. The constant c is an arbitrary initial value. The first assignment "copies in" the

[8]In the call-by-value/pass-by-worth language λ-abstractions are sufficient for specifying the necessary sequential behavior. However, the different semantics of procedures in the call-by-value-result language precludes this possibility, so we must introduce a new construct.

value of the argument to the parameter. The second assignment evaluates the body of the procedure and assigns the result to r. The third assignment copies the value of the parameter x back to the argument y and the last expression simply returns the result r.

The complete transformation function relies on these two rules, previously defined rules in Figure 2.1, and the $E.D'_v$ rule from the pass-by-reference transformation function:

$$\triangleright_{cc} = E.\delta \cup E.\beta_c \cup E.D'_v \cup \sigma_v \cup \rho_\cup \cup E.seq$$

As usual, the transitive closure of $\triangleright_{cc}$ and the notion of reduction **gc** define the semantics $eval_{cc}$. Observational equivalence, $\simeq_{cc}$, is based upon $eval_{cc}$.

Calculus

The equational calculus for the call-by-value-result language closely resembles the λ_v-R-calculus for the call-by-value/pass-by-reference language. The sequence expression requires an additional axiom:

$$\langle v; e\rangle = e \qquad (seq)$$

The axiom for procedure application, β_c, specifies the same order of evaluation as $E.\beta_c$:

$$(\lambda x.e)y = \rho\{(x,c),(r,c)\}.\langle(\text{set! } x\ y);(\text{set! } r\ e);(\text{set! } y\ x);r\rangle \qquad (\beta_c)$$

The problems encountered in the previous section with the $E.D'_v$ relation as a dereference rule also occur with the call-by-value-result calculus; the same D'_v axiom solves the problem. The basic set of axioms for the call-by-value-result theory $\boldsymbol{\lambda_v}$**-VR** consists of these and other axioms:

$$\mathbf{cc} = \delta \cup \beta_c \cup D'_v \cup \sigma_v \cup \rho_\cup \cup \rho_{lift} \cup seq \cup \mathbf{gc}$$

The calculus is constructed in the usual way; we write $\boldsymbol{\lambda_v}\textbf{-VR} \vdash e = e'$ if $e =_{cc} e'$.

Again, the problem with the D'_v axiom prevents an exact correspondence between the calculus and the semantics. However, as with the pass-by-reference calculus, the call-by-value-result calculus satisfies a Weak Correspondence Theorem.

Theorem 2.3.1 (Weak Correspondence) $\boldsymbol{\lambda_v}\textbf{-VR} \models \text{WCorr}(eval_{cc})$

The imperative fragment of this calculus extends the imperative fragment of the call-by-value/pass-by-reference calculus with the *seq* axiom. With this addition, the Strong Normalization Theorem for the notion of reduction s'_v also holds for $\mathrm{s}'_v \cup seq$.

Theorem 2.3.2 (Strong Normalization) $\mathrm{s}'_v \cup seq \models \text{SN}$

2.4 Call-by-value/Pass-by-worth: Reference Cells as Values

The motivation for the pass-by-reference parameter passing technique is the need to affect argument variables. To accomplish this, the pass-by-reference passing technique binds the formal parameter to the argument variable so that assignments to the parameter become assignments to the argument. Languages such as Scheme and ML do not use pass-by-reference, but instead have a new class of values, namely, *reference cells* or *boxes* that can achieve a similar result. A reference cell is a language object that refers to a value. A dereference expression returns the value to which a cell refers; an assignment expression changes this value. Because it is a value, when a cell occurs as an argument to a procedure, the application binds the formal parameter to the cell. In this way, assignments to the formal parameter become assignments to the argument, as in pass-by-reference.

Although the reference cell is not a proper parameter-passing technique per se, we include it here because of its relation to both pass-by-worth and pass-by-reference parameter passing.

Syntax

The term language for the call-by-value language with reference cells, IS_b differs significantly from IS_v. We abandon the assignment of IS_v in favor of the reference cell assignment expression. Since variables are no longer assignable, we treat them as values. The term language IS_b extends the language of the λ_v-calculus with the ρ-expression, a set of reference cells and expressions for assigning and dereferencing cells. Adopting Chez Scheme terminology [6], we use the primitives **box**, **setbox!**, and **unbox** to perform these operations in IS_b. Definition 2.4.1 provides the definition of the term language IS_b.

As usual, both λ-abstractions and ρ-expressions are binding expressions, but while λ binds variables, ρ binds reference cells. There is also a notion of *free cells* analogous to free variables.

Semantics

Despite the differences between the reference cell language IS_b and IS_v, the semantics closely resembles call-by-value/pass-by-worth semantics. Since the assignment is a primitive, there is no need for an evaluation context for assignments. The evaluation

Definition 2.4.1. (*Reference Cell Term Language, IS_b*) Let b range over the set of reference cells, *Boxes*. The set of expressions consists of values, applications and ρ-expressions. Values are constants, boxes, variables, λ-abstractions and primitive operations. The ρ-expression binds reference cells to values.

$$\begin{array}{rcll} e & ::= & v \mid (e\ e) \mid \rho\theta.e & (\textit{Expressions}) \\ v & ::= & c \mid b \mid x \mid \lambda x.e \mid \textbf{box} \mid \textbf{setbox!} \mid \textbf{unbox} \mid (\textbf{setbox!}\ b) & (\textit{Values}) \\ \theta & ::= & \epsilon \mid \theta(b,v),\text{ where } (b,v') \notin \theta. & (\rho\textit{-lists}) \end{array}$$

contexts for this language are the contexts for applications:

$$E ::= [\,] \mid (v\ E) \mid (E\ e)$$

The program transformation function for the call-by-value/pass-by-worth language with reference cells is defined in Figure 2.2. As in the λ_v-calculus the $E.\delta$ and $E.\beta_v$ rules specify the semantics of function and procedure applications. The **box** expression creates a ρ-expression that binds a cell to a value and returns the cell. Cell dereference gets the cell value from the ρ-set and cell assignment changes the cell value in the ρ-set.

$$\begin{array}{rcll} \rho\theta.E[fv] & \rhd_{bx} & \rho\theta.E[\delta(f,v)] & (E.\delta) \\ \rho\theta.E[(\lambda x.e)v] & \rhd_{bx} & \rho\theta.E[e[x \leftarrow v]] & (E.\beta_v) \\ \rho\theta.E[\textbf{box}\ v] & \rhd_{bx} & \rho\theta.E[\rho\{(b,v)\}.b] & (E.box) \\ \rho\theta \cup \{(b,v)\}.E[\textbf{unbox}\ b] & \rhd_{bx} & \rho\theta \cup \{(b,v)\}.E[v] & (D_b) \\ \rho\theta \cup \{(b,u)\}.E[(\textbf{setbox!}\ b)v)] & \rhd_{bx} & \rho\theta \cup \{(b,v)\}.E[v] & (\sigma_b) \\ \rho\theta.E[\rho\theta'.e] & \rhd_{bx} & \rho\theta \cup \theta'.E[e] & (\rho_\cup) \end{array}$$

Figure 2.2 Reference Cell Transformation Function

Remark. Although some of the rules in Figure 2.2 have the same names as relations introduced in previous sections, there are differences due to the change in definition of the evaluation contexts. Throughout the rest of this thesis we give the same name to relations that share reduction *schema*. The precise specifications of relations can be determined from context. **End.**

As usual, the transitive closure of the transformation function and the garbage collection reductions define the semantics $eval_{bx}$. The notion of reduction **gc** is slightly different in this setting because it uses the notion of free *cells* instead of free variables to determine the unneeded items in ρ-sets. Observational equivalence for this language, $\simeq_{bx}$, is based upon $eval_{bx}$.

Calculus

The theory for reasoning with reference cells, $\boldsymbol{\lambda}_v$-**B**, resembles the λ_v-W-calculus. In addition to δ and β_v, we need a relation for **box** expressions:

$$(\mathbf{box}\ v) = \rho\{(b, v)\}.b \qquad (box)$$

The set of axioms is:

$$\mathsf{bx} = \delta \cup \beta_v \cup D_b \cup \sigma_b \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}.$$

The theory $\boldsymbol{\lambda}_v$-**B** is constructed in the usual way from this set of axioms.

Like the call-by-value/pass-by-worth language, this calculus satisfies a strong Correspondence Theorem.

Theorem 2.4.2 (Correspondence) *The λ_v-B-calculus corresponds to the call-by-value/pass-by-worth semantics of IS_b, $eval_{bx}$:*

$$\boldsymbol{\lambda}_v\text{-}\mathbf{B} \models \mathrm{Corr}(eval_{bx})$$

Similarly, the imperative fragment of this calculus is decidable:

Theorem 2.4.3 (Strong Normalization) *Let* $\mathsf{s}_b = \delta_c \cup D_b \cup \sigma_b \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}$. *Then,* $\mathsf{s}_b \models \mathrm{SN}$.

Furthermore, the λ_v-B-calculus is a conservative extension of the λ_v-calculus. Thus, the equivalence proofs from the λ_v-calculus also hold in the λ_v-B-calculus.

Theorem 2.4.4 (Conservative Extension) *Let Λ be the language of the λ-calculus. Then,* $\boldsymbol{\lambda}_v = \boldsymbol{\lambda}_v\text{-}\mathbf{B}|\Lambda$.

Proof. Clearly, any proof in $\boldsymbol{\lambda}_v$ is also a proof in $\boldsymbol{\lambda}_v$-**B**. On the other hand, a proof in $\boldsymbol{\lambda}_v$-**B** may use axioms that are not in $\boldsymbol{\lambda}_v$. In this case, the Church-Rosser property ensures that two equivalent expressions *reduce* to a third expression. These reductions must use only the β_v and δ axioms of the λ_v-calculus since the expressions are in Λ. ∎

Analysis

The reference cell is interesting because of its relation to both pass-by-worth and pass-by-reference languages. The connection to the pass-by-worth language is apparent: the reference cell language employs call-by-value/pass-by-worth parameter passing and box assignments in IS_b are similar to variable assignments in IS_v. In fact, a simple structure-preserving function, $[\![\cdot]\!]$, maps IS_v expressions into IS_b expressions while preserving the meaning of the original expression. This mapping converts a variable reference to an **unbox** expression, an assignment to a cell assignment, and and a procedure to a procedure that first creates a reference cell containing the value of the argument:

$$\begin{aligned} [\![x]\!] &= (\textbf{unbox}\ x) \\ [\![(\textbf{set!}\ x\ e)]\!] &= (\textbf{setbox!}\ x\ [\![e]\!]) \\ [\![\lambda x.e]\!] &= (\lambda y.(\lambda x.[\![e]\!])(\textbf{box}\ y)) \end{aligned}$$

The mappings for the other expressions are just the composition of the mappings of the subexpressions.[9]

For the pass-by-reference language, the connection is not as clear because there is not a simple mapping from the pass-by-reference language into the reference cell language. However, reference cells in the pass-by-worth language provide the ability to abstract over assignments, which was the original motivation for pass-by-reference! The reason is that cells are *values* and can be passed as arguments, whereas variables in IS_v are not values and are dereferenced when they occur as arguments in the pass-by-worth semantics. The pass-by-reference semantics provides an alternative: it passes only variables and does not allow dereferencing variables when they occur as arguments. Reference cells in a call-by-value/pass-by-worth language provide both options within the same, simple language.

[9]Although we do not give the details, by using the expressibility framework of Felleisen [7] we can show that in the language that is the union of IS_b and IS_v, the assignable variables and the variable assignments are *eliminable*.

Chapter 3

Delayed Evaluation

In languages with delayed evaluation, the evaluation of procedure arguments occurs *after* the binding of formal parameters to the argument expressions. In this chapter, we examine the pass-by-worth and pass-by-reference binding strategies in combination with call-by-name evaluation.

3.1 Call-by-name/Pass-by-worth

According to the call-by-name/pass-by-worth parameter passing technique, procedure applications bind the parameters to the unevaluated arguments, and each instance of the formal parameter in the procedure body evaluates the argument to a value.

Syntax

The term language for the call-by-name/pass-by-worth language, IS_n, differs only slightly from IS_v. To exploit the freedom of the delayed evaluation strategy, ρ-expressions do not bind variables to just values, but rather they bind variables to arbitrary expressions:

$$\theta ::= \epsilon \mid \theta(x, e), \text{ where } (x, e') \notin \theta$$

All other syntactic constructs are the same as IS_v, except that procedures represent call-by-name/pass-by-worth procedures.

Semantics

The *call-by-name* evaluation contexts specify the delayed evaluation strategy. Since the context $(\lambda x.e)E$ specifies the evaluation of procedure arguments prior to the procedure call, it cannot be an evaluation context for the delayed evaluation strategy. On the other hand, the application of functional constants remains strict:

$$E_n ::= [\,] \mid (f\ E_n) \mid (E_n\ e) \mid (\text{set!}\ x\ E_n)$$

As usual, we use a program transformation function to define the semantics function $eval_{nw}$. The rules for this function are similar to those of the call-by-value/pass-by-worth function except that arbitrary expressions occur in places in which only values occurred in the call-by-value rules. The program transformation function $\triangleright_{nw}$ is defined in Figure 3.1.

$$
\begin{array}{rcll}
\rho\theta.E_n[fv] & \triangleright_{nw} & \rho\theta.E_n[\delta(f,v)] & (E.\delta) \\
\rho\theta.E_n[(\lambda x.e)e'] & \triangleright_{nw} & \rho\theta.E_n[e[x \leftarrow e']], \text{ if } x \notin AV(e) & (E.\beta_n) \\
\rho\theta.E_n[(\lambda x.e)e'] & \triangleright_{nw} & \rho\theta \cup \{(x_\sigma, e')\}.E_n[e], \text{ if } x \in AV(e) & (E.\beta_{n\sigma}) \\
\rho\theta \cup \{(x,e)\}.E_n[x] & \triangleright_{nw} & \rho\theta \cup \{(x,e)\}.E_n[e] & (D_n) \\
\rho\theta \cup \{(x,e)\}.E_n[(\mathbf{set!}\ x\ v)] & \triangleright_{nw} & \rho\theta \cup \{(x,v)\}.E_n[v] & (\sigma_n) \\
\rho\theta.E_n[\rho\theta'.e] & \triangleright_{nw} & \rho\theta \cup \theta'.E_n[e] & (\rho_\cup)
\end{array}
$$

Figure 3.1 Call-by-name/Pass-by-worth Transformation Function

Calculus

Because of the similarities between the call-by-name/pass-by-worth and the call-by-value/pass-by-worth languages, the derivation of the λ_n-W-calculus is straightforward. The theory $\boldsymbol{\lambda_n}$**-W** is based upon relations in Figure 3.1, previously defined relations and the following axioms for procedure applications:

$$
\begin{array}{rcll}
(\lambda x.e)e' & = & e[x \leftarrow e'], \text{ if } x \notin AV(e) & (\beta_n) \\
(\lambda x.e)e' & = & \rho\{(x,e')\}.e, \text{ if } x \in AV(e) & (\beta_{n\sigma})
\end{array}
$$

The set of axioms for the λ_n-W-calculus is:

$$\mathbf{nw} = \delta \cup \beta_n \cup \beta_{n\sigma} \cup D_n \cup \sigma_n \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}.$$

As before, the relation $=_{nw}$ is the equivalence relation generated by the compatible closure of **nw**. The theory $\boldsymbol{\lambda_n}$**-W** consists of the equivalences $e_1 =_{vw} e_2$. We write $\boldsymbol{\lambda_n}\text{-}\mathbf{W} \vdash e_1 = e_2$ if $e_1 =_{nw} e_2$.

Over the functional subset of IS_n, the β_n relation is equivalent to Church's β axiom [3]. Thus, the λ_n-W-calculus is a conservative extension of the λ-calculus: equivalences in the λ-calculus are preserved in the λ_n-W-calculus.

Theorem 3.1.1 (Conservative Extension) $\lambda = \lambda_n\text{-W}|\Lambda$

The Correspondence Theorem states that the λ_n-W-calculus corresponds to the call-by-value/pass-by-worth language.

Theorem 3.1.2 (Correspondence) *The λ_n-W-calculus corresponds to the call-by-name/pass-by-worth semantics of IS_n, $eval_{nw}$:*

$$\lambda_n\text{-W} \models \mathrm{Corr}(eval_{nw})$$

Proof. The proof is essentially the same as the proof for the call-by-value/pass-by-worth calculus and relies upon the Church-Rosser property and Curry-Feys Standardization lemma for the corresponding notion of reduction. See Appendix A.3. ∎

Because ρ-sets may have recursive references, the imperative subtheory, ρ_n, is *not* decidable on the full language IS_n. For example, there are infinite reduction sequences in ρ_n for the diverging IS_n program $\rho\{(x,x)\}.x$. On the other hand, ρ_n is strongly normalizing over the language IS_v, in which the *ranges* of all ρ-sets are in *Values*.

Theorem 3.1.3 (Strong Normalization) *Let* $\mathbf{s}_n = \delta_c \cup D_n \cup \sigma_n \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}$. *The notion of reduction* $\mathbf{s}_n$ *is strongly normalizing over the term language* IS_v.

Proof. Since the term language is IS_v, the reduction relations have the same *schema* as the reductions in $\mathbf{s}_v$. With minor modifications to account for the different definition of evaluation contexts, the proof that $\mathbf{s}_v$ is strongly normalizing also serves as the proof for this theorem. ∎

3.2 Call-by-name/Pass-by-reference

The Revised Report on Algol 60 [20] informally defines the semantics of call-by-name parameter passing with the *copy rule*. This rule roughly corresponds to Church's β axiom and generalizes the β_n relation from our call-by-name/pass-by-worth language. Whereas the β_n relation only applies to applications in which the formal parameter is *not* assigned, the copy rule applies to all procedure applications. For procedures in which the formal parameter is assigned, the procedure argument should evaluate to a variable so that assignments to the parameter become assignments to this variable. Since assignments to the parameter affect the argument, the copy rule defines a pass-by-reference semantics.

Syntax

Because the copy rule substitutes expressions for free variables, the language must permit expressions to occur in any position where variables are allowed. In particular, we extend the syntax of IS_n to allow expressions in the variable position of **set!** expressions: (**set!** e e) replaces (**set!** x e) in the definition of the term language. The rest of IS_n remains the same with $\lambda x.e$ representing a call-by-name/pass-by-reference procedure.

Semantics

The extension of the language syntax for **set!** expressions requires an extension to the call-by-name evaluation contexts to permit evaluation of expressions in the variable position of a **set!** expression:

$$E_n ::= \ldots \mid (\textbf{set!}\ E_n\ e)$$

The program transformation function, $\triangleright_{nr}$, for the call-by-name/pass-by-reference language uses only the copy rule for defining the behavior of *all* procedure applications:

$$\rho\theta.E_n[(\lambda x.e)e'] \triangleright \rho\theta.E_n[e[x \leftarrow e']] \qquad (E.\beta)$$

The transformation function should not dereference a variable when it occurs in the variable position of an assignment, so we add a restriction to the dereference rule D_n:

$$\rho\theta \cup \{(x,e)\}.E_n[x] \triangleright \rho\theta \cup \{(x,e)\}.E_n[e], \text{ if } E_n \neq E'_n[(\textbf{set!}\ [\,]\ e')] \qquad (E.D'_n)$$

The complete transformation function is the union of these and other rules:

$$\triangleright_{nr} = E.\delta \cup E.\beta \cup E.D'_n \cup \sigma_n \cup \rho_\cup$$

The situation is analogous to the call-by-value/pass-by-reference language, in which the partitioning of a program into an evaluation context and a redex is not unique. The added restriction on the $E.D'_n$ rule is sufficient to correct this problem and ensures that the rules define a function. The semantics $eval_{nr}$ is determined by the transitive closure of $\triangleright_{nr}$ and **gc** as in previous sections. Observational equivalence for this language is denoted $\simeq_{nr}$.

Calculus

The derivation of a calculus from the semantics follows the same course as the call-by-value/pass-by-reference language, but a naïve approach results in an unsound system. As with the call-by-value/pass-by-reference calculus, the dereferencing rule causes the problem. In particular, consider $E.D'_n$ as the variable dereference axiom. This axiom proves the following equivalence:

$$(\textbf{set!}\ \rho\{(x,c)\}.x\ e) = (\textbf{set!}\ \rho\{(x,c)\}.c\ e)$$

However, the meaning of the left expression is the meaning of e while the meaning of the right is undefined. Fortunately, as in the call-by-value case, this problem only occurs when the dereference of the variable x occurs within the empty context. Adding the appropriate condition to the dereference axiom solves the problem:

$$\begin{array}{rcl} \rho\theta \cup \{(x,e)\}.E_n[x] & = & \rho\theta \cup \{(x,e)\}.E_n[e] \\ & & \text{if } E_n \neq E'_n[(\textbf{set!}\ [\ ]\ e')] \text{ and } E_n \neq [\ ] \end{array} \qquad (D'_n)$$

The set of axioms for the λ_n-R-calculus is:

$$\mathbf{nr} = \delta \cup \beta \cup D'_n \cup \sigma_n \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}.$$

The theory $\boldsymbol{\lambda}_n$-$\mathbf{R}$ is constructed in the usual way; we write $\boldsymbol{\lambda}_n$-$\mathbf{R} \vdash e_1 = e_2$ if $e_1 =_{nr} e_2$.

The call-by-name/pass-by-reference and call-by-value/pass-by-reference languages have several similarities. In both languages the redexes do not specify a unique partition of programs into evaluation context and redex. As a result, both require a modified dereference rule. Similarly, the dereferencing axioms in both calculi require an added condition to guarantee soundness. Not surprisingly, as with the λ_v-R-calculus, the λ_n-R-calculus only *weakly* corresponds to the semantics. However, this weak correspondence is even weaker than for the call-by-value language. The reason is that while neither calculus can perform a variable dereference in an empty context, this kind of dereference in the call-by-value language would return an answer, whereas in the call-by-name language it does *not*. Thus, the λ_v-R-calculus can evaluate programs to expressions that are just one transformation step from an answer, but the λ_n-R-calculus requires an arbitrary number of these transformation steps throughout the derivation.[10]

[10] As with the call-by-value/pass-by-reference language, we can avoid the weak correspondence relationship by redefining the semantics to return expressions of the form $\rho\theta.x$ as *answers*. Again, this notion of call-by-name/pass-by-reference in which a reference to a thunk is an answer is not used by programming languages.

Theorem 3.2.1 (Very Weak Correspondence) *The λ_n-R-calculus* weakly *corresponds to the call-by-name/pass-by-reference semantics of IS_n, $eval_{nr}$. In particular,*

1. *$\boldsymbol{\lambda}_n$-**R** is weakly adequate:*

   (a) *if $eval_{nr}(e) = a$ then either*

   - *$\boldsymbol{\lambda}_n\text{-}\mathbf{R} \vdash e = a$, or*
   - *there exists an m such that for all n between 0 and m:*

$$\begin{array}{lll}\boldsymbol{\lambda}_n\text{-}\mathbf{R} & \vdash & e = \rho\theta_1 \cup \{(x_1, e_1')\}.x_1 \\ \boldsymbol{\lambda}_n\text{-}\mathbf{R} & \vdash & \rho\theta_n \cup \{(x_n, e_n')\}.e_n' = \rho\theta_{n+1} \cup \{(x_{n+1}, e_{n+1}')\}.x_{n+1} \\ \boldsymbol{\lambda}_n\text{-}\mathbf{R} & \vdash & \rho\theta_m \cup \{(x_m, e_m')\}.e_m' = a\end{array}$$

   (b) *if $\boldsymbol{\lambda}_n\text{-}\mathbf{R} \vdash e = a$, then $eval_{nr}(e)$ is defined; and*

2. *$\boldsymbol{\lambda}_n$-**R** is sound with respect to $\simeq_{nr}$:*

$$\boldsymbol{\lambda}_n\text{-}\mathbf{R} \vdash e_1 = e_2 \text{ implies } e_1 \simeq_{nr} e_2.$$

The imperative fragment, ρ_n', is the subtheory based upon the axioms in $\mathbf{nr} \setminus \beta$. Like the theory ρ_n, ρ_n' is decidable over the language that restricts ranges of ρ-sets to values.

Theorem 3.2.2 (Strong Normalization) *The notion of reduction* $\mathbf{s}_n' = \delta_c \cup D_n' \cup \sigma_n \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}$ *is strongly normalizing over the subset of IS_n in which the ranges of ρ-sets is restricted to values.*

Finally, the λ_n-R-calculus is a conservative extension of the λ-calculus.

Theorem 3.2.3 (Conservative Extension) $\boldsymbol{\lambda} = \boldsymbol{\lambda}_n\text{-}\mathbf{R}|\Lambda$

Note: Call-by-need as an Optimization

The call-by-name evaluation strategy evaluates procedure arguments for every reference to the parameter. A variant strategy, *call-by-need,* attempts to eliminate redundant evaluations. Specifically, it evaluates an argument only for the first occurrence of a formal parameter in the procedure body. All other occurrences of the parameter receive the value obtained from the first evaluation.

In functional languages, the call-by-need evaluation is an *optimization* of call-by-name evaluation; they both produce the same results. In the presence of side-effects, however, the strategy differs from call-by-name because assignments within an argument are evaluated only once.

Using our imperative language as an *implementation* language, we can define the call-by-need semantics of *functional* languages in which the syntax is restricted to Λ. The rule for call-by-need procedure application is given as follows:

$$\rho\theta.E_n[(\lambda x.e)e'] \triangleright \rho\theta.E_n[\rho\{(x, (\mathtt{set!}\ x\ e'))\}.e],$$

The first deference to the variable x in the procedure body e receives the expression (set! x e'). This expression evaluates e' and assigns to x this value so that subsequent dereferences to x receive the *value* of e'. The argument e' is evaluated at most once.

Chapter 4

Combining Techniques

Each of the languages we have considered uses one parameter passing technique exclusively. In reality, many programming languages permit different parameter passing techniques for different parameters. For example, Pascal allows both *value* parameters, which correspond to call-by-value/pass-by-worth, and *variable* parameters, which correspond to call-by-value/pass-by-reference. Algol 60 uses two different evaluation strategies; it has *name* parameters, which correspond to call-by-name/pass-by-reference and *value* parameters, which correspond to call-by-value/pass-by-worth.

The languages and equational systems presented thus far do not deal with multiple parameter passing techniques. However, it is possible to combine two or more languages to produce a language that has multiple parameter passing techniques. In this chapter, we briefly describe some of the requirements for combining two languages and for defining equational calculi for these languages.

Syntax

If a language incorporates two different parameter passing techniques then there must be an indication of which one to use for each procedure application. To accomplish this the language can either define an application operator for each technique, or define a procedure abstraction for each technique. Most languages adopt the second option and specify a different type of binding construct for each technique. For example, an abstraction $\lambda_{vw}x.e$ may specify a call-by-value/pass-by-worth procedure while $\lambda_{nr}x.e$ specifies a call-by-name/pass-by-reference procedure.

Semantics

The evaluation contexts determine the evaluation strategy. We defined call-by-value contexts and call-by-name contexts for the eager and delayed evaluation strategies, respectively. If a language such as Algol employs both call-by-value and call-by-name evaluation, then neither type of context is sufficient to specify the semantics. A new type of context that specifies call-by-name for some procedures and call-by-value for others is necessary:

$$\begin{array}{rcl} E & ::= & [\,] \mid (S\ E) \mid (E\ e) \mid \ldots \\ S & ::= & f \mid \lambda_v x.e \mid \ldots \end{array}$$

In this specification, S contains the strict functions and call-by-value procedures.

One problem that occurs when defining the semantics of combined languages is the need to specify the set of assigned variables, AV. In a pass-by-worth language, the assigned variables are those that occur in a variable position of an assignment. In a pass-by-reference language, however, it is impossible to determine which variables are assigned because an argument to a pass-by-reference procedure may be assigned in the body of the procedure. For example, in the procedure $\lambda x.xy$, the variable y may be assigned if x is bound to a pass-by-reference procedure that assigns to its parameter. For this reason, when we combine pass-by-worth with pass-by-reference, we must abandon the notion of assigned variables and the two relations $E.\beta_v$ and $E.\beta_n$ that rely upon that definition.[11]

As an example, consider a call-by-value language like Pascal that has both pass-by-worth and pass-by-reference parameters. The syntax is IS_v with λ replaced by λ_w for pass-by-worth and λ_r for pass-by-reference parameter passing techniques:

$$v \ ::= \ c \mid \lambda_w x.e \mid \lambda_r x.e$$

The program transformation function is the union of $\rhd_{vw}$ and $\rhd_{vr}$ specified in Figure 4.1. Notice that the rule $E.D'_v$ is equivalent to D_v if there are no pass-by-reference procedures in the language. By removing one type of procedure from the language, we can obtain a semantics for the other.

A more complicated example is Algol 60, which has both call-by-name and call-by-value evaluation strategies. Figure 4.2 defines the semantics of a language with both call-by-value/pass-by-worth procedures, $(\lambda_{vw} x.e)$ and call-by-name/pass-by-reference procedures, $(\lambda_{nr} x.e)$.

Calculus

The calculus for a combined language is a straightforward modification of the union of the two calculi. For the first example above, there are two procedure application

[11]One way to avoid this problem is to *designate* a set of assignable variables as in Felleisen et al [8, 9] instead of trying to determine which variables are assigned.

$$
\begin{array}{rcll}
\rho\theta.E_v[fv] & \rhd & \rho\theta.E_v[\delta(f,v)], \text{ if } \delta(f,v) \text{ defined.} & (E.\delta)\\
\rho\theta.E_v[(\lambda_w x.e)v] & \rhd & \rho\theta.E_v[\rho\{(x,v)\}.e] & (E.\beta_{v\sigma})\\
\rho\theta.E_v[(\lambda_r x.e)y] & \rhd & \rho\theta.E_v[e[x \leftarrow y]] & (E.\beta_r)\\
\rho\theta \cup \{(x,v)\}.E_v[x] & \rhd & \rho\theta \cup \{(x,v)\}.E_v[v], \text{ if } E_v \neq E'_v[(\lambda_r y.e)[\,]] & (E.D'_v)\\
\rho\theta \cup \{(x,u)\}.E_v[(\text{set! } x\ v)] & \rhd & \rho\theta \cup \{(x,v)\}.E_v[v] & (\sigma_v)\\
\rho\theta.E_v[\rho\theta'.e] & \rhd & \rho\theta \cup \theta'.E_v[e] & (\rho\cup)
\end{array}
$$

Figure 4.1 Call-by-value Language with Pass-by-worth and Pass-by-reference

Syntax:

$$
\begin{array}{rcll}
e & ::= & v \mid x \mid (e\ e) \mid (\text{set! } e\ e) \mid \rho\theta.e & (\textit{Expressions})\\
v & ::= & c \mid \lambda_{vw} x.e \mid \lambda_{nr} x.e & (\textit{Values})\\
\theta & ::= & \epsilon \mid \theta(x,e), \text{ where } (x,e') \notin \theta & (\rho\textit{-lists})
\end{array}
$$

Semantics:

$$
\begin{array}{rcl}
E & ::= & [\,] \mid (S\ E) \mid (E\ e) \mid (\text{set! } E\ e) \mid (\text{set! } x\ E)\\
S & ::= & f \mid \lambda_{vw} x.e
\end{array}
$$

$$
\begin{array}{rcll}
\rho\theta.E[fv] & \rhd & \rho\theta.E[\delta(f,v)], \text{ if } \delta(f,v) \text{ defined.} & (E.\delta)\\
\rho\theta.E[(\lambda_{vw} x.e)v] & \rhd & \rho\theta.E[\rho\{(x,v)\}.e] & (E.\beta_{v\sigma})\\
\rho\theta.E[(\lambda_{nr} x.e)e'] & \rhd & \rho\theta.E[e[x \leftarrow e']] & (E.\beta)\\
\rho\theta \cup \{(x,e)\}.E[x] & \rhd & \rho\theta \cup \{(x,e)\}.E[v], \text{ if } E \neq E'[(\lambda_{nr} y.e')[\,]] & (E.D'_n)\\
\rho\theta \cup \{(x,e)\}.E[(\text{set! } x\ v)] & \rhd & \rho\theta \cup \{(x,v)\}.E[v] & (\sigma_n)\\
\rho\theta.E[\rho\theta'.e] & \rhd & \rho\theta \cup \theta'.E[e] & (\rho\cup)
\end{array}
$$

Figure 4.2 Call-by-value/Pass-by-worth and Call-by-name/Pass-by-reference Language

axioms; one for pass-by-worth and one for pass-by-reference:

$$
\begin{array}{rcll}
(\lambda_w x.e)v & = & \rho\{(x,v)\}.e & (\beta_{v\sigma})\\
(\lambda_r x.e)y & = & e[x \leftarrow y] & (\beta_r)
\end{array}
$$

For the second example, there are also two axioms for the different parameter-passing techniques:

$$(\lambda_{vw}x.e)v = \rho\{(x,v)\}.e \qquad (\beta_{v\sigma})$$
$$(\lambda_{nr}x.e)e' = e[x \leftarrow e'] \qquad (\beta)$$

In both of these cases, the modified dereference axioms D'_v and D'_n are required to ensure soundness of the calculi. As with the pass-by-reference calculi, these axioms prohibit an exact correspondence between the calculi and the semantics. Thus, these calculi only satisfy *weak* correspondence theorems. On the other hand, the imperative fragments of the combined calculi are essentially the same as the corresponding fragments of the pass-by-reference calculi. Thus, these calculi satisfy strong normalization theorems.

Chapter 5

Conclusions

In this final chapter we give concluding remarks for the thesis. In the first section we give two examples of reasoning with the parameter-passing calculi. The second section discusses related work and the final section summarizes our work.

5.1 Examples

Before we provide the example programs, a few explanations of the presentation language are in order.

- The expression $\lambda xy.e$ abbreviates the *curried* procedure $\lambda x.\lambda y.e$. Similarly, the corresponding application of a curried procedure, $(M\ N\ L)$, abbreviates $((M\ N)\ L)$.
- We assume the language contains a sequence expression $\langle e_1; e_2\rangle$ that evaluates the left subexpressions followed by the right. The semantics of sequencing was defined for the call-by-value-result language in Section 2.3 and can easily be added to the other languages.
- The set of constants *BConsts*, includes the integers and appropriate functions on integers. Specifically, the primitive function + performs addition, and the functions n+ adds n to a number:

$$\begin{aligned} \delta(+, n) &= \mathsf{n+} \\ \delta(\mathsf{n+}, m) &= n + m \end{aligned}$$

For the first example, consider the IS_v program P in Figure 5.1 that applies a procedure to three arguments and returns the value of the variable a. The program P_s is the same program written in Scheme syntax. As expected, the behavior of the program depends upon the parameter-passing technique used to evaluate it. In fact, P returns a different answer for each technique and hence, it can determine which type of parameter-passing its implementation language uses.

$P \overset{df}{\equiv} \rho\{(a,\text{-}1),(b,0),(p,(\lambda xyz.(\textbf{set!}\ a\ (+\ y\ (\textbf{set!}\ x\ y)\ a))))\}.$
$\quad\langle(p\ a\ \langle(\textbf{set!}\ b\ (1{+}\ b));b\rangle\ a);a\rangle$

$P_s \overset{df}{\equiv}$
```
(letrec ([a -1] [b 0] [p (lambda (x y z) (set! a (+ y (set! x y) a)))])
  (begin
    (p a (begin (set! b (1+ b)) b) a)
    a))
```

Figure 5.1 Example Program P

Proposition 5.1.1 *The program P returns a different answer for each parameter-passing technique. Specifically,*

$$\begin{aligned} eval_{vw}(P) &= 1 \\ eval_{vr}(P) &= 3 \\ eval_{cc}(P) &= -1 \end{aligned} \qquad \begin{aligned} eval_{nw}(P) &= 2 \\ eval_{nr}(P) &= 5 \end{aligned}$$

Because of the adequacy requirement, the calculi can also evaluate the program to different values. However, since some calculi are not completely adequate, they do not give the answer given by the semantics.

Proposition 5.1.2 *The parameter-passing calculi prove the following equivalences for the program P:*

$$\begin{aligned} \lambda_v\text{-W} &\vdash P = 1 \\ \lambda_v\text{-R} &\vdash P = \rho\{(a,3)\}.a \\ \lambda_v\text{-VR} &\vdash P = \rho\{(a,-1)\}.a \end{aligned} \qquad \begin{aligned} \lambda_n\text{-W} &\vdash P = 2 \\ \lambda_n\text{-R} &\vdash P = \rho\{(a,5)\}.a \end{aligned}$$

For a second example, consider the familiar procedure swap that exchanges the values associated with two variables:

$$\textsf{swap} \overset{df}{\equiv} \lambda xy.\rho\{(t,\mathsf{c})\}.\langle(\textbf{set!}\ t\ x);(\textbf{set!}\ x\ y);(\textbf{set!}\ y\ t)\rangle$$

To verify that (swap a b) actually swaps the values associated with a and b, we must surround the expression with a ρ-expression that has bindings for the variables a and b. In particular, we wish to show the following equivalence:

$$\rho\theta \cup \{(a,\mathsf{a}_0),(b,\mathsf{b}_0)\}.E[\textsf{swap}\ a\ b] = \rho\theta \cup \{(a,\mathsf{b}_0),(b,\mathsf{a}_0)\}.E[\mathsf{a}_0]$$

where a_0 and b_0 are values.

Proposition 5.1.3 *Let* $e \overset{df}{\equiv} \rho\theta \cup \{(a, \mathsf{a}_0), (b, \mathsf{b}_0)\}.E[\mathsf{swap}\ a\ b]$. *Then we have the following:*

$$\begin{array}{rcl}
\boldsymbol{\lambda}_v\textbf{-W} & \vdash & e = \rho\theta \cup \{(a, \mathsf{a}_0), (b, \mathsf{b}_0)\}.E[\mathsf{a}_0] \\
\boldsymbol{\lambda}_v\textbf{-R} & \vdash & e = \rho\theta \cup \{(a, \mathsf{b}_0), (b, \mathsf{a}_0)\}.E[\mathsf{a}_0] \\
\boldsymbol{\lambda}_v\textbf{-VR} & \vdash & e = \rho\theta \cup \{(a, \mathsf{b}_0), (b, \mathsf{a}_0)\}.E[\mathsf{a}_0] \\
\boldsymbol{\lambda}_n\textbf{-W} & \vdash & e = \rho\theta \cup \{(a, \mathsf{a}_0), (b, \mathsf{b}_0)\}.E[\mathsf{a}_0] \\
\boldsymbol{\lambda}_n\textbf{-R} & \vdash & e = \rho\theta \cup \{(a, \mathsf{b}_0), (b, \mathsf{a}_0)\}.E[\mathsf{a}_0]
\end{array}$$

For the call-by-name/pass-by-reference calculus, this proposition does not hold if a_0 and b_0 are not values. The proof in Figure 5.2 using the λ_v-R-calculus verifies that swap works as expected using call-by-value/pass-by-reference parameter-passing.

5.2 Related Work

Our equational systems were motivated by the work of Plotkin [22], who studied equational reasoning systems for call-by-name and call-by-value in functional languages, and Felleisen et al [8, 9] who developed calculi for call-by-value *imperative* languages. Demers and Donahue [4] give an equational logic for reasoning about Russell, a higher-order language that uses call-by-value parameter-passing and has memory objects similar to reference cells. The equational theory contains several dozens axioms for which they present no formal results. Mason and Talcott [18] present a deduction system for reasoning about first-order Lisp programs with side-effects, a language similar to the imperative fragment of our reference cell language. Their logic is complete for the recursion-free fragment of Lisp with side-effects.

In the area of denotation semantics, several authors [11, 25] have given denotational descriptions of the different parameter-passing techniques. Such descriptions specify precise mathematical definitions of parameter-passing. They do not provide a canonical set of equations nor other axiomatic theories for reasoning about programs.

Finally, a number of researchers have studied parameter-passing in the context of Hoare-like axiomatic semantics [5, 12, 14]. These systems are generally for first-order subsets of Pascal and have a number of restrictions on procedure calls, such as aliasing. Cartwright and Oppen [2] overcome the aliasing restriction, but still do not allow procedures as arguments. Olderog [21] eliminates the restrictions on procedures, but his Hoare-like calculi explicitly require operational specifications of procedure calls using the copy rule. The copy rule semantics is based on a mixed formalization that uses denotational as well as operational elements.

I.

$$
\begin{aligned}
\lambda_v\text{-}\mathbf{R} \vdash\ & (\mathbf{swap}\ a\ b) \\
=\ & ((\lambda xy.\rho\{(t,\mathrm{c})\}.\langle(\mathbf{set!}\ t\ x);(\mathbf{set!}\ x\ y);(\mathbf{set!}\ y\ t)\rangle)\ a\ b) \\
=\ & \rho\{(t,\mathrm{c})\}.\langle(\mathbf{set!}\ t\ a);(\mathbf{set!}\ a\ b);(\mathbf{set!}\ b\ t)\rangle && (\beta_r)
\end{aligned}
$$

II.

$$
\begin{aligned}
\lambda_v\text{-}\mathbf{R} \vdash\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0})\}.E[(\mathbf{swap}\ a\ b)] \\
=\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0})\}.E[\rho\{(t,\mathrm{c})\}.\langle(\mathbf{set!}\ t\ a);(\mathbf{set!}\ a\ b);(\mathbf{set!}\ b\ t)\rangle] && (\mathrm{I.}) \\
=\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0}),(t,\mathrm{c})\}.E[\langle(\mathbf{set!}\ t\ a);(\mathbf{set!}\ a\ b);(\mathbf{set!}\ b\ t)\rangle] && (\rho_\cup) \\
=\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0}),(t,\mathrm{c})\}.E[\langle(\mathbf{set!}\ t\ \mathrm{a_0});(\mathbf{set!}\ a\ b);(\mathbf{set!}\ b\ t)\rangle] && (D'_v) \\
=\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0}),(t,\mathrm{a_0})\}.E[\langle\mathrm{a_0};(\mathbf{set!}\ a\ b);(\mathbf{set!}\ b\ t)\rangle] && (\sigma_v) \\
=\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0}),(t,\mathrm{a_0})\}.E[\langle(\mathbf{set!}\ a\ b);(\mathbf{set!}\ b\ t)\rangle] && (seq) \\
=\ & \rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0}),(t,\mathrm{a_0})\}.E[\langle(\mathbf{set!}\ a\ \mathrm{b_0});(\mathbf{set!}\ b\ t)\rangle] && (D'_v) \\
=\ & \rho\{(a,\mathrm{b_0}),(b,\mathrm{b_0}),(t,\mathrm{a_0})\}.E[\langle\mathrm{b_0};(\mathbf{set!}\ b\ t)\rangle] && (\sigma_v) \\
=\ & \rho\{(a,\mathrm{b_0}),(b,\mathrm{b_0}),(t,\mathrm{a_0})\}.E[(\mathbf{set!}\ b\ t)] && (seq) \\
=\ & \rho\{(a,\mathrm{b_0}),(b,\mathrm{b_0}),(t,\mathrm{a_0})\}.E[(\mathbf{set!}\ b\ \mathrm{a_0})] && (D'_v) \\
=\ & \rho\{(a,\mathrm{b_0}),(b,\mathrm{a_0}),(t,\mathrm{a_0})\}.E[\mathrm{a_0}] && (\sigma_v) \\
=\ & \rho\{(a,\mathrm{b_0}),(b,\mathrm{a_0})\}.E[\mathrm{a_0}] && (gc)
\end{aligned}
$$

III.

$$
\begin{aligned}
\lambda_v\text{-}\mathbf{R} \vdash\ & \rho\theta\cup\{(a,\mathrm{a_0}),(b,\mathrm{b_0})\}.E[(\mathbf{swap}\ a\ b)] \\
=\ & \rho\theta.\rho\{(a,\mathrm{a_0}),(b,\mathrm{b_0})\}.E[(\mathbf{swap}\ a\ b)] && (\rho_\cup) \\
=\ & \rho\theta.\rho\{(a,\mathrm{b_0}),(b,\mathrm{a_0})\}.E[\mathrm{a_0}] && (\mathrm{II.}) \\
=\ & \rho\theta\cup\{(a,\mathrm{b_0}),(b,\mathrm{a_0})\}.E[\mathrm{a_0}] && (\rho_\cup)
\end{aligned}
$$

Figure 5.2 Behavior of Swap in Call-by-value/Pass-by-reference

5.3 Summary

We have presented an operational semantics and an appropriate calculus for several common parameter-passing techniques. In most cases, the calculi are relatively simple equational theories. The exceptions are the pass-by-reference techniques in which the calculi lack the ability to evaluate programs completely. The correspondence between the calculi and the semantics in these cases is weaker than in the other cases. Furthermore, the imperative fragments of most calculi are strongly normalizing. We believe this work is a basis for the development of stronger, sequent-based proof systems with induction principles in the spirit of Mason and Talcott [18].

Evaluation Strategy	Binding Strategy	Corres-pondence	Beta Axiom			Strong Normalization
				Param	Arg	
call-by-value	worth	exact	β_v	non-assign	value	ρ_v
	reference	weak	β_r	any	variable	ρ'_v; β_r
	value-result	weak	β_c	–	–	$\rho'_v \cup seq$
	reference cell	exact	β_v	any	value	ρ_b
call-by-name	worth	exact	β_n	non-assign	any	restricted ρ_n
	reference	very weak	β	any	any	restricted ρ'_n

Table 5.1 Summary

Table 5.1 summarizes our results. The *Correspondence* column refers to how closely a calculus corresponds to the semantics. The necessary modifications to the familiar β axiom are shown under the *Beta Axiom* heading. The *Param* and *Arg* columns show the restrictions on the formal parameter and on the argument in procedure applications. The entry "non-assign" indicates that the rule only applies to applications in which the formal parameter is not assigned in the procedure body. The last column indicates which fragments are strongly normalizing.

A comparison of the calculi reveals some aspects of the relative semantic complexity of the parameter-passing techniques. In particular, this study verifies the folklore that the Algol call-by-name/pass-by-reference parameter-passing technique satisfies a full β axiom. Unfortunately, this comes at the expense of a weak correspondence theorem and complicated rules in the imperative fragment. The simplest and most appealing axiom system appears to be the one for a call-by-value/pass-by-worth language with reference cells as first-class values. It is a simple extension of the λ_v-calculus, satisfies a strong correspondence theorem, and has a large decidable fragment. Furthermore, reference cells in the call-by-value/pass-by-worth language provide some of the capabilities of call-by-value/pass-by-reference.

Appendix A

Proofs of Main Theorems

In this appendix, we provide outlines for the proofs of the Correspondence and Strong Normalization theorems in this thesis. Since the theorems all have the same basic structure, we present the complete proofs for the first theorems and then make observations about the changes required for the other theorems.

The proofs of the theorems in this thesis rely upon the definition of *notions of reduction* based upon the axioms of the calculi. These reduction systems are defined as follows.

Definition A.0.1. (*Notion of Reduction*) A notion of reduction **r** for a calculus is the set of axioms **r** of the calculus interpreted as reduction relations from left to right. The *one-step* **r**-*reduction* $\longrightarrow_r$ is the compatible closure of **r**:

$$e \longrightarrow_r e' \text{ if } e \equiv C[p],\ e' \equiv C[q] \text{ and } (p,q) \in \mathbf{r} \text{ for some } p,q \text{ and context } C.$$

The r-*reduction* $\twoheadrightarrow_r$ is the transitive, reflexive closure of $\longrightarrow_r$. Finally, r-*equality* $=_r$ is the least equivalence relation generated by $\twoheadrightarrow_r$.

A.1 Call-by-value/Pass-by-worth

Correspondence Theorem

Theorem 2.1.5 (Correspondence) *The λ_v-W-calculus corresponds to the call-by-value/pass-by-worth semantics of IS_v, $eval_{vw}$. In particular,*

1. $\boldsymbol{\lambda}_v$**-W** *is adequate:*

   (a) *if $eval_{vw}(e) = a$, then $\boldsymbol{\lambda}_v\text{-}\mathbf{W} \vdash e = a$,*

   (b) *if $\boldsymbol{\lambda}_v\text{-}\mathbf{W} \vdash e = a$, for answer a, then $eval_{vw}(e)$ is defined; and*

2. $\boldsymbol{\lambda}_v$**-W** *is sound with respect to $\simeq_{vw}$:*

$$\boldsymbol{\lambda}_v\text{-}\mathbf{W} \vdash e_1 = e_2 \textit{ implies } e_1 \simeq_{vw} e_2.$$

Proof. The proof relies upon the Church-Rosser and standardization properties of the calculus.

1. Adequacy.

   (a) Assume $eval_{vw}(e) = a$. We must show $\boldsymbol{\lambda}_v$-**W** $\vdash e = a$.

   By definition of $eval_{vw}$, $\rho\varnothing.e \rhd^*_{vw} \rho\theta.v$ and $\rho\theta.v \twoheadrightarrow_{vw} a$ by the **gc** reduction. Since $\rhd^*_{vw} \subseteq \twoheadrightarrow_{vw}$, we have $\rho\varnothing.e \twoheadrightarrow_{vw} \rho\theta.v \twoheadrightarrow_{vw} a$. Also, by **gc**, $\rho\varnothing.e \longrightarrow_{vw} e$. Thus, $\boldsymbol{\lambda}_v$-**W** $\vdash e = a$.

   (b) Assume $\boldsymbol{\lambda}_v$-**W** $\vdash e = a$, for answer a. We show $eval_{vw}(e)$ is defined.

   By soundness (part 2 of the theorem), $e \simeq_{vw} a$. Thus, $eval_{vw}(e)$ is defined iff $eval_{vw}(a)$ is defined. Since $eval_{vw}(a)$ is clearly defined, $eval_{vw}(e)$ is defined.

2. Soundness.

   A. Assume $\boldsymbol{\lambda}_v$-**W** $\vdash e_1 = e_2$.

   B. Assume C is a program context for e_1 and e_2.

   C. Assume $eval_{vw}(C[e_1])$ is defined. Then,

   - $eval_{vw}(C[e_1]) = a$, where a is an answer;
   - $\boldsymbol{\lambda}_v$-**W** $\vdash C[e_1] = a$, by Adequacy, part (a);
   - $\boldsymbol{\lambda}_v$-**W** $\vdash C[e_2] = a$, by transitivity of $=_{vw}$;
   - $C[e_2] \twoheadrightarrow_{vw} a'$, where a' is an answer, by Church-Rosser Lemma;
   - $\rho\varnothing.C[e_2] \rhd^*_{vw} a''$, where a'' is an answer, by Lemma A.1.3;
   - $eval_{vw}(C[e_2])$ is defined, by definition of $eval_{vw}$.

   Therefore, $eval_{vw}(C[e_2])$ is defined iff $eval_{vw}(C[e_1])$ is defined.

   C. Assume $eval_{vw}(C[e_1]) = c$, for basic constant c. Then,

   - $\boldsymbol{\lambda}_v$-**W** $\vdash C[e_1] = c$, by Adequacy, part (a);
   - $\boldsymbol{\lambda}_v$-**W** $\vdash C[e_2] = c$, by transitivity of $=_{vw}$;
   - $C[e_2] \twoheadrightarrow_{vw} c$, by Church-Rosser Lemma;
   - $\rho\varnothing.C[e_2] \rhd^*_{vw} \rho\theta.c$, by Lemma A.1.3;
   - $eval_{vw}(C[e_2]) = c$, by definition of $eval_{vw}$.

Therefore, $eval_{vw}(C[e_1]) = c$ iff $eval_{vw}(C[e_2]) = c$.

Therefore, $e_1 \simeq_{vw} e_2$.

Therefore, $\boldsymbol{\lambda}_v$-**W** $\vdash e_1 = e_2$ implies $e_1 \simeq_{vw} e_2$.

To complete the proof, we need only prove Lemma A.1.3 and the Church-Rosser Lemma. But, before proving Lemma A.1.3, we first need a few auxilliary definitions. The notion of reduction **vw-** is **vw** without garbage collection: **vw-** = **vw** − **gc**. Let $\longrightarrow_{vw-}$ be the corresponding one-step reduction and $\twoheadrightarrow_{vw-}$ the reflexive, transitive closure of $\longrightarrow_{vw-}$. Similarly, for the notion of reduction **gc**, let $\longrightarrow_{gc}$ be the one-step **gc**-reduction relation and $\twoheadrightarrow_{gc}$ be the **gc**-reduction relation.

The standard reduction function and standard reduction sequences define a standard reduction on terms that does not use the garbage collection reductions.

Definition A.1.1. (**vw**-*Standard Reduction Function*) A term e *standard reduces* to e', $e \longmapsto_{vw-} e'$, if there exist terms $(p, q) \in$ **vw-** such that either

- $e \equiv p$ and $e' \equiv q$; or
- $e \equiv E[p]$, $e' \equiv E[q]$, and $(p, q) \in \delta \cup \beta_v \cup \beta_{v\sigma}$; or
- $e = \rho\theta.E[p]$, $e' = \rho\theta.E[q]$, and $(p, q) \in \delta \cup \beta_v \cup \beta_{v\sigma}$,

for some evaluation context E and ρ-set θ.

An important consequence of this definition is that if $e \longmapsto^*_{vw-} e'$, then either $\rho\emptyset.e \rhd^*_{vw} \rho\emptyset.e'$ or $\rho\emptyset.e \rhd^*_{vw} e'$.

Lemma A.1.3 shows the necessary connection between the reduction relation $\twoheadrightarrow_{vw}$ and the transformation function $\rhd^*_{vw}$. The proof uses the Standardization Lemma, which states that for all reductions that do not use the garbage collection rules, there is a corresponding SR-sequence. We prove the Standardization Lemma in a subsequent section.

Lemma A.1.3 *For a program e, if* $e \twoheadrightarrow_{vw} a$ *where a is an answer, then there exists an answer a' such that*

1. $\rho\emptyset.e \rhd^*_{vw} a'$, *and*
2. *if a is a basic constant, then* $a' \equiv \rho\theta.a$.

Definition A.1.2. (*Standard Reduction Sequences*)
The standard reduction sequences (SR-sequences) are defined as follows:

1. Every constant and variable is a standard reduction sequence. The empty ρ-set is a standard reduction sequence.
2. If $e_1, \ldots, e_n$ is a standard reduction sequence, then so is $\lambda x.e_1, \ldots, \lambda x.e_n$.
3. If $e_1, \ldots, e_n$ is a standard reduction sequence, then so is
$$(\text{set! } x\ e_1), \ldots, (\text{set! } x\ e_n).$$
4. If $v_1, \ldots, v_n$ and $\theta_1, \ldots, \theta_m$ are standard reduction sequences, then so is
$$\{(x, v_1)\} \cup \theta_1, \{(x, v_2)\} \cup \theta_1, \ldots, \{(x, v_n)\} \cup \theta_1, \{(x, v_n)\} \cup \theta_2, \ldots, \{(x, v_n)\} \cup \theta_m.$$
5. If $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ are standard reduction sequences, then so is
$$p_1 q_1, p_2 q_1, \ldots, p_n q_1, p_n q_2, \ldots, p_n q_m.$$
6. If $e_1, \ldots, e_n$ and $\theta_1, \ldots, \theta_m$ are standard reduction sequences, then so is
$$\rho\theta_1.e_1, \ldots, \rho\theta_1.e_n, \rho\theta_2.e_n, \ldots, \rho\theta_m.e_n.$$
7. If $e_1, \ldots, e_n$ is a standard reduction sequence and $e \longmapsto_{vw-} e_1$, then $e, e_1, \ldots, e_n$ is a standard reduction sequence.

Proof. Assume $e \twoheadrightarrow_{vw} a$, for answer a. By Lemma A.1.4, there exists an answer a_1 such that $e \twoheadrightarrow_{vw-} a_1$. By the Standardization Lemma, there exists a SR-sequence $e, \ldots, a_1$. By taking the first answer in the SR-sequence, we get $e \longmapsto^*_{vw-} a_2$. Since $e \longmapsto^*_{vw-} a_2$, then either $\rho\emptyset.e \rhd^*_{vw} a_2$ or $\rho\emptyset.e \rhd^*_{vw} \rho\emptyset.a_2$. For the first case, take $a' \equiv a_2$. For the second case, if a_2 is a value then take $a' \equiv \rho\emptyset.a_2$. Otherwise, if $a_2 \equiv \rho\theta.v$, then $\rho\emptyset.a_2 \rhd_{vw} a_2$ and take $a' \equiv a_2$.

For the second part, assume that a is a basic constant. By Lemma A.1.4 either $a_1 \equiv a$ or $a_1 \equiv \rho\theta.a$. Similarly, either $a_2 \equiv a$ or $a_2 \equiv \rho\theta'.a$. In the first case, $a' \equiv \rho\emptyset.a$ and in the second case $a' \equiv \rho\theta'.a$ which satisfy the condition that $a' \equiv \rho\theta.a$. $\blacksquare_{A.1.3}$

Lemma A.1.4 *If $e \twoheadrightarrow_{vw} a$, for answer a, there exists an answer a' such that*

1. *$e \twoheadrightarrow_{vw-} a'$,*

2. *if a is a basic constant c, then either $a' \equiv c$ or $a' \equiv \rho\theta.c$.*

Proof. Assume $e \twoheadrightarrow_{vw} a$. By the following two lemmas, which we state without proof, we have $e \twoheadrightarrow_{vw-} e' \twoheadrightarrow_{gc} a$. Furthermore, since $e' \twoheadrightarrow_{gc} a$, we have $e' \twoheadrightarrow_{vw-} a' \twoheadrightarrow_{gc} a$, for answer a'. Thus, $e \twoheadrightarrow_{vw-} e' \twoheadrightarrow_{vw-} a'$. The second part follows from the fact that $a' \twoheadrightarrow_{gc} a$.

Lemma A.1.5 *If $e \longrightarrow_{gc} e_1 \longrightarrow_{vw-} e_2$, then $e \longrightarrow_{vw-} e_1' \longrightarrow_{gc} e_2$.*

Lemma A.1.6 *If $e \twoheadrightarrow_{gc} a$, for answer a, then $e \twoheadrightarrow_{vw-} a' \twoheadrightarrow_{gc} a$, for some answer a'.*

$\blacksquare_{A.1.4}$

With the proof of the Church-Rosser Lemma and the Standardization Lemma in the following sections, the proof of the Correspondence Theorem is complete. $\blacksquare_{2.1.5}$

Church-Rosser

The first major lemma required by the proof of the Correspondence Theorem is the Church-Rosser Lemma:

Lemma A.1.7 (Church-Rosser) *The notion of reduction* **vw** *(call-by-value/pass-by-worth calculus) is Church-Rosser.*

Proof. We define two notions of reduction **vwa** and **vwb** such that **vw** = **vwa** ∪ **vwb** and show that both **vwa** and **vwb** are Church-Rosser. We then show that **vwa**-reduction *commutes* with **vwb**-reduction, and therefore, by the Hindley-Rosen Lemma [1:64], **vw** is Church-Rosser.

The notions of reduction **vwa** and **vwb** are defined as follows:

$$\begin{aligned} \textbf{vwa} &= \delta \cup \beta_v \cup \beta_\sigma \cup D_v \cup \sigma_v \cup \textbf{gc} \\ \textbf{vwb} &= \rho_\cup \cup \rho_{lift}. \end{aligned}$$

Lemma A.1.8 *The notion of reduction* **vwa** *is Church-Rosser.*

Proof. We use a method due to Tait and Martin-Löf. We define a *parallel* reduction relation $\underset{1a}{\twoheadrightarrow}$ such that its transitive, reflexive closure is $\twoheadrightarrow_{vwa}$ and show that $\underset{1a}{\twoheadrightarrow}$ satisifies the diamond lemma. Therefore, $\twoheadrightarrow_{vwa}$ satisfies the diamond lemma and **vwa** is Church-Rosser.

Definition A.1.9. (*Parallel Reduction I*) The parallel reduction relation $\underset{1a}{\twoheadrightarrow}$ for the call-by-value/pass-by-worth language is defined as follows. We assume the natural extension to contexts.

1. If $\theta \underset{1a}{\twoheadrightarrow} \theta'$, $e \underset{1a}{\twoheadrightarrow} e'$, $v \underset{1a}{\twoheadrightarrow} v'$ and $E \underset{1a}{\twoheadrightarrow} E'$, then

$$\begin{array}{llcl}
a) & fv & \underset{1a}{\twoheadrightarrow} & \delta(f,v), \text{ if } \delta(f,v) \text{ defined.} \\
b) & (\lambda x.e)v & \underset{1a}{\twoheadrightarrow} & e'[x \leftarrow v'], \text{ if } x \notin AV(e) \\
c) & (\lambda x.e)v & \underset{1a}{\twoheadrightarrow} & \rho\{x,v'\}.e', \text{ if } x \in AV(e) \\
d) & \rho\theta \cup \{(x,v)\}.E[x] & \underset{1a}{\twoheadrightarrow} & \rho\theta' \cup \{(x,v')\}.E'[v'] \\
e) & \rho\theta \cup \{(x,u)\}.E[(\text{set! } x\ v)] & \underset{1a}{\twoheadrightarrow} & \rho\theta' \cup \{(x,v')\}.E'[v'] \\
f) & \rho\theta_0 \cup \theta_1.e & \underset{1a}{\twoheadrightarrow} & \rho\theta_1'.e', \text{ if } \theta_0 \neq \emptyset \\
 & & & \text{and } FV(\rho\theta_1.\grave{e}) \cap Dom(\theta_0) = \emptyset \\
g) & \rho\emptyset.e & \underset{1a}{\twoheadrightarrow} & e'
\end{array}$$

2. For all terms e, $e \underset{1a}{\twoheadrightarrow} e$.

3. If $\theta \underset{1a}{\twoheadrightarrow} \theta'$, $e \underset{1a}{\twoheadrightarrow} e'$, $e_i \underset{1a}{\twoheadrightarrow} e_i'$ and $v_i \underset{1a}{\twoheadrightarrow} v_i'$, then

$$\begin{array}{llcl}
a) & \lambda x.e & \underset{1a}{\twoheadrightarrow} & \lambda x.e' \\
b) & (\text{set! } x\ e) & \underset{1a}{\twoheadrightarrow} & (\text{set! } x\ e') \\
c) & \rho\theta.e & \underset{1a}{\twoheadrightarrow} & \rho\theta'.e' \\
d) & e_1e_2 & \underset{1a}{\twoheadrightarrow} & e_1'e_2' \\
e) & \{(x_1,v_1),\ldots,(x_n,v_n)\} & \underset{1a}{\twoheadrightarrow} & \{(x_1,v_1'),\ldots,(x_n,v_n')\}
\end{array}$$

Assume $e \underset{1a}{\twoheadrightarrow} e_1$ and $e \underset{1a}{\twoheadrightarrow} e_2$. We use case analysis of the definition of $e \underset{1a}{\twoheadrightarrow} e_1$ and show for all possible cases of $e \underset{1a}{\twoheadrightarrow} e_2$ there exists a term e_3 that satisfies the diamond property.

1. We pick representative cases from group 1 of the definition of parallel reduction.

   a) $e \equiv (f\ v) \underset{1a}{\twoheadrightarrow} \delta(f,v) \equiv e_1$

   The only possibility for e_2 is the following:

   - $e_2 \equiv (f\ v_2)$, where $v \underset{1a}{\twoheadrightarrow} v_2$.

   By the restriction on δ, we have $\delta(f,v) = \delta(f,v_2)$.

   b) $e \equiv (\lambda x.M)v \underset{1a}{\twoheadrightarrow} M_1[x \leftarrow v_1] \equiv e_1$, where $x \notin AV(M)$, $M \underset{1a}{\twoheadrightarrow} M_1$ and $v \underset{1a}{\twoheadrightarrow} v_1$.

Two possibilities for e_2:

- $e_2 \equiv (\lambda x.M_2)v_2$.
- $e_2 \equiv M_2[x \leftarrow v_2]$.

By inductive hypothesis, there exist M_3 and v_3 such that $M_i \xrightarrow[1a]{}\!\!\!\!\to M_3$, $v_i \xrightarrow[1]{}\!\!\!\!\to v_3$. By Lemma A.1.15, $M_i[x \leftarrow v_i] \xrightarrow[1a]{}\!\!\!\!\to M_3[x \leftarrow v_3]$. Thus, take $e_3 \equiv M_3[x \leftarrow v_3]$.

d) $e \equiv \rho\theta \cup \{(x,v)\}.E[x] \xrightarrow[1a]{}\!\!\!\!\to \rho\theta_1 \cup \{(x,v_1)\}.E_1[v_1] \equiv e_1$.

There are several possibilities for e_2. We select two:

- $e_2 \equiv \rho\theta_2 \cup \{(x,v_2)\}.E_2[x]$
- $e_2 \equiv \rho\theta_2 \cup \{(x,v_2)\}.E_2[v_2]$,

where $\theta \xrightarrow[1a]{}\!\!\!\!\to \theta_2$, $v \xrightarrow[1a]{}\!\!\!\!\to v_2$, $E \xrightarrow[1a]{}\!\!\!\!\to E_2$.

By the inductive hypothesis, there exists θ_3, v_3, and E_3, such that $e_3 \equiv \rho\theta_3 \cup \{(x,v_3)\}.E_3[v_3]$ satisifies the diamond property.

f) $e \equiv \rho\theta \cup \theta'.M \xrightarrow[1a]{}\!\!\!\!\to \rho\theta_1.M_1 \equiv e_1$, where $Dom(\theta') \cap FV(\rho\theta.M) = \varnothing$.

- $e_2 \equiv \rho\theta_2 \cup \theta_2'.M_2$
- $e_2 \equiv \rho\theta_2.M_2$
- $e \equiv \rho\theta^a \cup \theta^b.M \xrightarrow[1a]{}\!\!\!\!\to \rho\theta_2^b.M_2$, where $\theta \cup \theta' \equiv \theta^a \cup \theta^b$ and $Dom(\theta^a) \cap FV(\rho\theta^b.M) = \varnothing$.

For the first two cases, there exists θ_3, θ_3', and M_3 such that $e_3 \equiv \rho\theta_3.M_3$ satifies the diamond property.

For third case, take $e_3 \equiv \rho\theta_c.M_3$, where $Dom(\theta_c) = Dom(\theta^b) \cap Dom(\theta)$.

g) $e \equiv \rho\varnothing.M \xrightarrow[1a]{}\!\!\!\!\to M_1 \equiv e_1$.

The two cases for e_2 are $\rho\varnothing.M_2$ and M_2. Taking $e_3 \equiv M_3$, where $M_i \xrightarrow[1a]{}\!\!\!\!\to M_3$ satisfies the diamond property.

2. $e \xrightarrow[1a]{}\!\!\!\!\to e \equiv e_1$.

If $e \xrightarrow[1a]{}\!\!\!\!\to e_2$ then we can take $e_3 \equiv e_2$ to satisfy the diamond property.

3. $e \equiv C[M] \xrightarrow[1a]{}\!\!\!\!\to C_1[M_1] \equiv e_1$, where $M \xrightarrow[1a]{}\!\!\!\!\to M_1$ and $C \xrightarrow[1a]{}\!\!\!\!\to C_1$.

Besides the cases we have already considered, there is one other case:

- $e_2 \equiv C_2[M_2]$, where $M \xrightarrow[1a]{}\!\!\!\!\to M_2$, and $C \xrightarrow[1a]{}\!\!\!\!\to C_2$.

By the inductive hypothesis (we assume that C is not empty), there exists M_3 such that $e_3 \equiv C_3[M_3]$.

∎A.1.8

Lemma A.1.10 *The notion of reduction* vwb *is Church-Rosser.*

Proof. We show that the one-step reduction $\longrightarrow_{vwb}$ satisfies the diamond property.

$\rho_\cup$) $e \equiv \rho\theta.E[\rho\theta'.M] \longrightarrow_{vwb} \rho\theta \cup \theta'.E[M] \equiv e_1$.

- $e_2 \equiv \rho\theta_2.E[\rho\theta'.M]$.
- $e_2 \equiv \rho\theta.N$, where $E[\rho\theta'.M] \longrightarrow_{vwb} N$.

For the first case, take $e_3 \equiv \rho\theta_2 \cup \theta'.E[M]$. For the second case, we use a case analysis on the reduction $E[\rho\theta'.M] \longrightarrow_{vwb} N$ as follows:

- $\rho\theta'.M \equiv \rho\theta'.F[\rho\theta''.L] \longrightarrow_{vwb} \rho\theta' \cup \theta''.F[L] \equiv N$
  Take $e_3 \equiv \rho\theta \cup \theta' \cup \theta''.E[F[L]]$.
- $E[\rho\theta'.M] \equiv F_a[F_b[\rho\theta'.M] \longrightarrow_{vwb} F_a[\rho\theta'.F_b[M]] \equiv N$
  Then $e_2 \longrightarrow_{vwb} e_1 \equiv \rho\theta \cup \theta'.E[M]$.

ρ_{lift}) $e \equiv E[\rho\theta.M] \longrightarrow_{vwb} \rho\theta.E[M] \equiv e_1$, where $E \not\equiv [\ ]$.

As in the previous case, there are a number of possibilities for e_2, such as:

- $e_2 \equiv E_2[\rho\theta.M]$, where $E \longrightarrow_{vwb} E_2$.
- $e_2 \equiv E[N]$, where $\rho\theta.M \longrightarrow_{vwb} N$.

A similar case analysis applies here.

∎A.1.10

Lemma A.1.11 *The* vwa*-reduction commutes with* vwb*-reduction.*

Proof. For this proof, let $\longrightarrow_{vwb}$ be the *reflexive closure* of the one-step reduction. By Barendregt [1:65], it is sufficient to show that if $e \xrightarrow[1a]{} \!\!\!\!\! \rightarrow e_1$ and $e \longrightarrow_{vwb} e_2$ there exists an e_3 such that $e_1 \twoheadrightarrow_{vwb} e_3$ and $e_2 \xrightarrow[1a]{} \!\!\!\!\! \rightarrow e_3$.

1. First consider cases in which $e \xrightarrow[1a]{} \!\!\!\!\! \rightarrow e_1$. We include only the interesting cases:

   a) $e \equiv fv \xrightarrow[1a]{} \!\!\!\!\! \rightarrow \delta(f, v) \equiv e_1$.

- $e_2 \equiv f v_2$, where $v \longrightarrow_{vwb} v_2$.

By the restriction on δ, $\delta(f, v) = \delta(f, v_2)$. Thus, $e_2 \longrightarrow_{vwa} e_1$.

b) $e \equiv (\lambda x.M)v \underset{1a}{\twoheadrightarrow} M_1[x \leftarrow v_1] \equiv e_1$.

- $e_2 \equiv (\lambda x.M)v_2$, where $v \longrightarrow_{vwb} v_2$.

By induction, there exists v_3 such that $v_1 \twoheadrightarrow_{vwb} v_3$ and $v_2 \underset{1a}{\twoheadrightarrow} v_3$. Thus, $e_1 \equiv M_1[x \leftarrow v_1] \twoheadrightarrow_{vwb} M_1[x \leftarrow v_3]$ and we take $e_3 \equiv M[x \leftarrow v_3]$.

d) $e \equiv \rho\theta \cup \{(x, v)\}.E[x] \underset{1a}{\twoheadrightarrow} \rho\theta_1 \cup \{(x, v_1)\}.E_1[v_1] \equiv e_1$.

- $e_2 \equiv \rho\theta \cup \{(x, v_2)\}.E[x]$.

This case is similar to β_v. By induction, there exists v_3, such that $v_2 \underset{1a}{\twoheadrightarrow} v_3$ and $v_1 \twoheadrightarrow_{vwb} v_3$. Thus, $\rho\theta_1 \cup \{(x, v_1)\}.E_1[v_1] \underset{1a}{\twoheadrightarrow} \rho\theta_1 \cup \{(x, v_3)\}.E_1[v_3]$ and we can take e_3 to be the latter expression.

2. Next, consider the cases in which $e \longrightarrow_{vwb} e_2$, because $(e, e_2) \in$ **vwb**:

ρ_{lift}) $e \equiv E[\rho\theta.M] \longrightarrow_{vwb} \rho\theta.E[M] \equiv e_2$.

There are several subcases for $e \underset{1a}{\twoheadrightarrow} e_1$:

- $e \equiv E[\rho\theta \cup \{(x, v)\}.F[x]]) \underset{1a}{\twoheadrightarrow} E_1[\rho\theta_1 \cup \{(x, v_1)\}.F_1[v_1]]) \equiv e_1$.
  Then, $e_2 \equiv \rho\theta \cup \{(x, v)\}.E[F[x]])$ and we can take
  $e_3 \equiv \rho\theta_1 \cup \{(x, v_1)\}.E_1[F_1[v_1]]$.
- $e \equiv E[\rho\theta \cup \{(x, u)\}.F[(\text{set! } x\ v)]] \underset{1a}{\twoheadrightarrow} E_1[\rho\theta_1 \cup \{(x, v_1)\}.F_1[v_1]] \equiv e_1$.
  Similarly.
- $e \equiv E[\rho\theta \cup \theta'.N] \underset{1a}{\twoheadrightarrow} E_1[\rho\theta'_1.N_1] \equiv e_1$, where $Dom(\theta) \cap FV(\rho\theta'.N) = \emptyset$.
  Then, $e_2 \equiv \rho\theta \cup \theta'.E[N]$. Since $Dom(\theta_1) \cap FV(\rho\theta'_1.E[N_1]) = \emptyset$, we have $e_3 \equiv \rho\theta'_1.E[N_1]$.
- $e \equiv E[\rho\emptyset.N] \underset{1a}{\twoheadrightarrow} E_1[N_1] \equiv e_1$.
  Then, $e_2 \equiv \rho\emptyset.E[N] \underset{1a}{\twoheadrightarrow} E_1[N_1] \equiv e_3$.

$\rho_\cup$) $e \equiv \rho\theta.E[\rho\theta'.M] \longrightarrow_{vwb} \rho\theta \cup \theta'.E[M] \equiv e_2$.

The same four subcases from ρ_{lift} apply here also.

3. Finally, the only remaining cases are those in which $e \equiv C[M] \underset{1a}{\twoheadrightarrow} C_1[M_1]$, by rule 3 of definition of $\underset{1a}{\twoheadrightarrow}$, and $e \equiv C'[N] \longrightarrow_{vwb} C'[N_2]$, because $(N, N_2) \in$ **vwb**, where C and C' are non-empty contexts.

For these cases, we use the inductive hypothesis to get the result.

$\blacksquare_{A.1.11}$

The final lemma for the Chuch-Rosser proof is Lemma A.1.15 for showing properties about substitution and parallel reduction. Because it is also useful for the standardization proof, we prove an extended version of this lemma in the next section. This completes the proof of the Church-Rosser Lemma. $\blacksquare_{A.1.7}$

Standardization

The second major lemma required by the Correspondence Theorem is the Standardization Lemma:

Lemma A.1.12 (Standardization) $e \twoheadrightarrow_{vw-} e'$ *iff there exists a Standard Reduction Sequence* $e, \ldots, e'$.

Proof. We use the proof method of Plotkin [22]. The implication from right to left is straightforward because $\longrightarrow_{vw-}$ can simulate $\longmapsto_{vw-}$.

For the implication from left to right, we define another *parallel* reduction relation $\underset{1}{\twoheadrightarrow}$ similar to the one in the previous section, except that its transitive, reflexive closure is $\twoheadrightarrow_{vw-}$ not $\twoheadrightarrow_{vwa}$. If $e \twoheadrightarrow_{vw-} e'$ then there must be a sequence $e \underset{1}{\twoheadrightarrow} e_1 \underset{1}{\twoheadrightarrow} e_2 \underset{1}{\twoheadrightarrow} \cdots \underset{1}{\twoheadrightarrow} e'$. The proof is by induction on the length of the sequence. If the length is 0, i.e., it contains just e, then the result is a SR-sequence. By Lemma A.1.14, if there is a SR-sequence $e_1, \ldots, e'$ and $e \underset{1}{\twoheadrightarrow} e_1$, then there is a SR-sequence $e, \ldots, e'$.

The parallel reduction relation $\underset{1}{\twoheadrightarrow}$ is defined in Definition A.1.13.

Lemma A.1.14 *If* $e \underset{1}{\twoheadrightarrow} e_1$ *and* $e_1, \ldots, e_n$ *is a* SR*-sequence, then there exists a* SR*-sequence* $e, \ldots, e_n$.

Proof. By lexicographic induction on the structure $\langle n, s_{e \underset{1}{\twoheadrightarrow} e_1}, e\rangle$ and case analysis of $e \underset{1}{\twoheadrightarrow} e_1$.

1. For group 1, we observe that there exists a term e' such that $e \longmapsto_{vw-} e' \underset{1}{\twoheadrightarrow} e_1$. In these cases we show that the size of the reduction $e' \underset{1}{\twoheadrightarrow} e_1$ is less than the size of the reduction $e \underset{1}{\twoheadrightarrow} e_1$, (i.e. $s_{e' \underset{1}{\twoheadrightarrow} e_1} < s_{e \underset{1}{\twoheadrightarrow} e_1}$) and then employ the inductive hypothesis on the smaller reduction. That is, $e' \underset{1}{\twoheadrightarrow} e_1$ and $e_1, \ldots, e_n$ is a SR-sequence implies there is a SR-sequence $e', \ldots, e_n$. Since $e \longmapsto_{vw-} e'$ we have $e, e', \ldots, e_n$ is a SR-sequence by the definition of a SR-sequence.

   We give two prototypical examples:

Definition A.1.13. (*Parallel Reduction II*) The parallel reduction relation $\underset{1}{\twoheadrightarrow}$ for the call-by-value/pass-by-worth language is defined as follows, where $s_{e \underset{1}{\twoheadrightarrow} e'}$ is the *size* of a parallel reduction $e \underset{1}{\twoheadrightarrow} e'$ and $n(x,e)$ is the number of free occurrences of x in e. We assume the natural extension to contexts.

1. If $\theta \underset{1}{\twoheadrightarrow} \theta'$, $e \underset{1}{\twoheadrightarrow} e'$, $v \underset{1}{\twoheadrightarrow} v'$ and $E \underset{1}{\twoheadrightarrow} E'$, then

$$\begin{array}{lrcl}
a) & fv & \underset{1}{\twoheadrightarrow} & \delta(f,v), \text{ if } \delta(f,v) \text{ defined.} \\
 & s & = & 1 \\
b) & (\lambda x.e)v & \underset{1}{\twoheadrightarrow} & e'[x \leftarrow v'], \text{ if } x \notin AV(e) \\
 & s & = & s_{e \underset{1}{\twoheadrightarrow} e'} + n(x,e') s_{v \underset{1}{\twoheadrightarrow} v'} + 1 \\
c) & (\lambda x.e)v & \underset{1}{\twoheadrightarrow} & \rho\{x,v'\}.e', \text{ if } x \in AV(e) \\
 & s & = & s_{e \underset{1}{\twoheadrightarrow} e'} + s_{v \underset{1}{\twoheadrightarrow} v'} + 1 \\
d) & \rho\theta \cup \{(x,v)\}.E[x] & \underset{1}{\twoheadrightarrow} & \rho\theta' \cup \{(x,v')\}.E'[v'] \\
 & s & = & s_{\theta \underset{1}{\twoheadrightarrow} \theta'} + 2 s_{v \underset{1}{\twoheadrightarrow} v'} + s_{E \underset{1}{\twoheadrightarrow} E'} + 1 \\
e) & \rho\theta \cup \{(x,u)\}.E[(\textsf{set!}\ x\ v)] & \underset{1}{\twoheadrightarrow} & \rho\theta' \cup \{(x,v')\}.E'[v'] \\
 & s & = & s_{\theta \underset{1}{\twoheadrightarrow} \theta'} + 2 s_{v \underset{1}{\twoheadrightarrow} v'} + s_{E \underset{1}{\twoheadrightarrow} E'} + 1 \\
f) & \rho\theta_1.E[\rho\theta_2.e] & \underset{1}{\twoheadrightarrow} & \rho\theta_1' \cup \theta_2'.E'[e'] \\
 & s & = & s_{\theta_1 \underset{1}{\twoheadrightarrow} \theta_1'} + s_{\theta_2 \underset{1}{\twoheadrightarrow} \theta_2'} + \\
 & & & s_{E \underset{1}{\twoheadrightarrow} E'} + s_{e \underset{1}{\twoheadrightarrow} e'} + 1 \\
g) & E[\rho\theta.e] & \underset{1}{\twoheadrightarrow} & \rho\theta'.E'[e'], \text{ if } E \neq [\,] \\
 & s & = & s_{\theta \underset{1}{\twoheadrightarrow} \theta'} + s_{E \underset{1}{\twoheadrightarrow} E'} + s_{e \underset{1}{\twoheadrightarrow} e'} + 1
\end{array}$$

2. For all terms e, $e \underset{1}{\twoheadrightarrow} e$, $s_{e \underset{1}{\twoheadrightarrow} e'} = 0$.

3. If $\theta \underset{1}{\twoheadrightarrow} \theta'$, $e \underset{1}{\twoheadrightarrow} e'$, $e_i \underset{1}{\twoheadrightarrow} e_i'$ and $v_i \underset{1}{\twoheadrightarrow} v_i'$, then

$$\begin{array}{lrcl}
a) & \lambda x.e & \underset{1}{\twoheadrightarrow} & \lambda x.e' \\
 & s & = & s_{e \underset{1}{\twoheadrightarrow} e'} \\
b) & (\textsf{set!}\ x\ e) & \underset{1}{\twoheadrightarrow} & (\textsf{set!}\ x\ e') \\
 & s & = & s_{e \underset{1}{\twoheadrightarrow} e'} \\
c) & \rho\theta.e & \underset{1}{\twoheadrightarrow} & \rho\theta'.e' \\
 & s & = & s_{e \underset{1}{\twoheadrightarrow} e'} + s_{\theta \underset{1}{\twoheadrightarrow} \theta'} \\
d) & e_1 e_2 & \underset{1}{\twoheadrightarrow} & e_1' e_2' \\
 & s & = & s_{e_1 \underset{1}{\twoheadrightarrow} e_1'} + s_{e_2 \underset{1}{\twoheadrightarrow} e_2'} \\
e) & \{(x_1,v_1),\ldots,(x_n,v_n)\} & \underset{1}{\twoheadrightarrow} & \{(x_1,v_1'),\ldots,(x_n,v_n')\} \\
 & s & = & \sum_{i=1}^{n} s_{v_i \underset{1}{\twoheadrightarrow} v_i'}
\end{array}$$

b) $e \equiv (\lambda x.M)v \xrightarrow[1]{} M_1[x \leftarrow v_1] \equiv e_1$, where $M \xrightarrow[1]{} M_1$ and $v \xrightarrow[1]{} v_1$.

$$e \longmapsto_{vw-} M[x \leftarrow v] \xrightarrow[1]{} M_1[x \leftarrow v_1] \equiv e_1$$

Using Lemma A.1.15 we have:

$$s_{M[x\leftarrow v]\xrightarrow[1]{} M_1[x\leftarrow v_1]} < s_{(\lambda x.M)v \xrightarrow[1]{} M_1[x\leftarrow v_1]}$$

d) $e \equiv \rho\theta \cup \{(x,v)\}.E[x] \xrightarrow[1]{} \rho\theta_1 \cup \{(x,v_1)\}.E_1[v_1] \equiv e_1$,
where $\theta \xrightarrow[1]{} \theta_1$, $v \xrightarrow[1]{} v_1$ and $E \xrightarrow[1]{} E_1$.

$$e \longmapsto_{vw-} \rho\theta \cup \{(x,v)\}.E[v] \xrightarrow[1]{} \rho\theta_1 \cup \{(x,v_1)\}.E_1[v_1] \equiv e_1$$

$$\begin{aligned} s_{\rho\theta\cup\{(x,v)\}.E[v]\xrightarrow[1]{}\rho\theta_1\cup\{(x,v_1)\}.E_1[v_1]} &= s_{\theta\xrightarrow[1]{}\theta_1} + s_{v\xrightarrow[1]{}v_1} + s_{E\xrightarrow[1]{}E_1} + s_{v\xrightarrow[1]{}v_1} \\ &< s_{\theta\xrightarrow[1]{}\theta_1} + 2s_{v\xrightarrow[1]{}v_1} + s_{E\xrightarrow[1]{}E_1} + 1 \\ &= s_{\rho\theta\cup\{(x,v)\}.E[x]\xrightarrow[1]{}\rho\theta_1\cup\{(x,v_1)\}.E_1[v_1]} \end{aligned}$$

2. $e \xrightarrow[1]{} e \equiv e_1$. Trivial.

3. For group 3, we use induction on either the length of the SR-sequence (n), or on the structure of the term e. We give two examples.

a) $e \equiv \lambda x.M \xrightarrow[1]{} \lambda x.M_1 \equiv e_1$, where $M \xrightarrow[1]{} M_1$.
Since $e_1, \ldots, e_n$ is a SR-sequence then $M_1, \ldots, M_n$ is a SR-sequence. Since $M \xrightarrow[1]{} M_1$ and M is a subterm of e, by the inductive hypothesis there exists a SR-sequence $M, \ldots, M_n$. Thus, $\lambda x.M, \ldots \lambda x.M_n$ is a SR-sequence.

c) $e \equiv \rho\theta.M \xrightarrow[1]{} \rho\theta_1.M_1 \equiv e_1$, where $\theta \xrightarrow[1]{} \theta_1$ and $M \xrightarrow[1]{} M_1$.
Since $\rho\theta_1.M_1, e_2, \ldots, e_n$ is a SR-sequence, there are two possible cases:

i) $e_1 \equiv \rho\theta_1.M_1 \longmapsto_{vw-} e_2$. By Lemma A.1.16, if $e \xrightarrow[1]{} e_1 \longmapsto_{vw-} e_2$, then there exists e' such that $e \longmapsto^{+}_{vw-} e' \xrightarrow[1]{} e_2$. By induction we have that $e', \ldots, e_n$ is SR-sequence.

ii) $\theta_1, \ldots, \theta_i$ and $M_1, \ldots, M_j$ are SR-sequences and $e_n \equiv \rho\theta_i.M_j$. Because i and j must be less than n we can use the inductive hypothesis on the length of the smaller sequences. That is, since $\theta \xrightarrow[1]{} \theta_1$ and $\theta_1, \ldots, \theta_i$ is a SR-sequence, then there exists a SR-sequence $\theta, \ldots, \theta_i$. Similarly, we know that there is a SR-sequence $M, \ldots, M_j$. Thus, $\rho\theta.M, \ldots, \rho\theta.M_j, \ldots, \rho\theta_i.M_j \equiv e_n$ is a SR-sequence.

■*A*.1.14

Lemma A.1.15 *There are two parts to the lemma; one for $\underset{1a}{\twoheadrightarrow}$ and one for $\underset{1}{\twoheadrightarrow}$:*

1. *If* $e \underset{1a}{\twoheadrightarrow} e'$, $v \underset{1a}{\twoheadrightarrow} v'$, *and* $x \notin AV(e)$, *then*
   - $e[x \leftarrow v] \underset{1a}{\twoheadrightarrow} e'[x \leftarrow v']$.
2. *If* $e \underset{1}{\twoheadrightarrow} e'$, $v \underset{1}{\twoheadrightarrow} v'$, *and* $x \notin AV(e)$, *then*
   (a) $e[x \leftarrow v] \underset{1}{\twoheadrightarrow} e'[x \leftarrow v']$
   (b) $s_{e[x\leftarrow v]\underset{1}{\twoheadrightarrow} e'[x\leftarrow v']} < s_{(\lambda x.e)v\underset{1}{\twoheadrightarrow} e'[x\leftarrow v']}$

Proof. The parallel reduction relations differ only in the first group in the definition. Specifically, clauses f and g are different for each relation. In the proof, we use $\underset{1}{\twoheadrightarrow}$ for cases which are shared and $\underset{1a}{\twoheadrightarrow}$ only for the cases that are not defined in $\underset{1}{\twoheadrightarrow}$.

We use induction on the proof of $e \underset{1}{\twoheadrightarrow} e'$ and case analysis of the last proof step. For the second part, we prove the equivalent statement:

$$s_{e[x\leftarrow v]\underset{1}{\twoheadrightarrow} e'[x\leftarrow v']} \leq s_{e\underset{1}{\twoheadrightarrow} e'} + n(x, e')s_{v\underset{1}{\twoheadrightarrow} v'}$$

From the definition of $\underset{1}{\twoheadrightarrow}$, we know that $e \underset{1}{\twoheadrightarrow} e'$ must have one of the following forms:

1. We pick two examples from the first group of the definition of $\underset{1}{\twoheadrightarrow}$.

   b) $e \equiv (\lambda y.M)u \underset{1}{\twoheadrightarrow} M'[y \leftarrow u'] \equiv e'$ where $y \notin AV(M)$, $M \underset{1}{\twoheadrightarrow} M'$ and $u \underset{1}{\twoheadrightarrow} u'$.

   Applying the inductive hypothesis to the subproofs $M \underset{1}{\twoheadrightarrow} M'$ and $N \underset{1}{\twoheadrightarrow} N'$, we get $f[x \leftarrow v] \underset{1}{\twoheadrightarrow} M'[x \leftarrow v']$ and $u[x \leftarrow v] \underset{1}{\twoheadrightarrow} u'[x \leftarrow v']$.

$$\begin{aligned}
e[x \leftarrow v] &\equiv ((\lambda y.M)u)[x \leftarrow v] && \text{(A.1)}\\
&\equiv (\lambda y.M[x \leftarrow v])u[x \leftarrow v] && \text{(A.2)}\\
&\underset{1}{\twoheadrightarrow} M'[x \leftarrow v'][y \leftarrow u'[x \leftarrow v']] && \text{(A.3)}\\
&\equiv M'[y \leftarrow u'][x \leftarrow v'] && \text{(A.4)}\\
&\equiv e'[x \leftarrow v'] && \text{(A.5)}
\end{aligned}$$

The inductive hypothesis and the fact that $u[x \leftarrow v]$ is a value allows the reduction in step A.3 above.

$$\begin{aligned} s_{e[x\leftarrow v]\underset{1}{\twoheadrightarrow} e'[x\leftarrow v']} &= s_{M[x\leftarrow v]\underset{1}{\twoheadrightarrow} M'[x\leftarrow v']} + \\ &\quad n(y, M'[x \leftarrow v'])s_{u[x\leftarrow v]\underset{1}{\twoheadrightarrow} u'[x\leftarrow v']} + 1 \\ &\leq s_{M\underset{1}{\twoheadrightarrow} M'} + n(x, M')s_{v\underset{1}{\twoheadrightarrow} v'} + \\ &\quad n(y, M'[x \leftarrow v'])(s_{u\underset{1}{\twoheadrightarrow} u'} + n(x, u')s_{v\underset{1}{\twoheadrightarrow} v'}) + 1 \end{aligned}$$

d) $e \equiv \rho\theta \cup \{(y, u)\}.E[y] \underset{1}{\twoheadrightarrow} \rho\theta' \cup \{(y, u')\}.E'[u'] \equiv e'$
where $\theta \underset{1}{\twoheadrightarrow} \theta'$, $u \underset{1}{\twoheadrightarrow} u'$, and $E \underset{1}{\twoheadrightarrow} E'$.
By the inductive hypothesis, $\theta[x \leftarrow v] \underset{1}{\twoheadrightarrow} \theta'[x \leftarrow v']$, $u[x \leftarrow v] \underset{1}{\twoheadrightarrow} u'[x \leftarrow v']$ and $E[x \leftarrow v] \underset{1}{\twoheadrightarrow} E'[x \leftarrow v']$.

$$\begin{aligned} e[x \leftarrow v] &\equiv (\rho\theta \cup \{(y, u)\}.E[y])[x \leftarrow v] \\ &\equiv \rho\theta[x \leftarrow v] \cup \{(y, u[x \leftarrow v])\}.E[y][x \leftarrow v] \\ &\underset{1}{\twoheadrightarrow} \rho\theta'[x \leftarrow v'] \cup \{(y, u'[x \leftarrow v'])\}.E'[u'[x \leftarrow v']][x \leftarrow v'] \\ &\equiv (\rho\theta' \cup \{(y, u')\}.E'[u'])[x \leftarrow v'] \\ &\equiv e'[x \leftarrow v'] \end{aligned}$$

$$\begin{aligned} s_{e[x\leftarrow v]\underset{1}{\twoheadrightarrow} e'[x\leftarrow v']} &= s_{\theta[x\leftarrow v]\underset{1}{\twoheadrightarrow} \theta'[x\leftarrow v']} + 2s_{u[x\leftarrow v]\underset{1}{\twoheadrightarrow} u'[x\leftarrow v']} + \\ &\quad s_{E[x\leftarrow v]\underset{1}{\twoheadrightarrow} E'[x\leftarrow v']} + 1 \\ &\leq s_{\theta\underset{1}{\twoheadrightarrow} \theta'} + n(x, \theta')s_{v\underset{1}{\twoheadrightarrow} v'} + 2(s_{u\underset{1}{\twoheadrightarrow} u'} + n(x, u')s_{v\underset{1}{\twoheadrightarrow} v'}) \\ &\quad + s_{E\underset{1}{\twoheadrightarrow} E'} + n(x, E')s_{v\underset{1}{\twoheadrightarrow} v'} + 1 \\ &= s_{\theta\underset{1}{\twoheadrightarrow} \theta'} + 2s_{u\underset{1}{\twoheadrightarrow} u'} + s_{E\underset{1}{\twoheadrightarrow} E'} + n(x, e')s_{v\underset{1}{\twoheadrightarrow} v'} + 1 \\ &= s_{e\underset{1}{\twoheadrightarrow} e'} + n(x, e')s_{v\underset{1}{\twoheadrightarrow} v'} \end{aligned}$$

f_1) $e \equiv \rho\theta_0 \cup \theta_1.M \underset{1a}{\twoheadrightarrow} \rho\theta_1'.M' \equiv e'$,
where $\theta_1 \underset{1a}{\twoheadrightarrow} \theta_1'$, $M \underset{1}{\twoheadrightarrow} M'$, and $Dom(\theta_0) \cap FV(\rho\theta_1.M) = \varnothing$.
Since $Dom(\theta_0) \cap FV(\rho\theta_1[x \leftarrow v].M[x \leftarrow v]) = \varnothing$, we have

$$\begin{aligned} e[x \leftarrow v] &\equiv (\rho\theta_0 \cup \theta_1.M)[x \leftarrow v] \\ &\equiv \rho\theta_0[x \leftarrow v] \cup \theta_1[x \leftarrow v].M[x \leftarrow v] \\ &\underset{1a}{\twoheadrightarrow} \rho\theta_1'[x \leftarrow v'].M'[x \leftarrow v'] \\ &\equiv (\rho\theta_1'.M')[x \leftarrow v'] \\ &\equiv e'[x \leftarrow v'] \end{aligned}$$

2. $e \underset{1}{\twoheadrightarrow} e \equiv e'$

   We use induction on the structure of e and case analysis of e. We give two examples:

   - $e \equiv (\textbf{set!}\ y\ M)$

     Since $x \notin AV(e)$, then $y \neq x$. Thus, $e[x \leftarrow v] \equiv (\textbf{set!}\ y\ M[x \leftarrow v])$. By the inductive hypothesis, $M[x \leftarrow v] \underset{1}{\twoheadrightarrow} M[x \leftarrow v']$. Thus, $(\textbf{set!}\ y\ M[x \leftarrow v]) \underset{1}{\twoheadrightarrow} (\textbf{set!}\ y\ M[x \leftarrow v']) \equiv e'[x \leftarrow v']$.
   - $e \equiv \rho\theta.M$

     $e[x \leftarrow v] \equiv \rho\theta[x \leftarrow v].M[x \leftarrow v]$

     By the inductive hypothesis, $\theta[x \leftarrow v] \underset{1}{\twoheadrightarrow} \theta[x \leftarrow v']$ and $M[x \leftarrow v] \underset{1}{\twoheadrightarrow} M[x \leftarrow v']$. Thus, $\rho\theta[x \leftarrow v].M[x \leftarrow v] \underset{1}{\twoheadrightarrow} \rho\theta[x \leftarrow v'].M[x \leftarrow v'] \equiv e'[x \leftarrow v']$.

3. We use the inductive hypothesis for subterms for each of these cases. For example,

   a) $e \equiv \lambda y.M \underset{1}{\twoheadrightarrow} \lambda y.M' \equiv e'$, where $M \underset{1}{\twoheadrightarrow} M'$.

   $e[x \leftarrow v] \equiv \lambda y.M[x \leftarrow v]$. By the inductive hypothesis, $M[x \leftarrow v] \underset{1}{\twoheadrightarrow} M'[x \leftarrow v']$. Thus, $\lambda y.M[x \leftarrow v] \underset{1}{\twoheadrightarrow} \lambda y.M'[x \leftarrow v'] \equiv e'[x \leftarrow v']$.

■A.1.15

Lemma A.1.16 *If $e \underset{1}{\twoheadrightarrow} e_1 \longmapsto_{vw-} e_2$, then there exists e_1' such that $e \longmapsto^{+}_{vw-} e_1' \underset{1}{\twoheadrightarrow} e_2$.*

Proof. By lexicographic induction on $\langle s_{e \underset{1}{\twoheadrightarrow} e_1}, e\rangle$.

1. $e \underset{1}{\twoheadrightarrow} e_1 \longmapsto_{vw-} e_2$.

   We showed in the first part of Lemma A.1.14 that there exists a term e' such that $e \longmapsto_{vw-} e' \underset{1}{\twoheadrightarrow} e_1$ and $s_{e' \underset{1}{\twoheadrightarrow} e_1} < s_{e \underset{1}{\twoheadrightarrow} e_1}$. By the inductive hypothesis of this lemma, there exists e_1' such that $e' \longmapsto^{+}_{vw-} e_1' \underset{1}{\twoheadrightarrow} e_2$. Since $e \longmapsto_{vw-} e'$, we have $e \longmapsto^{+}_{vw-} e_1' \underset{1}{\twoheadrightarrow} e_2$.

2. $e \equiv e_1$. Trivial.

3. We give three prototypical examples.

b) $e \equiv (\textbf{set!}\ x\ M) \underset{1}{\twoheadrightarrow} (\textbf{set!}\ x\ M_1) \longmapsto_{vw-} e_2$, where $M \underset{1}{\twoheadrightarrow} M_1$.

By definition of standard reduction we have $(\textbf{set!}\ x\ M_1) \equiv E[p]$ and $e_2 \equiv E[q]$, where $(p,q) \in \beta_v \cup \beta_{v\sigma} \cup \delta$. Clearly, $E \equiv (\textbf{set!}\ x\ F)$ and $M_1 \equiv F[p]$ for some context F. So we have $e_2 \equiv (\textbf{set!}\ x\ M_2)$ where $M_2 \equiv F[q]$. Thus, we have $M \underset{1}{\twoheadrightarrow} M_1 \longmapsto_{vw-} M_2$. By the inductive hypothesis, $M \longmapsto^{+}_{vw-} M_1' \underset{1}{\twoheadrightarrow} M_2$ and therefore, $e \longmapsto^{+}_{vw-} (\textbf{set!}\ x\ M_1') \underset{1}{\twoheadrightarrow} (\textbf{set!}\ x\ M_2) \equiv e_2$.

c) $e \equiv \rho\theta.M \underset{1}{\twoheadrightarrow} \rho\theta_1.M_1 \longmapsto_{vw-} e_2$, where $\theta \underset{1}{\twoheadrightarrow} \theta_1$ and $M \underset{1}{\twoheadrightarrow} M_1$.

By definition of standard reduction, there are three possibilites:

i.) $\rho\theta_1.M_1 \equiv p \longmapsto_{vw-} q \equiv e_2$,

ii.) $\rho\theta_1.M_1 \equiv E[p] \longmapsto_{vw-} E[q] \equiv e_2$, or

iii.) $\rho\theta_1.M_1 \equiv \rho\theta'.E[p] \longmapsto_{vw-} \rho\theta'.E[q] \equiv e_2$,

where $(p,q) \in$ **vw-** in the first case and $(p,q) \in \delta \cup \beta_v \cup \beta_{v\sigma}$ in the last cases.

The second case is impossible for non-empty E. Consider the first case. There are three possible instantiations for p and q; we give a prototypical example below:

$\rho\theta_1.M_1 \equiv \rho\theta_1' \cup \{(x,v_1)\}.E_1[x] \longmapsto_{vw-} \rho\theta_1' \cup \{(x,v_1)\}.E_1[v_1] \equiv e_2$

- $\theta_1 \equiv \theta_1' \cup \{(x,v_1)\}$, and $M_1 \equiv E_1[x]$.
- $\theta \equiv \theta' \cup \{(x,v)\}$, where $\theta' \underset{1}{\twoheadrightarrow} \theta_1'$, $v \underset{1}{\twoheadrightarrow} v_1$ (because $\theta \underset{1}{\twoheadrightarrow} \theta_1$).
- $M \underset{1}{\twoheadrightarrow} E_1[x]$ (because $M \underset{1}{\twoheadrightarrow} M_1$).
- $M \longmapsto^{*}_{vw-} E[x]$, where $E \underset{1}{\twoheadrightarrow} E_1$ (Lemma A.1.17).
- $\rho\theta.M \longmapsto^{*}_{vw-} \rho\theta' \cup \{(x,v)\}.E[x] \longmapsto_{vw-} \rho\theta' \cup \{(x,v)\}.E[v] \underset{1}{\twoheadrightarrow}$ $\rho\theta_1' \cup \{(x,v_1)\}.E_1[v_1]$.

For the third case we have the following:

$$e \equiv \rho\theta.M \underset{1}{\twoheadrightarrow} \rho\theta'.E[p] \longmapsto_{vw-} \rho\theta'.E[q] \equiv e_2$$

Clearly, $\theta \underset{1}{\twoheadrightarrow} \theta'$ and $M \underset{1}{\twoheadrightarrow} E[p] \longmapsto_{vw-} E[q]$. By the inductive hypothesis, there exists M' such that $M \longmapsto^{+}_{vw-} M' \underset{1}{\twoheadrightarrow} E[q]$. Thus, we have

$$e \equiv \rho\theta.M \longmapsto^{+}_{vw-} \rho\theta.M' \underset{1}{\twoheadrightarrow} \rho\theta'.E[q] \equiv e_2$$

d) $e \equiv MN \underset{1}{\twoheadrightarrow} M_1N_1 \longmapsto_{vw-} e_2$, where $M \underset{1}{\twoheadrightarrow} M_1$ and $N \underset{1}{\twoheadrightarrow} N_1$.

By definition of standard reduction, there are two cases:

- $M_1N_1 \equiv E_1[\rho\theta_1.L_1] \longmapsto_{vw-} \rho\theta_1.E_1[L_1] \equiv e_2$, or
- $M_1N_1 \equiv E[p] \longmapsto_{vw-} E[q]$, where $(p,q) \in \delta \cup \beta_v \cup \beta_{v\sigma}$.

For the first case, we have $MN \longmapsto^*_{vw-} E[\rho\theta.L]$, by Lemma A.1.17. Thus, we have $e \equiv MN \longmapsto^*_{vw-} E[\rho\theta.L] \longmapsto_{vw-} \rho\theta.E[L] \xrightarrow[1]{} \rho\theta_1.E_1[L_1]$.

For the second case, consider empty E. Although there are three possible subcases, we only give the one with δ.

$M_1N_1 \equiv fv_1 \longmapsto_{vw-} \delta(f,v_1) \equiv e_2$

- $M_1 \equiv f$, $N_1 \equiv v_1$.
- $M \longmapsto^*_{vw-} f$ (Lemma A.1.17).
- $N \longmapsto^*_{vw-} v$, where $v \xrightarrow[1]{} v_1$ (Lemma A.1.17).
- $MN \longmapsto^*_{vw-} fN \longmapsto^*_{vw-} fv \longmapsto_{vw-} \delta(f,v) \equiv \delta(f,v_1)$.

This holds because of the restriction on δ.

Finally, consider the case when $E \not\equiv [\;]$:

$$MN \xrightarrow[1]{} E[p] \longmapsto_{vw-} E[q] \equiv e_2$$

Since p must occur within either M_1 or N_1 we can use the inductive hypothesis on the subterm. For example, if p is in M_1 then we have the following: $MN \xrightarrow[1]{} F[p]N \longmapsto_{vw-} F[q]N$, where $M \xrightarrow[1]{} F[p] \longmapsto_{vw-} F[q]$. By the inductive hypothesis, there exists M' such that $M \longmapsto^+_{vw-} M' \xrightarrow[1]{} F[q]$. Thus, $MN \longmapsto^+_{vw-} M'N \xrightarrow[1]{} F[q]N \equiv e_2$.

∎A.1.16

Lemma A.1.17 *If $e \xrightarrow[1]{} e'$ and*

1. *if $e' \equiv E'[x]$, then $e \longmapsto^*_{vw-} E[x]$, where $E \xrightarrow[1]{} E'$.*
2. *if $e' \equiv E'[(\texttt{set!}\ x\ v')]$, then $e \longmapsto^*_{vw-} E[(\texttt{set!}\ x\ v)]$, where $E \xrightarrow[1]{} E'$ and $v \xrightarrow[1]{} v'$.*
3. *if $e' \equiv E'[\rho\theta'.M']$, then $e \longmapsto^*_{vw-} E[\rho\theta.M]$, where $E \xrightarrow[1]{} E'$, $\theta \xrightarrow[1]{} \theta'$ and $M \xrightarrow[1]{} M'$.*
4. *if $e' \equiv \lambda x.M'$, then $e \longmapsto^*_{vw-} \lambda x.M$, where $M \xrightarrow[1]{} M'$.*
5. *if $e' \equiv c$, for constant c, then $e \longmapsto^*_{vw-} c$.*

Proof. By induction on $\langle s_{e \underset{1}{\twoheadrightarrow} e'}, e\rangle$. We use case analysis on the reduction $e \underset{1}{\twoheadrightarrow} e'$.

1. From the previous lemmas, there exists a term e'' such that $e \longmapsto_{vw-} e'' \underset{1}{\twoheadrightarrow} e'$ and $s_{e'' \underset{1}{\twoheadrightarrow} e'} < s_{e \underset{1}{\twoheadrightarrow} e'}$. By the inductive hypothesis on $e'' \underset{1}{\twoheadrightarrow} e'$ we have the following:

   - if $e' \equiv E'[x]$, then $e \longmapsto_{vw-} e'' \longmapsto^{*}_{vw-} E[x]$
   - if $e' \equiv E'[(\text{set! } x\ v')]$, then $e \longmapsto_{vw-} e'' \longmapsto^{*}_{vw-} E[(\text{set! } x\ v)]$
   - if $e' \equiv E'[\rho\theta'.M']$, then $e \longmapsto_{vw-} e'' \longmapsto^{*}_{vw-} E[\rho\theta.M]$
   - if $e' \equiv \lambda x.M'$, then $e \longmapsto_{vw-} e'' \longmapsto^{*}_{vw-} \lambda x.M$
   - if $e' \equiv c$, then $e \longmapsto_{vw-} c$.

2. $e \equiv e'$ Trivial.

3. We give two typical examples.

   b) $e \equiv (\text{set! } x\ M) \underset{1}{\twoheadrightarrow} (\text{set! } x\ M') \equiv e'$.

   Assume that $e' \equiv E'[p']$ where p' is one of x, $(\text{set! } x\ v')$ or $\rho\theta'.q'$. There are two cases:

   - $e' \equiv E'[(\text{set! } x\ M')]$, where $E' \equiv [\,]$. Then we have $e \equiv E[(\text{set! } x\ M)]$, where $E \equiv [\,]$.
   - $e' \equiv (\text{set! } x\ F'[p'])$. Since $M \underset{1}{\twoheadrightarrow} F'[p']$, we can use the inductive hypothesis to get $M \longmapsto^{*}_{vw-} F[p]$. Thus, we have $e \equiv (\text{set! } x\ M) \longmapsto^{*}_{vw-} (\text{set! } x\ F[p])$.

   d) $e \equiv MN \underset{1}{\twoheadrightarrow} M'N' \equiv e'$

   Assume that $e' \equiv E'[p']$ where p' is one of x, $(\text{set! } x\ v')$ or $\rho\theta'.q'$. If $e' \equiv E'[p']$, then p' must occur with M' or N'. Suppose it occurs in M', then we have $e' \equiv F'[p']N'$. Since $M \underset{1}{\twoheadrightarrow} F'[p']$, we can use the inductive hypothesis to get $M \longmapsto^{*}_{vw-} F[p]$ where $F \underset{1}{\twoheadrightarrow} F'$. Thus, we have $e \equiv MN \longmapsto^{*}_{vw-} F[p]N \equiv E[p]$ where $E \underset{1}{\twoheadrightarrow} E'$. If p' is in N', then a similar argument holds.

$\blacksquare_{A.1.17}$

This completes the proof of the Standardization Lemma. $\blacksquare_{A.1.12}$

Strong Normalization

In this section we prove the Strong Normalization Theorem for the notion of reduction $\mathbf{s}_v$, which differs from imperative fragment of **vw** in one important way: the range of the δ function is restricted to constants.

Theorem 2.1.6 (Strong Normalization) *Let δ_c be the restriction of the relation δ as follows:*

$$(f\ v) = \delta(f,v), \text{ where } \delta(f,v) \in \mathit{Consts} \qquad (\delta_c)$$

Let $\mathbf{s}_v = \delta_c \cup D_v \cup \sigma_v \cup \rho_\cup \cup \rho_{lift} \cup \mathbf{gc}$. *The notion of reduction $\mathbf{s}_v$ is strongly normalizing.*

Proof. We define a potential function, P, that maps terms (and ρ-sets) to $\mathbb{N} \times \mathbb{N}$ and show that for any reduction $e \longrightarrow_{s_v} e'$, we have $P(e) \succ P(e')$, where $\succ$ is the lexicographic ordering relation. The definition of P relies upon the subordinate functions μ, D, and S_m. To simplify the presentation, we let $m = \mu(e)$ and $n = \mu(e')$.

1. $P(e) = \langle S_m(e), D(e)\rangle$; $P(e') = \langle S_n(e'), D(e')\rangle$, by definition of P.
2. $m \geq n$, by Lemma A.1.19(part 1).
3. $\langle S_m(e), D(e)\rangle \succ \langle S_m(e'), D(e')\rangle$, by Lemma A.1.19(part 2).
4. $S_m(e') \geq S_n(e')$, by Lemma A.1.21 and 2.
5. $\langle S_m(e), D(e)\rangle \succ \langle S_n(e'), D(e')\rangle$, by 3 and 4.
6. $P(e) \succ P(e')$, by 1 and 5.

We need only provide the definition of P and prove the two lemmas required by the proof above.

Definition A.1.18. (*Potential Function*) The potential function P maps terms to $\mathbb{N} \times \mathbb{N}$ as defined below. For term e,

$$P(e) = \langle S_{\mu(e)}(e), D(e) \rangle$$

The function S_m gives the maximum size of a term, given that m is the maximum subterm. It is defined for terms and ρ-sets as follows:

$$\begin{aligned}
S_m(c) &= 1 \\
S_m(x) &= m + 1 \\
S_m(\lambda x.e) &= S(e) \\
S_m(e_1 e_2) &= S_m(e_1) + S_m(e_2) \\
S_m(\mathsf{set!}\ x\ e) &= S_m(e) + m \\
S_m(\rho\theta.e) &= S_m(\theta) + S_m(e) + 1 \\
S_m(\theta \cup \{(x, v)\}) &= S_m(\theta) + S_m(v)
\end{aligned}$$

The function μ gives the maximum possible size of values in a term or a ρ-set.

$$\begin{aligned}
\mu(c) &= 1 \\
\mu(x) &= 0 \\
\mu(\lambda x.e) &= S(e) \\
\mu(e_1 e_2) &= max(\mu(e_1), \mu(e_2)) \\
\mu(\mathsf{set!}\ x\ e) &= \mu(e) \\
\mu(\rho\theta.e) &= max(\mu(\theta), \mu(e)) \\
\mu(\theta \cup \{(x, v)\}) &= max(\mu(\theta), \mu(v))
\end{aligned}$$

The function D measures the *depth* of nested expressions:

$$\begin{aligned}
D(e_1 e_2) &= D(e_1) + D(e_2) \\
D(\mathsf{set!}\ x\ e) &= D(e) + 1 \\
D(e) &= 1 \text{ for all other } e
\end{aligned}$$

We extend the functions S_m, D and μ for contexts in the natural way with $S_m([\,]) = D([\,]) = \mu([\,]) = 0$.

Lemma A.1.19 *For any two terms (or ρ-sets) e and e', if $e \longrightarrow_{s_v} e'$ then,*

1. $\mu(e) \geq \mu(e')$

2. $\langle S_m(e), D(e)\rangle \succ \langle S_m(e'), D(e')\rangle$, *where* $m \geq \mu(e)$.

Proof. The proof is by induction on the structure of e and case analysis of $e \longrightarrow_{s_v} e'$. We show that $\mu(e) \geq \mu(e')$ and that for all but one case, $S_m(e) > S_m(e')$, for $m \geq \mu(e)$. For one case, we show that $S_m(e) = S_m(e')$ and $D(e) > D(e')$. The proof requires the following lemmas, which we prove later:

Lemma A.1.20: $\forall m : S_m(E[e]) = S_m(E) + S_m(e)$, $D(E[e]) = D(E) + D(e)$ and $\mu(E[e]) = max(\mu(E), \mu(e))$.

Lemma A.1.21: If $m \geq n$ then $S_m(e) \geq S_n(e)$.

Lemma A.1.22: $\mu(e) \geq S_{\mu(e)}(v)$ for all subterms v of e.

We only show a few cases of $e \longrightarrow_{s_v} e'$. The other cases are similar.

- $e \equiv \rho\theta \cup \{(x, v)\}.E[x] \longrightarrow_{s_v} \rho\theta \cup \{(x, v)\}.E[v] \equiv e'$

$$\begin{aligned}
\mu(e) &= max(\mu(\theta), \mu(v), \mu(E[x])) & \\
&= max(\mu(\theta), \mu(v), \mu(E), \mu(x)) & (A.1.20)\\
&\geq max(\mu(\theta), \mu(v), \mu(E)) & \\
&= max(\mu(\theta), \mu(v), \mu(E), \mu(v)) & \\
&= max(max(\mu(\theta), \mu(v)), \mu(E[v])) & (A.1.20)\\
&= \mu(e') &
\end{aligned}$$

$$\begin{aligned}
S_m(e) &= S_m(\theta \cup \{(x, v)\}) + S_m(E[x]) + 1 & \\
&= S_m(\theta \cup \{(x, v)\}) + S_m(E) + S_m(x) + 1 & (A.1.20)\\
&= S_m(\theta \cup \{(x, v)\}) + S_m(E) + (m + 1) + 1 & \\
&> S_m(\theta \cup \{(x, v)\}) + S_m(E) + S_m(v) + 1 & (A.1.22)\\
&= S_m(\theta \cup \{(x, v)\}) + S_m(E[v]) + 1 & (A.1.20)\\
&= S_m(e') &
\end{aligned}$$

- $e \equiv E[\rho\theta.M] \longrightarrow_{s_v} \rho\theta.E[M] \equiv e'$, where $E \neq [\,]$.

$$\begin{aligned} \mu(e) &= max(\mu(\theta), \mu(M)), \mu(E)) && (A.1.20) \\ &= \mu(e') \end{aligned}$$

  For this case, $S_m(e) = S_m(e')$. Thus, we must show that $D(e) > D(e')$. We make use of the fact that $D(E) > 0$, if $E \neq [\,]$:

$$\begin{aligned} D(e) &= D(E[\rho\theta.M]) \\ &= D(E) + D(\rho\theta.M) && (A.1.20) \\ &= D(E) + 1 \\ &> 1 && (E \neq [\,]) \\ &= D(\rho\theta.E[M]) \end{aligned}$$

- $e \equiv \lambda x.M \longrightarrow_{s_v} \lambda x.M' \equiv e'$, where $M \longrightarrow_{s_v} M'$.

  For the first part, $\mu(e) = S(M) = S_m(e)$ and $\mu(e') = S(M') = S_m(e')$, for any m. Thus, by showing $S_m(e) > S_m(e')$ we prove both parts.

  Observe that $S_m(e) = S(M) = S_{\mu(M)}(M)$. By inductive hypothesis, $S_{\mu(M)}(M) > S_{\mu(M)}(M')$. By inductive hypothesis of part 1 of this lemma, $\mu(M) \geq \mu(M')$. Thus, by Lemma A.1.21, $S_{\mu(M)}(M') \geq S_{\mu(M')}(M') = S(e') = S_m(e')$:

$$\begin{aligned} S_m(e) &= S_{\mu(M)}(M) \\ &> S_{\mu(M)}(M') && (I.H.) \\ &\geq S_{\mu(M')}(M') && (\mu(M) \geq \mu(M'), A.1.21) \\ &= S_m(e') \end{aligned}$$

- $e \equiv (\textbf{set!}\ x\ M) \longrightarrow_{s_v} (\textbf{set!}\ x\ M') \equiv e'$, where $M \longrightarrow_{s_v} M'$.

  By inductive hypothesis, $\mu(M) \geq \mu(M')$ and $S_m(M) > S_m(M')$, where $m \geq \mu(M) = \mu(e)$.

$$\begin{aligned} \mu(e) &= \mu(M) \\ &\geq \mu(M') && (I.H.) \\ &= \mu(e') \end{aligned}$$

$$\begin{aligned} S_m(e) &= S_m(M) + m \\ &> S_m(M') + m && (I.H.) \\ &= S_m(e') \end{aligned}$$

The other cases are similar to the above. $\blacksquare_{A.1.19}$

The next lemma states that the S_m, D and μ functions are compositional.

Lemma A.1.20 *For term e and evaluation context E,*

1. $S_m(E[e]) = S_m(E) + S_m(e)$,
2. $D(E[e]) = D(E) + D(e)$, *and*
3. $\mu(E[e]) = max(\mu(E), \mu(e))$.

Proof. By simple induction on the structure of E. $\blacksquare_{A.1.20}$

Lemma A.1.21 *If* $m \geq n$ *then* $S_m(e) \geq S_n(e)$.

Proof. Suppose $m \geq n$. We prove by induction on the structure of an expression e that $S_m(e) \geq S_n(e)$. A few cases of e are given below. The other cases are similar.

- $e \equiv c$: $S_m(e) = 1 = S_n(e)$.
- $e \equiv x$: $S_m(e) = m \geq n = S_n(x)$.
- $e \equiv \lambda x.M$: $S_m(e) = S(M) = S_n(e)$.
- $e \equiv MN$: $S_m(MN) = S_m(M) + S_m(N)$. By inductive hypthesis, $S_m(M) + S_m(N) \geq S_n(M) + S_n(N) = S_n(e)$.

$\blacksquare_{A.1.21}$

Lemma A.1.22 *If a value v is a subterm of e then* $\mu(e) \geq S_{\mu(e)}(v)$.

Proof. Note that $S(v) = S_m(v)$ for any m and value v. Thus, it is sufficient to prove that $\mu(e) \geq S(v)$, where v is a subterm of e. The proof proceeds by induction on the structure of e. $\blacksquare_{A.1.22}$

A.2 Call-by-value/Pass-by-reference

In this section we prove the Weak Correspondence Theorem for the call-by-value/pass-by-reference language. The proof structures are similar to the proofs in the previous sections.

Correspondence Theorem

Theorem 2.2.1 (Weak Correspondence) *The λ_v-R-calculus* weakly *corresponds to the call-by-value/pass-by-reference semantics of IS_v, $eval_{vr}$. In particular,*

1. *$\boldsymbol{\lambda}_v$-**R** is "almost" adequate:*

   (a) *if $eval_{vr}(e) = a$ then either*

   - *$\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e = a$, or*
   - *$\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e = \rho\theta \cup \{(x, v)\}.x$ and $\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash \rho\theta \cup \{(x, v)\}.v = a$,*

   (b) *if $\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e = a$, for answer a then $eval_{vr}(e)$ is defined; and*

2. *$\boldsymbol{\lambda}_v$-**R** is sound with respect to $\simeq_{vr}$:*

$$\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e_1 = e_2 \textit{ implies } e_1 \simeq_{vr} e_2.$$

Proof. The proof has the same structure as the proof for the call-by-value/pass-by-worth correspondence theorem. We include the important differences.

1. (Weak) Adequacy.

   (a) Assume $eval_{vr}(e) = a$.

   By definition of $eval_{vr}$, $\rho\emptyset.e \rhd^*_{vr} \rho\theta.v$. Furthermore, by the garbage collection rules, we have $\rho\theta.v \twoheadrightarrow_{vr} a$. The transformation function $\rhd^*_{vr}$ is not a subset of $\twoheadrightarrow_{vr}$ because of the restriction on D'. However, if $\rho\theta \cup \{(x, v)\}.x \rhd_{vr} \rho\theta \cup \{(x, v)\}.v$ then the result is an answer. Thus, this step can only happen as the very last in a sequence. Since $\twoheadrightarrow_{vr}$ can simulate all other rules, then it follows that either $\rho\emptyset.e \twoheadrightarrow_{vr} \rho\theta \cup \{(x, v)\}.x$ or $\rho\emptyset.e \twoheadrightarrow_{vr} \rho\theta.v \twoheadrightarrow_{vr} a$. Also, by *elim*, $\rho\emptyset.e \longrightarrow_{vr} e$. Thus, $\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e = a$ or $\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e = \rho\theta \cup \{(x, v)\}.x$.

   (b) Same as before with $\boldsymbol{\lambda}_v$-**W**.

2. Soundess.

   A. Assume $\boldsymbol{\lambda}_v\text{-}\mathbf{R} \vdash e_1 = e_2$.

   B. Assume C is a program context for e_1 and e_2.

   C. Assume $eval_{vr}(C[e_1])$ is defined. Then,

- $eval_{vr}(C[e_1]) = a$, where a is an answer;
- Either $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = a$ or $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = \rho\theta \cup \{(x,v)\}.x$, by Adequacy, part (a);
- If $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = a$ the proof is same as with $\boldsymbol{\lambda}_v$-**W**. Thus, assume the other case:
- $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_2] = \rho\theta \cup \{(x,v)\}.x$, by transitivity of $=_{vr}$;
- $C[e_2] \twoheadrightarrow_{vr} \rho\theta' \cup \{(x,v')\}.x$, by Church-Rosser Lemma;
- $\rho\varnothing.C[e_2] \rhd^{*}_{vr} \rho\theta''.v''$, by Lemma A.2.1;
- $eval_{vr}(C[e_2])$ is defined, by definition of $eval_{vr}$.

Therefore, $eval_{vr}(C[e_2])$ is defined iff $eval_{vr}(C[e_1])$ is defined.

C. Assume $eval_{vr}(C[e_1]) = c$, for basic constant c.

- $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = c$, by Adequacy, part (a);
- Either $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = c$ or $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = \rho\theta \cup \{(x,c)\}.x$, by Adequacy, part (a);
- If $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_1] = c$ the proof is same as with $\boldsymbol{\lambda}_v$-**W**. Thus, assume the other case:
- $\boldsymbol{\lambda}_v$-**R** $\vdash C[e_2] = \rho\theta \cup \{(x,c)\}.x$, by transitivity of $=_{vr}$;
- $C[e_2] \twoheadrightarrow_{vr} \rho\theta' \cup \{(x,c)\}.x$, by Church-Rosser Lemma;
- $\rho\varnothing.C[e_2] \rhd^{*}_{vr} \rho\theta''.c$, by Lemma A.2.1;
- $eval_{vr}(C[e_2]) = c$, by definition of $eval_{vr}$.

Therefore, $eval_{vr}(C[e_1]) = c$ iff $eval_{vr}(C[e_2]) = c$.

Therefore, $e_1 \simeq_{vr} e_2$.

Therefore, $\boldsymbol{\lambda}_v$-**R** $\vdash e_1 = e_2$ implies $e_1 \simeq_{vr} e_2$.

As with the λ_v-W-calculus, the λ_v-R-calculus satisfies Church-Rosser and Standardization Lemmas. We assume the natural extension of the definitions for the notion of reduction **vr**-, the **vr**-Standard Reduction Function $\longmapsto_{vr-}$ and Standard Reduction Sequences. The proof of the following lemma, which connects the reduction relation $\twoheadrightarrow_{vr}$ with the transformation function $\rhd_{vr}$, completes the proof of correspondence.

Lemma A.2.1 *For a program e, if $e \twoheadrightarrow_{vr} e'$ where e' is either an answer or of the form $\rho\theta \cup \{(x,v)\}.x$, then there exists an answer a such that*

1. *$\rho\emptyset.e \rhd^*_{vr} a$, and*

2. *if e' is a basic constant c, then $a \equiv \rho\theta.c$, and*

3. *if $e' \equiv \rho\theta \cup \{(x,v)\}.x$, then $a \equiv \rho\theta' \cup \{(x,v')\}.v'$.*

Proof. Assume $e \twoheadrightarrow_{vr} e'$. Parts 1 and 2 are the same as in Lemma A.1.3. For part 3, assume e' is of the form $\rho\theta \cup \{(x,v)\}.x$. By Lemma A.2.2 $e \twoheadrightarrow_{vr-} \rho\theta_1 \cup \{(x,v_1)\}.x$. By the Standardization Lemma, there exists an SR-sequence $e, \ldots, \rho\theta_1 \cup \{(x,v_1)\}.x$. Thus, we have $e \longmapsto^*_{vr-} \rho\theta_2 \cup \{(x,v_2)\}.x$ and therefore,

$$\rho\emptyset.e \rhd^*_{vr} \rho\emptyset.\rho\theta_2 \cup \{(x,v_2)\}.x \rhd_{vr} \rho\theta_2 \cup \{(x,v_2)\}.x \rhd_{vr} \rho\theta_2 \cup \{(x,v_2)\}.v_2.$$

Take $a \equiv \rho\theta_2 \cup \{(x,v_2)\}.v_2$. ∎$_{A.2.1}$

The following lemma shows that the garbage collection reductions can be removed from a reduction sequence, without affecting the type of answer.

Lemma A.2.2 *If $e \twoheadrightarrow_{vr} e'$, where e' is either an answer or of the form $\rho\theta \cup \{(x,v)\}.x$, then there exists an e'' such that $e \twoheadrightarrow_{vr-} e''$ and*

1. *if e' is a basic constant c, then either $e'' \equiv c$ or $e'' \equiv \rho\theta.c$,*

2. *if e' is $\rho\theta.v$, then $e'' \equiv \rho\theta'.v'$,*

3. *if e' is $\rho\theta \cup \{(x,v)\}.x$, then $e'' \equiv \rho\theta' \cup \{(x,v')\}.x$.*

Proof.

Assume $e \twoheadrightarrow_{vr} e'$. By Lemma A.2.3, we have $e \twoheadrightarrow_{vr-} e_1 \twoheadrightarrow_{gc} e'$. By Lemma A.2.4 we have $e_1 \twoheadrightarrow_{vr-} e'' \twoheadrightarrow_{gc} e'$, where e'' is of the proper form. Thus, we have $e \twoheadrightarrow_{vr-} e''$.

The last two lemmas show that the garbage collection reductions can be moved to the end of a reduction sequence without affecting the shape of the answer.

Lemma A.2.3 *If $e \longrightarrow_{gc} e_1 \longrightarrow_{vr-} e_2$, then $e \longrightarrow_{vr-} e_1' \longrightarrow_{gc} e_2$.*

Lemma A.2.4 1. *If $e \twoheadrightarrow_{gc} a$, for an answer a, then $e \twoheadrightarrow_{vr-} a' \twoheadrightarrow_{gc} a$, where a' is an answer.*

2. *If $e \twoheadrightarrow_{gc} \rho\theta.x$, where $x \in Dom(\theta)$, then $e \twoheadrightarrow_{vr-} \rho\theta'.x \twoheadrightarrow_{gc} \rho\theta.x$.*

∎$_{A.2.2}$

With the proofs of the Church-Rosser and Standardization Lemmas in the following sections, the proof of the Correspondence Theorem is complete. ∎$_{2.2.1}$

Church-Rosser

Lemma A.2.5 *The notion of reduction* **vr** *(the call-by-value/pass-by-reference calculus) is Church-Rosser.*

Proof. The proof uses a similar technique as the Church-Rosser Lemma for the call-by-value/pass-by-worth calculus. We define the notions of reduction **vra** and **vrb** as follows:

$$\begin{aligned} \textbf{vra} &= \delta \cup \beta_r \cup D'_v \cup \sigma_v \cup \textbf{gc} \\ \textbf{vrb} &= \rho_\cup \cup \rho_{lift}. \end{aligned}$$

Lemma A.2.6 *The notion of reduction* **vra** *is Church-Rosser.*

Proof. As before, we define a parallel reduction relation $\xrightarrow[1a]{}\!\!\!\!\to$. This relation is the same as in definition A.1.9, except clauses b, c and d are replaced with b and d below:

1. If $\theta \xrightarrow[1a]{}\!\!\!\!\to \theta'$, $e \xrightarrow[1a]{}\!\!\!\!\to e'$, $v \xrightarrow[1a]{}\!\!\!\!\to v'$ and $E \xrightarrow[1a]{}\!\!\!\!\to E'$, then

$$\begin{array}{lrcl} b) & (\lambda x.e)y & \xrightarrow[1a]{}\!\!\!\!\to & e'[x \leftarrow y] \\ d) & \rho\theta \cup \{(x,v)\}.E[x] & \xrightarrow[1a]{}\!\!\!\!\to & \rho\theta' \cup \{(x,v')\}.E'[v'], \\ & & & \text{if } E \neq [\;] \text{ and } E \neq F[(\lambda y.e)[\;]] \end{array}$$

Next, we assume that $e \xrightarrow[1a]{}\!\!\!\!\to e_1$ and $e \xrightarrow[1a]{}\!\!\!\!\to e_2$ and use induction on the structure of $e \xrightarrow[1a]{}\!\!\!\!\to e_1$ that there exists an e_3 such that $e_i \xrightarrow[1a]{}\!\!\!\!\to e_3$, for $i = 1, 2$.

We use case analysis on $e \xrightarrow[1a]{}\!\!\!\!\to e_1$:

1. We pick the two clauses in group 1 that differ from the call-by-value/pass-by-worth definition:

   b) $e \equiv (\lambda x.M)y \xrightarrow[1a]{}\!\!\!\!\to M_1[x \leftarrow y] \equiv e_1$, where $M \xrightarrow[1a]{}\!\!\!\!\to M_1$.

   Two possibilities for e_2:

   - $e_2 \equiv (\lambda x.M_2)y$.
   - $e_2 \equiv M_2[x \leftarrow y]$.

   d) $e \equiv \rho\theta \cup \{(x,v)\}.E[x] \xrightarrow[1a]{}\!\!\!\!\to \rho\theta_1 \cup \{(x,v_1)\}.E_1[v_1] \equiv e_1$, where $\theta \xrightarrow[1a]{}\!\!\!\!\to \theta_1$, $v \xrightarrow[1a]{}\!\!\!\!\to v_1$, $E \xrightarrow[1a]{}\!\!\!\!\to E_1$, $E \neq F[(\lambda y.M)[\;]]$, and $E \neq [\;]$.

   There are several possibilities for e_2. We select two:

   - $e_2 \equiv \rho\theta_2 \cup \{(x,v_2)\}.E_2[x]$
   - $e_2 \equiv \rho\theta_2 \cup \{(x,v_2)\}.E_2[v_2]$,

where $\theta \underset{1a}{\twoheadrightarrow} \theta_2$, $v \underset{1a}{\twoheadrightarrow} v_2$, $E \underset{1a}{\twoheadrightarrow} E_2$.
Clearly, $E_2 \neq [\ ]$ and $E_2 \neq F[(\lambda y.M)[\ ]]$. By the inductive hypothesis, there exists θ_3, v_3, and E_3, such that $e_3 \equiv \rho\theta_3 \cup \{(x, v_3)\}.E_3[v_3]$ satisifies the diamond property.

2. Same as Lemma A.1.8.

3. Same as Lemma A.1.8.

■A.2.6

Since **vrb** = **vwb**, we know from Lemma A.1.10 that **vrb** is Church-Rosser. We need only show the commutation of the two subreductions.

Lemma A.2.7 *The* vra*-reduction commutes with* vrb*-reduction.*

Proof. For this proof, let $\longrightarrow_{vrb}$ be the *reflexive closure* of the one-step reductions. By Barendregt [1:65], it is sufficient to show that if $e \underset{1a}{\twoheadrightarrow} e_1$ and $e \longrightarrow_{vrb} e_2$ there exists an e_3 such that $e_1 \twoheadrightarrow_{vrb} e_3$ and $e_2 \underset{1a}{\twoheadrightarrow} e_3$.

1. First consider cases in which $e \underset{1a}{\twoheadrightarrow} e_1$ by group 1 of the definition of $\underset{1a}{\twoheadrightarrow}$. We include only the interesting cases:

   b) $e \equiv (\lambda x.M)y \underset{1a}{\twoheadrightarrow} M_1[x \leftarrow y] \equiv e_1$.

      – $e_2 \equiv (\lambda x.M_2)y$, where $M \longrightarrow_{vrb} M_2$.

      By the inductive hypothesis there exists M_3 such that $M_1 \twoheadrightarrow_{vrb} M_3$ and $M_2 \underset{1a}{\twoheadrightarrow} M_3$. Since $M_1[x \leftarrow y] \twoheadrightarrow_{vrb} M_3[x \leftarrow y]$ we can take $e_3 \equiv M_3[x \leftarrow y]$.

   c) $e \equiv \rho\theta \cup \{(x, v)\}.E[x] \underset{1a}{\twoheadrightarrow} \rho\theta_1 \cup \{(x, v_1)\}.E_1[v_1] \equiv e_1$, where $E \neq F[(\lambda x.M)[\ ]]$ and $E \neq [\ ]$.

      There are a number of possibilities for e_2. The following are representative of all the cases:

      – $e_2 \equiv \rho\theta_2 \cup \{(x, v)\}.E[x]$, where $\theta \longrightarrow_{vrb} \theta_2$.

      – $e_2 \equiv \rho\theta \cup \{(x, v)\}.E_2[x]$, where $E \longrightarrow_{vrb} E_2$.

      For the first case, there exists θ_3 such that $\theta_2 \underset{1a}{\twoheadrightarrow} \theta_3$ and $\theta_1 \twoheadrightarrow_{vrb} \theta_3$. Thus, we take $e_3 \equiv \rho\theta_3 \cup \{(x, v_1)\}.E_1[v_1]$. For the second case, we know that since $E \neq F[(\lambda x.M)[\ ]]$ and $E \neq [\ ]$, then $E_2 \neq [\ ]$ and $E_2 \neq F[(\lambda x.M)[\ ]$. Thus, take $e_3 \equiv \rho\theta_1 \cup \{(x, v_1)\}.E_3[v_1]$. The other cases are similar.

2. Next, consider the cases in which $e \longrightarrow_{vrb} e_2$, because $(e, e_2) \in \mathbf{vrb}$:

   ρ_{lift}) $e \equiv E[\rho\theta.M] \longrightarrow_{vrb} \rho\theta.E[M] \equiv e_2$, where $E \not\equiv [\,]$.

   There are several subcases for $e \xrightarrow[1a]{}\!\!\!\!\!\rightarrow e_1$:

   - $e \equiv E[\rho\theta \cup \{(x,v)\}.F[x]]) \xrightarrow[1a]{}\!\!\!\!\!\rightarrow E_1[\rho\theta_1 \cup \{(x,v_1)\}.F_1[v_1]]) \equiv e_1$, where $F \not\equiv [\,]$ and $F \not\equiv F'[(\lambda y.e)[\,]]$.

     Then, $e_2 \equiv \rho\theta \cup \{(x,v)\}.E[F[x]])$ and we can take $e_3 \equiv \rho\theta_1 \cup_1 \{(x,v_1)\}.E_1[F_1[v_1]]$. Since $F \not\equiv [\,]$, $E[F] \not\equiv [\,]$. Since $F \not\equiv F'[(\lambda y.e)[\,]]$, $E[F] \not\equiv F'[(\lambda y.e)[\,]]$.

   $\rho_\cup$) $e \equiv \rho\theta.E[\rho\theta'.M] \longrightarrow_{vrb} \rho\theta \cup \theta'.E[M] \equiv e_2$.

   Similar to the previous case.

3. Finally, the only remaining cases are those in which $e \equiv C[M] \xrightarrow[1a]{}\!\!\!\!\!\rightarrow C_1[M_1]$ by group 3 of the definition of $\xrightarrow[1a]{}\!\!\!\!\!\rightarrow$, and $e \equiv C'[N] \longrightarrow_{vrb} C'[N_2]$, because $(N, N_2) \in \mathbf{vrb}$, where C and C' are non-empty contexts. We use induction on the subterms for these cases.

∎$_{A.2.7}$

A lemma similar to Lemma A.1.15 must be proven. This completes the proof of the Church-Rosser Lemma. ∎$_{A.2.5}$

Standardization

Lemma A.2.8 (Standardization) $e \longrightarrow\!\!\!\!\!\rightarrow_{vr-} e'$ *iff there exists a Standard Reduction Sequence* $e, \ldots, e'$.

Proof.

We use the same technique as the corresponding proof for the call-by-value/pass-by-worth calculus.

The definition of parallel reduction changes slightly. We delete clause b and replace clauses c and d with the following:

1. If $\theta \xrightarrow[1]{}\!\!\!\!\!\rightarrow \theta'$, $e \xrightarrow[1]{}\!\!\!\!\!\rightarrow e'$, $v \xrightarrow[1]{}\!\!\!\!\!\rightarrow v'$ and $E \xrightarrow[1]{}\!\!\!\!\!\rightarrow E'$, then

$$\begin{array}{lrcl}
b) & (\lambda x.e)y & \xrightarrow[1]{}\!\!\!\!\!\rightarrow & e'[x \leftarrow y] \\
 & s & = & s_{e \xrightarrow[1]{}\!\!\!\rightarrow e'} + 1 \\
d) & \rho\theta \cup \{(x,v)\}.E[x] & \xrightarrow[1]{}\!\!\!\!\!\rightarrow & \rho\theta' \cup \{(x,v')\}.E'[v'], \\
 & & & \text{if } E \not\equiv [\,] \text{ and } E \not\equiv F[(\lambda y.e)[\,]] \\
 & s & = & s_{\theta \xrightarrow[1]{}\!\!\!\rightarrow \theta'} + 2s_{v \xrightarrow[1]{}\!\!\!\rightarrow v'} + s_{E \xrightarrow[1]{}\!\!\!\rightarrow E'} + 1
\end{array}$$

Lemma A.2.9 *If* $e \underset{1}{\twoheadrightarrow} e_1$ *and* $e_1, \ldots, e_n$ *is a* SR*-sequence, then there exists a* SR*-sequence* $e, \ldots, e_n$.

Proof. The proof uses the same method as before. We include only the two different cases from group 1.

b) $e \equiv (\lambda x.M)y \underset{1}{\twoheadrightarrow} M_1[x \leftarrow y] \equiv e_1$, where $M \underset{1}{\twoheadrightarrow} M_1$.

$$e \longmapsto_{vr-} M[x \leftarrow y] \underset{1}{\twoheadrightarrow} M_1[x \leftarrow y] \equiv e_1$$

By Lemma A.2.10:

$$s_{M[x \leftarrow y] \underset{1}{\twoheadrightarrow} M_1[x \leftarrow y]} < s_{(\lambda x.M)y \underset{1}{\twoheadrightarrow} M_1[x \leftarrow y]}$$

d) $e \equiv \rho\theta \cup \{(x, v)\}.E[x] \underset{1}{\twoheadrightarrow} \rho\theta_1 \cup \{(x, v_1)\}.E_1[v_1] \equiv e_1$
where $\theta \underset{1}{\twoheadrightarrow} \theta_1$, $v \underset{1}{\twoheadrightarrow} v_1$, $E \underset{1}{\twoheadrightarrow} E_1$, $E \not\equiv [\,]$ and $E \not\equiv F[(\lambda y.M)[\,]]$.

$$e \longmapsto_{vr-} \rho\theta \cup \{(x, v)\}.E[v] \underset{1}{\twoheadrightarrow} \rho\theta_1 \cup \{(x, v_1)\}.E_1[v_1] \equiv e_1$$

$$\begin{aligned} s_{\rho\theta\cup\{(x,v)\}.E[v] \underset{1}{\twoheadrightarrow} \rho\theta_1\cup\{(x,v_1)\}.E_1[v_1]} &= s_{\theta \underset{1}{\twoheadrightarrow} \theta_1} + s_{v \underset{1}{\twoheadrightarrow} v_1} + s_{E \underset{1}{\twoheadrightarrow} E_1} + s_{v \underset{1}{\twoheadrightarrow} v_1} \\ &< s_{\theta \underset{1}{\twoheadrightarrow} \theta_1} + 2s_{v \underset{1}{\twoheadrightarrow} v_1} + s_{E \underset{1}{\twoheadrightarrow} E_1} + 1 \\ &= s_{\rho\theta\cup\{(x,v)\}.E[x] \underset{1}{\twoheadrightarrow} \rho\theta_1\cup\{(x,v_1)\}.E_1[v_1]} \end{aligned}$$

∎A.2.9

Lemma A.2.10 *If* $e \underset{1}{\twoheadrightarrow} e'$ *then*

1. $e[x \leftarrow y] \underset{1}{\twoheadrightarrow} e'[x \leftarrow y]$

2. $s_{e[x \leftarrow y] \underset{1}{\twoheadrightarrow} e'[x \leftarrow y]} < s_{(\lambda x.e)y \underset{1}{\twoheadrightarrow} e'[x \leftarrow y]}$

Lemma A.2.11 *If* $e \underset{1}{\twoheadrightarrow} e_1 \longmapsto_{vr-} e_2$*, then there exists* e_1' *such that* $e \longmapsto^{+}_{vr-} e_1' \underset{1}{\twoheadrightarrow} e_2$.

Proof. By lexicographic induction on $\langle s_{e \underset{1}{\twoheadrightarrow} e_1}, e \rangle$.

1. By induction on the $s_{e \underset{1}{\twoheadrightarrow} e_1}$, as in previous lemmas.

2. Trivial.

3. We include prototypical cases:

c) $e \equiv \rho\theta.M \xrightarrow[1]{} \rho\theta_1.M_1 \longmapsto_{vr-} e_2$, where $\theta \xrightarrow[1]{} \theta_1$ and $M \xrightarrow[1]{} M_1$.

$\rho\theta_1.M_1 \equiv \rho\theta_1' \cup \{(x, v_1)\}.E_1[x] \longmapsto_{vr-} \rho\theta_1' \cup \{(x, v_1)\}.E_1[v_1] \equiv e_2$, where $E_1 \not\equiv [\,]$ and $E_1 \not\equiv F[(\lambda y.N)[\,]]$.

- $\theta_1 \equiv \theta_1' \cup \{(x, v_1)\}$, and $M_1 \equiv E_1[x]$.
- $\theta \equiv \theta' \cup \{(x, v)\}$, where $\theta' \xrightarrow[1]{} \theta_1'$, $v \xrightarrow[1]{} v_1$ (because $\theta \xrightarrow[1]{} \theta_1$).
- $M \xrightarrow[1]{} E_1[x]$ (because $M \xrightarrow[1]{} M_1$).
- $M \longmapsto^*_{vr-} E[x]$, where $E \xrightarrow[1]{} E_1$ (Lemma A.2.12).
- $\rho\theta.M \longmapsto^*_{vr-} \rho\theta' \cup \{(x, v)\}.E[x] \longmapsto_{vr-} \rho\theta' \cup \{(x, v)\}.E[v] \xrightarrow[1]{} \rho\theta_1' \cup \{(x, v_1)\}.E_1[v_1]$.

We know that $E \not\equiv [\,]$ and $E \not\equiv F[(\lambda y.N)[\,]]$, by Lemma A.2.12.

d) $e \equiv MN \xrightarrow[1]{} M_1N_1 \longmapsto_{vr-} e_2$, where $M \xrightarrow[1]{} M_1$ and $N \xrightarrow[1]{} N_1$.

Consider the case that differs from the previous section:

$M_1N_1 \equiv (\lambda x.L_1)y \longmapsto_{vr-} L_1[x \leftarrow y] \equiv e_2$

- $M_1 \equiv \lambda x.L_1$, $N_1 \equiv y$
- $M \longmapsto^*_{vr-} \lambda x.L$.
- $N \longmapsto^*_{vr-} y$. (Lemma A.2.12)
- $MN \longmapsto^*_{vr-} (\lambda x.L)y \longmapsto_{vr-} L[x \leftarrow y] \xrightarrow[1]{} L_1[x \leftarrow y]$

▪*A*.2.11

Lemma A.2.12 *If $e \xrightarrow[1]{} e'$ and*

1. *if $e' \equiv E'[x]$, then $e \longmapsto^*_{vr-} E[x]$, where $E \xrightarrow[1]{} E'$ and if $E' \not\equiv [\,]$ and $E' \not\equiv F[(\lambda x.N)[\,]]$, then $E \not\equiv [\,]$ and $E \not\equiv F[(\lambda x.N)[\,]]$.*

2. *if $e' \equiv E'[(\text{set! } x\ v')]$, then $e \longmapsto^*_{vr-} E[(\text{set! } x\ v)]$, where $E \xrightarrow[1]{} E'$ and $v \xrightarrow[1]{} v'$.*

3. *if $e' \equiv E'[\rho\theta'.M']$, then $e \longmapsto^*_{vr-} E[\rho\theta.M]$, where $E \xrightarrow[1]{} E'$, $\theta \xrightarrow[1]{} \theta'$ and $M \xrightarrow[1]{} M'$.*

4. *if $e' \equiv \lambda x.M'$, then $e \longmapsto^*_{vw-} \lambda x.M$, where $M \xrightarrow[1]{} M'$.*

5. *if $e' \equiv c$, for constant c, then $e \longmapsto^*_{vw-} c$.*

6. *if $e' \equiv x$, then $e \longmapsto^*_{vr-} x$.*

Proof. Similar method as previous section. We prove only the last proposition by induction on the length of $e \underset{1}{\twoheadrightarrow} x$. Assume $e \underset{1}{\twoheadrightarrow} x$. There are only two possibilities for e:

- $e \equiv x$. Clearly, $e \longmapsto^{*}_{vr-} x$.
- $e \equiv (\lambda y.M)z \underset{1}{\twoheadrightarrow} M'[y \leftarrow z] \equiv x$.

  $$e \longmapsto_{vr-} M[y \leftarrow z] \underset{1}{\twoheadrightarrow} M'[y \leftarrow z] \equiv x$$

  and $s_{M[y\leftarrow z]\underset{1}{\twoheadrightarrow} M'[y\leftarrow z]} < s_{e\underset{1}{\twoheadrightarrow} x}$. Use the inductive hypothesis on the smaller reduction to get $M[y \leftarrow z] \longmapsto^{*}_{vr-} x$.

$\blacksquare_{A.2.12}$

This completes the proof of the Standardization Lemma for the call-by-value/pass-by-reference calculus. $\blacksquare_{A.2.8}$

A.3 Other Languages

The theorems and proofs for the remaining languages are analogous to the proofs already presented. Several modifications are necessary in some places, but the basic structure of the proofs including the lemmas is preserved.

Call-by-value-result

The proofs for the call-by-value-result language are analogous to the proofs for the call-by-value/pass-by-reference language, except there is an additional language construct ($\langle e; e\rangle$) and evaluation context ($\langle v; E\rangle$). With these additions, and with the β_c reduction replacing β_r, the proofs are the same.

Reference Cells

The proofs for the call-by-value/pass-by-worth language with reference cells are analogous to the proofs for the call-by-value/pass-by-worth language with assignment. The term language is different, but the reduction relations are similar.

Call-by-name/Pass-by-worth

Again, the proof structure is the same as with the call-by-value/pass-by-worth language. The difference is that arbitrary expressions my occur where only values were allowed in the call-by-value language.

Call-by-name/Pass-by-reference

The differences between proofs for pass-by-worth and pass-by-reference in call-by-value languages also occur with the call-by-name languages. The restrictions on the variable dereference rules for the two calculi are different, but have similar properties. Whenever this restriction appears in a proof for the call-by-value language, an analogous restriction appears in the proof for the call-by-name language.

Also, the proof for the weaker correspondence theorem is longer due to the complicated statement of the adequacy part of the theorem. However, the proof uses the same arguments.

References

1. BARENDREGT, H.P. *The Lambda Calculus: Its Syntax and Semantics.* Revised Edition. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.

2. CARTWRIGHT, R. AND D. OPPEN. The logic of aliasing. *Acta Inf.* **15**, 1981, 365–384.

3. CHURCH, A. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, 1941.

4. DEMERS, A. AND J. DONAHUE. Making variables abstract: an equational theory for Russell. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983, 59–72.

5. DONAHUE, J.E. *Complementary Definitions of Programming Language Semantics.* Lecture Notes in Computer Science 42, Springer-Verlag, Heidelberg, 1980.

6. DYBVIG, R. K. *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

7. FELLEISEN, M. On the expressive power of programming languages. In *Proc. 1990 European Symposium on Programming.* Neil Jones, Ed. Lecture Notes in Computer Science, 432. 1990, 134–151.

8. FELLEISEN, M. AND D.P. FRIEDMAN. A syntactic theory of sequential state. *Theor. Comput. Sci.* **69**(3), 1989, 243–287. Preliminary version in: *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.

9. FELLEISEN, M. AND R. HIEB. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989.

10. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, AND B. DUBA. A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**(3), 1987, 205–237. Preliminary version in: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

11. GORDON, M.J. *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

12. GRIES, D. AND G. LEVIN. Assignment and Procedure Call Proof Rules. *ACM Trans. Program. Lang. Syst.* **2**(4), 1980, 564–579.

13. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM* **12**, 1969, 576–580.

14. HOARE, C.A.R. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, E. Engeler (Ed.). Lecture Notes in Mathematics 188. Springer-Verlag, Berlin, 1971, 102–116.

15. LANDIN, P.J. A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM* **8**(2), 1965, 89–101; 158–165.

16. LANDIN, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.

17. LANDIN, P.J. The mechanical evaluation of expressions. *Comput. J.* 6(4), 1964, 308–320.

18. MASON, I.A. AND C. TALCOTT. A sound and complete axiomatization of operational equivalence between programs with memory. In *Proc. Symposium on Logic in Computer Science*, 1989.

19. MULLER, R. The operational semantics and equational logic of eval and fexprs. Unpublished manuscript. Harvard University, 1990.

20. NAUR, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM* **6**(1), 1963, 1–17.

21. OLDEROG, E. Sound and complete Hoare-like calculi based on copy rules. *Acta Inf.* **16**, 1981, 161–197.

22. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.

23. REES, J. AND W. CLINGER (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.

24. RIECKE, J.G. A complete and decidable proof system for call-by-value equalities. In *Proc. 17th International Conference on Automata, Languages and Programming*, 1990.

25. SCHMIDT, D.A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, Mass., 1986.

26. US DEPARTMENT OF DEFENSE. *The Programming Language Ada—Reference Manual*, Lecture Notes in Computer Science 106, Springer-Verlag, 1981.

27. WIRTH N. AND C.A.R. HOARE. A contribution to the development of ALGOL. *Commun. ACM* 9(6), 1966, 413–432.