Parameter-Passing and the Lambda Calculus

Erik Crank* Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

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, i.e., call-by-value and call-by-name combined with pass-by-worth and pass-by-reference, respectively. 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 that is derived from the rewriting semantics in a uniform manner;
3. a formal analysis of the correspondence between the calculus and the semantics; and
4. a strong normalization theorem for the imperative fragment of the theory (when applicable).

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/pass-by-worth language with reference cells as first-class values.

*The authors are supported in part by NSF and DARPA. The Instituto de Computacion of the Universidad de la Republica Oriental del Uruguay generously provided facilities for preparing the final version of the paper.

Appeared in:
18th POPL
January 19–22, 1991
Orlando, FL

0

1 Parameter-Passing and Lambda Calculi

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 consensus about the advantages and disadvantages of the various techniques. In this paper we analyze the most common techniques by studying and comparing equational theories for each of them.

Plotkin [20] was the first to consider equational theories for the analysis of parameter-passing techniques. 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 the 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 interchangeability in all program contexts.

Recently, Felleisen with Friedman [9] and Hieb [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 rewriting semantics, and that there are conservative extensions of the λ_v-calculus for reasoning about them, namely the λ_v-{C,S,CS}-calculi.

The extension of Plotkin's work to imperative languages is important because it enables us to consider the whole spectrum of conventional 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 call-by-value and call-by-name versions of a program is the 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 techniques 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 procedure arguments*, while the binding strategy determines *what part of the argument to bind to formal parameters*.

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.[1]

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 paper, we analyze the parameter-passing techniques produced by the various combinations of evaluation and binding strategies. Also included is a discussion of the *reference cell*, a language construct that provides an alternative to the pass-by-reference binding technique. For each technique we develop a rewriting 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, their imperative fragments satisfy strong normalization theorems and are therefore decidable.

The next section of this paper describes in more detail the steps of the analysis: the definition of an operational semantics, the derivation of an equational calculus, and the formal analysis of correspondence between the semantics and the calculus. Section 3 examines each of the parameter-passing techniques using the methods outlined in Section 2. We discuss related work and compare the different techniques in Sections 4 and 5.

[1] Two other delayed evaluation strategies, which we do not consider in this paper, are *call-by-text* and *call-by-need*. Call-by-text never evaluates procedure arguments, thus, delaying evaluation forever. Muller [17] recently analyzed some of the problems of deriving equational theories in the presence of call-by-text. Call-by-need is an optimization of call-by-name in a functional language, but produces different answers than call-by-name in the presence of side-effects. The extended version of this paper [4] contains a brief discussion of call-by-need.

2 From Operational Semantics to Calculi

In our analysis of parameter-passing, there are three main steps. First, we define an operational semantics for a language with a particular parameter-passing technique. Next, we derive an equational calculus directly from the semantics, and finally, we analyze the correspondence between the calculus and the semantics. This section describes each of these steps in more detail.

Definition of a Language and its Operational Semantics

The definition of a language consists of two parts: the specification of a syntax and an operational semantics. We provide one for each of the parameter-passing techniques. The core of our languages, Λ, consists of constants (c), variables (x), λ-abstractions and applications of the λ-calculus:

$$\begin{array}{rcl} e & ::= & x \mid v \mid (ee) \\ v & ::= & c \mid \lambda x.e \end{array}$$

The languages extend Λ with constructs for side-effects, which we describe in later sections.

Following Barendregt's [1] conventions, we assume that, in the following definitions and theorems, the bound variables are distinct from the free variables in expressions, and we identify expressions that differ only by a renaming of the bound variables. 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 free variables in an expression e is denoted by $FV(e)$. An expression with no free variables is *closed*.

Below we need the notion of a context for several definitions. A *context*, C, is a term with a "hole" ([ ]) in the place of a subexpression; $C[e]$ is the expression produced by replacing the hole in a context C with the expression e.

The operational semantics of a language is a partial function, *eval*, from *programs* to *answers*. A program is a closed expression; an answer is some (yet-to-be-defined) syntactic class of expressions (and programs). The *eval* function is based on an abstract machine. The state space of the machine is the set of programs; an evaluation step in the machine corresponds to the rewriting of an entire program:

$$p \rhd p'.$$

Thus, the transitive closure of the transformation function defines the semantics:[2]

$$eval(p) = a \text{ iff } p \rhd^* a,$$

[2] This characterization is too simple. For clarity, we omit the added complications that are described in the next section.

for program p and answer a.

Every evaluation step on the machine begins with partitioning the program into an *evaluation context* and a *redex*. The evaluation context is a special kind of context that specifies the evaluation order of subexpressions in a compound expression. Intuitively, the hole in an evaluation context points to the next subexpression to be evaluated. A redex is an expression that determines how the transformation function rewrites the program. For each redex there is a rewriting rule that specifies how to transform a program that contains the redex in the hole of an evaluation context into a new program. The union of these rewriting rules defines the machine transformation function.

Although we vary the evaluation and binding strategies for user defined procedures below, we assume that primitives have a fixed interpretation. Specifically, we assume that primitives are strict and that they only accept proper values. The set of *values* (*Values*) consists of constants and λ-abstractions. To avoid an overspecification of these constants (*Consts*), we only require the existence of a partial function, δ, from functional constants (*FConsts*) and closed values to closed values. We restrict δ so that functional constants cannot distinguish the intensional aspects of λ-abstractions. That is, if $\delta(f, v)$ is defined for some λ-abstraction v, then there exists a context $C[\ ]$ such that $C[v]$ is a value and for all λ-abstractions v', $\delta(f, v') \equiv C[v']$. This restriction on functional constants is reasonable in the context of most programming languages. Essentially, it prohibits from the language only those primitive functions that can examine the text of procedure bodies. But, the restriction still permits functions that can distinguish constants from procedures (e.g., predicates such as int? and proc?) or that can build larger values composed of procedures (e.g., arrays or lists of procedures).

Observational Equality and Calculi

Since an implementation of a language generally provides the programmer with nothing but the evaluation function, the programmer can only observe the behavior of expressions when they are a part of the program. This fact induces a natural notion of equivalence for arbitrary phrases of a programming language. Intuitively, two terms are observationally equivalent if they are interchangeable in 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. (*Observational Equality*) Two terms are observationally equivalent, $e \simeq e'$, if for all contexts C such that both $C[e]$ and $C[e']$ are programs,

1. $eval(C[e])$ is defined iff $eval(C[e'])$ is defined, and
2. $eval(C[e])$ is a basic constant iff $eval(C[e'])$ is the same constant.

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. For this reason, programming language research attempts to construct theories for proving observational equivalences in an axiomatic fashion. Such theories are often referred to as calculi in analogy to the λ-calculus.

In the following sections, we will attempt to derive an equational theory from the operational semantics of each parameter-passing technique. The most straightforward attempt is to make the transformation steps for programs into equational axioms for arbitrary terms.

Definition 2.2. (*Natural Calculus*) The relation $=$ is the least congruence relation generated by the transformation function $\triangleright$, extended to arbitrary expressions:

$$\begin{array}{rcll} e_1 \triangleright e_2 & \Rightarrow & e_1 = e_2 & (\textit{Axioms}) \\ e_1 = e_2 & \Rightarrow & C[e_1] = C[e_2] & (\textit{Compatible}) \\ & & e_1 = e_1 & (\textit{Reflexive}) \\ e_1 = e_2 & \Rightarrow & e_2 = e_1 & (\textit{Symmetric}) \\ e_1 = e_2, e_2 = e_3 & & & \\ & \Rightarrow & e_1 = e_3 & (\textit{Transitive}) \end{array}$$

For a specific theory **th**, we write $\mathbf{th} \vdash e_1 = e_2$ if $e_1 = e_2$ by the above axioms and rules. If $e_1 = e_2$ without use of the symmetry axiom, we also write $e_1 \longrightarrow e_2$ and refer to the relation as a reduction.

The Analysis of Calculi

Given a semantics, its observational equivalence relation and a calculus, it is natural to ask how the latter relates to the former. Plotkin [20] gave two criteria for correspondence between a calculus and a semantics. First, the calculus must be capable of evaluating programs. In particular, if the semantics defines the meaning of a program e to be a, then the two should be provably equivalent in the calculus. Similarly, if the calculus proves that a program is equivalent to an answer, then the meaning of the program must be defined by the semantics. Second, the calculus must be sound with respect to the observational equivalence relation: If two terms are equal in the calculus, they must be observationally equivalent with respect to *eval*.

Definition 2.3. (*Correspondence*) A theory **th** *corresponds* to a semantics *eval*:

$$\mathbf{th} \models \mathrm{Corr}(eval),$$

if the following conditions hold:

[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.

1. **th** is adequate:

   (a) if $eval(e) = a$, then $\mathbf{th} \vdash e = a$.

   (b) if $\mathbf{th} \vdash e = a$, where a is an answer, then $eval(e)$ is defined.

2. **th** is sound with respect to $\simeq$:

$$\mathbf{th} \vdash e_1 = e_2 \text{ implies } e_1 \simeq e_2$$

A secondary question about a calculus is whether it is decidable. Although our calculi are not decidable, in most cases the imperative fragment of the languages has a decidable subtheory. For these cases we show that there is a *strongly normalizing* notion of reduction that corresponds to the subtheory. We write $\mathbf{r} \models \text{SN}$, if the notion of reduction $\mathbf{r}$ is strongly normalizing.

3 Equational Theories of Parameter-Passing Techniques

3.1 Call-by-value/Pass-by-worth

Many common programming languages, e.g., C, 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. In this section, we present a call-by-value/pass-by-worth language with assignment, derive a calculus for reasoning about the language, and show that it corresponds to the semantics. Both the language and calculus are variations on the work by Felleisen and Hieb [10].

The call-by-value term language, *Idealized Scheme* or IS_v, extends Λ with two new expressions. First, there is an assignment statement, $(\mathbf{set!}\ x\ e)$, 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 sub-expression. The ρ-expression establishes mutually recursive bindings of the variables to their associated values and returns the value of the sub-expression. The term language IS_v is given in the first part of Figure 1.

There are two basic differences between this language and the language of Felleisen and Hieb [10]. First, we use the assignment statement **set!** instead of the *sigma capability*. 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 [20] and the λ_v-S-calculus [10], variables in our language are expressions, not values.[4]

In addition to adopting Barendregt's conventions for free and bound variables, we identify ρ-lists that differ only by the ordering of their pairs:

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

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 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.

Semantics

The semantics of the call-by-value/pass-by-worth language is the partial function, $eval_{vw}$, from programs to answers, where an answer is either a value, or a ρ-expression whose body is a value. As discussed in the previous section, the program rewriting function first partitions a program into an evaluation context and a redex and then transforms it to a new program. A *call-by-value* evaluation context specifies the eager evaluation strategy:

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

The call-by-value/pass-by-worth language redexes are the following expressions: fv, $(\lambda x.e)v$, x, $(\mathbf{set!}\ x\ v)$, and $\rho\theta.e$.

The machine rewrites the program until it produces an answer and then removes all unneeded ρ-bindings by applying "garbage collection" reductions to the answer. More technically, the garbage collection notion of reduction **gc** is the union of the following relations:

$$\begin{array}{rcl} \rho\theta_0 \cup \theta_1.e & \longrightarrow & \rho\theta_1.e \\ & & \quad \text{if } \theta_0 \neq \varnothing \text{ and} \\ & & \quad FV(\rho\theta_1.e) \cap Dom(\theta_0) = \varnothing \\ \rho\varnothing.e & \longrightarrow & e \end{array}$$

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.

[4] 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 [9, 10] 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. Since this property is highly context-sensitive and possibly difficult to determine by a programmer, we abandon this distinction.

Syntax, IS_v**:**

$$\begin{array}{rcl} x & \in & Vars \\ c & \in & Consts \\ f & \in & FConsts \end{array} \qquad \begin{array}{rcll} e & ::= & v \mid x \mid (e\ e) \mid (\textbf{set!}\ x\ e) \mid \rho\theta.e & \text{(Expressions)} \\ v & ::= & c \mid \lambda x.e & \text{(Values)} \\ \theta & ::= & \epsilon \mid \theta(x,v),\ \text{where}\ (x,v') \notin \theta & (\rho\text{-lists}) \\ a & ::= & v \mid \rho\theta.v & \text{(Answers)} \end{array}$$

Evaluation Contexts

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

Transformation Function:

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

Semantics:

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

FIG. 1: Call-by-value/Pass-by-worth Language

The program transformation function, $\triangleright_{vw}$, and the semantics, $eval_{vw}$, for the call-by-value/pass-by-worth language, are defined in the second part of Figure 1. The first two rules, $E.\delta$ and $E.\beta_v$, provide the call-by-value semantics for Λ; the others specify the effects of imperative constructs.

Calculus

We derive an equational theory, $\boldsymbol{\lambda}_v$-**W** or the λ_v-W-calculus, for the call-by-value/pass-by-worth language by generating an equivalence relation, $=_{vw}$, based upon the compatible closure of the program transformation function *and* the notion of reduction **gc**. We write $\boldsymbol{\lambda}_v\text{-}\mathbf{W} \vdash e_1 = e_2$ if $e_1 =_{vw} e_2$. The λ_v-W-calculus corresponds to the semantics $eval_{vw}$.

Theorem 3.1 $\boldsymbol{\lambda}_v\text{-}\mathbf{W} \models \mathrm{Corr}(eval_{vw})$.

Proof. Adequacy follows from the fact that $\triangleright_{vw}$ and **gc** are subsets of $=_{vw}$. For soundness, we derive a *notion of reduction* **vw** from $\triangleright_{vw}$ and **gc**. The reduction satisfies Church-Rosser and Curry-Feys Standardization lemmas that imply the soundness of the calculus.

Otherwise, the proof is an adaptation of the proofs for the calculi of Plotkin [20] and Felleisen et al [9, 10]. For details, we refer the reader to the extended report [4]. ∎

Finally, we examine the question of whether there are decidable subsets of the calculus. In our case, the imperative subtheory $\boldsymbol{\rho}_v$, that is, the theory based upon the transition relations that do *not* deal with procedure applications, is decidable. Specifically, the notion of reduction consisting of these relations 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 by the completeness theorem for a non-recursive fragment of first-order Lisp by Mason and Talcott [16].

Theorem 3.2 *Let δ_c be the modification of the relation $E.\delta$ as follows:*

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

Let $\mathbf{s} = \delta_c \cup D_v \cup \sigma_v \cup \rho_\cup \cup \mathbf{gc}$. *Then,* $\mathbf{s} \models \mathrm{SN}$.

Proof. The proof uses a size argument. We define a "potential" function, $P : Expressions \longrightarrow \mathbb{N} \times \mathbb{N}$, such that $P(e) \succ P(e')$, by lexicographical ordering, if $e \longrightarrow_s e'$. The naïve approaches to defining a size function fail, because certain reductions increase the size of the term and the number of redexes in the term. For example, the following reduction increases the size of the term and the number of redexes because the subexpression e is replicated:

$$\rho\{(x,\lambda x.e)\}.E[x] \longrightarrow_s \rho\{(x,\lambda x.e)\}.E[\lambda x.e]$$

Therefore, the function P not only counts the number of redexes in a term, but also takes into account *potential* redexes that may be introduced in reductions such as this. Again, the details can be found in the extended report [4]. ∎

3.2 Call-by-value/Pass-by-reference

Pascal, Fortran and other languages 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. Our semantics generalizes pass-by-reference and allows arbitrary expressions as arguments to procedures. If 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.

Semantics

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

$$\begin{array}{lll} \rho\theta.E_v[x] & \rhd & \rho\theta.E_v[v] \qquad (E.D'_v) \\ & & \text{if } \theta(x) = v, \text{ and} \\ & & E_v \neq E'_v[(\lambda y.e)[\,]] \end{array}$$

The pass-by-reference transformation function, $\rhd_{vr}$ is the union of the two new rules and previously defined rules in Figure 1:

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

The semantic function $eval_{vr}$ and observational equivalence relation $\simeq_{vr}$ are defined in the usual way.

Calculus

The first difference between the by-value and by-reference parameter passing systems is that the natural derivation of the calculus from the semantics (as specified in Section 2) results in an unsound theory. Specifically, the variable dereference relation $E.D'_v$ is an unsound axiom with respect to the observational equivalence relation. However, since this is only the case when the variable occurs in the empty context, adding a restriction solves the problem:

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

With this axiom replacing $E.D'_v$, the derivation of the λ_v-R-calculus proceeds as usual by constructing the appropriate equivalence relation from the modified transformation function and the notion of reduction **gc**.

Because of the soundness problem encountered with the $E.D'_v$ 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, i.e., the calculus can only evaluate the programs to an expression that is "one step away" from the answer by the program transformation function. Based on this observation, we prove the following *weak* correspondence theorem.[5]

Theorem 3.3 *The λ_v-R-calculus "weakly" corresponds to the call-by-value/pass-by-reference semantics of IS_v, $\boldsymbol{\lambda}_v$-$\mathbf{R} \models \mathrm{WCorr}(eval_{vr})$. In particular,*

1. *$\boldsymbol{\lambda}_v$-$\mathbf{R}$ is "almost" adequate:*
   (a) *if $eval_{vr}(e) = a$ then either*
      - *$\boldsymbol{\lambda}_v$-$\mathbf{R} \vdash e = a$, or*
      - *$\boldsymbol{\lambda}_v$-$\mathbf{R} \vdash e = \rho\theta \cup \{(x, v)\}.x$ and $\boldsymbol{\lambda}_v$-$\mathbf{R} \vdash \rho\theta \cup \{(x, v)\}.v = a$.*

   (b) *if $\boldsymbol{\lambda}_v$-$\mathbf{R} \vdash e = a$, then $eval_{vr}(e)$ is defined.*
2. *$\boldsymbol{\lambda}_v$-$\mathbf{R}$ is sound with respect to $\simeq_{vr}$:*

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

The imperative fragment of the pass-by-reference calculus, $\boldsymbol{\rho}'_v$, as well as its procedure call reduction, β_r, are strongly normalizing for the full language.

Theorem 3.4 *Let $\mathbf{s}' = \delta_c \cup D'_v \cup \sigma_v \cup \rho_\cup \cup \mathbf{gc}$. Then, $\mathbf{s}' \models \mathrm{SN}$ and $\beta_r \models \mathrm{SN}$.*

3.3 Call-by-value-result

The programming languages Ada [23] and Algol W [24] 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.

[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, existing programming languages do not use this notion of call-by-value/pass-by-reference semantics.

[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.

Semantics

The sequential nature of the call-by-value-result protocol requires the specification of a *sequence* statement for IS_v, $\langle e; e\rangle$, and an appropriate extension of evaluation contexts:

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

The expression $\langle e_1; e_2; \ldots e_n\rangle$ abbreviates $\langle e_1; \langle e_2; \ldots e_n\rangle\rangle$. The semantics of the sequence expression is straightforward; the left sub-expression is evaluated first, followed by the right:

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

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 result of the application is the value obtained from evaluation of the procedure body. We use the sequence expression to specify this series of events:

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

The variable r is a new variable that holds the return value of the procedure. The value $\lambda x.x$ is an arbitrary initial value. The complete rewriting function relies on these and previously defined rules:

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

As usual, the transitive closure of $\rhd_{cc}$ and the notion of reduction **gc** determine $eval_{cc}$.

Calculus

The equational calculus for the call-by-value-result language closely resembles the $\boldsymbol{\lambda}_v$-**R**-calculus for the call-by-value/pass-by-reference language. The problems encountered with the dereference axiom in the previous section also occur with the call-by-value-result calculus. The same D'_v axiom solves the problem.

The call-by-value-result calculus only "weakly" corresponds to the semantics in the same way as the call-by-value/pass-by-reference calculus.

Theorem 3.5 $\boldsymbol{\lambda}_v$-**VR** $\models$ 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* reduction. With this addition, the Strong Normalization Theorem for the notion of reduction $\mathbf{s}'$ also holds for $\mathbf{s}' \cup seq$.

3.4 Call-by-name/Pass-by-reference

The Revised Report on Algol 60 [18] informally defined the semantics of call-by-name parameter-passing with the *copy rule*. The intention was to implement Church's [3] β relation from the original λ-calculus as the parameter-passing technique of the new language. Copying an argument to every parameter occurrence was thought to be the computational equivalent to function evaluation in ordinary mathematics. A hidden motive may have been the desire to perceive Algol as an extension of the λ-calculus as a language as well as an equational theory.

The requirements of the copy rule imply a pass-by-reference semantics. If the formal parameter is assigned, then the procedure argument should evaluate to a variable. Assignments to the parameter become assignments to this variable argument, and these effects on the argument will be visible in the context of the procedure application.

Semantics

The term language for the call-by-name/pass-by-reference language, IS_n, differs from IS_v in two ways. First, to exploit the freedom of the delayed evaluation strategy, ρ-expressions can bind variables to arbitrary expressions:

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

Second, since 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 is the same as IS_v, with $\lambda x.e$ representing a call-by-name/pass-by-reference procedure.

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. The definitions of the evaluation contexts and the program transformation function are given in Figure 2.

Remark. Although some of the relations in Figure 2 have the same names as relations introduced for the call-by-value language, there are differences due to change in the definition of evaluation contexts. Relations with the same name share reduction *schema*. The textual context will disambiguate all uses of these names. **End.**

The transformation function uses only the $E.\beta$ rule for defining the behavior of all procedure applications.

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

$$\begin{array}{rcll}
\rho\theta.E_n[f v] & \rhd_{nr} & \rho\theta.E_n[\delta(f,v)] & (E.\delta) \\
\rho\theta.E_n[(\lambda x.e)e'] & \rhd_{nr} & \rho\theta.E_n[e[x \leftarrow e']] & (E.\beta) \\
\rho\theta \cup \{(x,e)\}.E_n[x] & \rhd_{nr} & \rho\theta \cup \{(x,e)\}.E_n[e], \text{ if } E_n \neq E'_n[(\mathbf{set!}\ [\,]\ e')] & (E.D_n) \\
\rho\theta \cup \{(x,e)\}.E_n[(\mathbf{set!}\ x\ v)] & \rhd_{nr} & \rho\theta \cup \{(x,v)\}.E_n[v] & (\sigma_n) \\
\rho\theta.E_n[\rho\theta'.e] & \rhd_{nr} & \rho\theta \cup \theta'.E_n[e] & (\rho_\cup)
\end{array}$$

FIG. 2: Call-by-name/Pass-by-reference Semantics

The restriction on the $E.D_n$ rule is necessary to disallow dereference of a variable when it occurs in the variable position of an assignment. The semantics $eval_{nr}$ is determined by the transitive closure of $\rhd_{nr}$ and **gc** as in previous sections.

Calculus

The derivation of a calculus from the semantics follows the same course as the call-by-value/pass-by-reference language, but the natural approach results in an unsound system. Again, the dereferencing rule causes the problem. Adding an appropriate condition to the dereference rule solves the problem:

$$\begin{array}{rcll}
\rho\theta.E_n[x] & \longrightarrow & \rho\theta.E_n[e] & (D_n) \\
& & \text{if } \theta(x) = e, & \\
& & E_n \neq E'_n[(\mathbf{set!}\ [\,]\ e')], & \\
& & \text{and } E_n \neq [\,] &
\end{array}$$

We then derive the calculus by constructing the appropriate equivalence relation from a modified transformation function in which D_n replaces $E.D_n$.

The call-by-name/pass-by-reference and call-by-value/pass-by-reference languages have several similarities. Both require a modified dereference rule to prohibit variable dereference in certain positions. Similarly, the direct derivation of a calculus from the semantics results in an unsound system. 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 a value and thus an answer, whereas in the call-by-name language it does *not*. As a result, 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.[7]

Theorem 3.6 *The λ_n-R-calculus only* very 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:*

   $$\begin{array}{lcl}
\boldsymbol{\lambda}_n\text{-}\mathbf{R} & \vdash & e = \rho\theta_1 \cup \{(x_1, e'_1)\}.x_1 \\
\boldsymbol{\lambda}_n\text{-}\mathbf{R} & \vdash & \rho\theta_n \cup \{(x_n, e'_n)\}.e'_n = \\
& & \rho\theta_{n+1} \cup \{(x_{n+1}, e'_{n+1})\}.x_{n+1} \\
\boldsymbol{\lambda}_n\text{-}\mathbf{R} & \vdash & \rho\theta_m \cup \{(x_m, e'_m)\}.e'_m = a
\end{array}$$

   (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 $\simeq_{nr}$:*

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

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

Theorem 3.7 *The notion of reduction $\mathbf{s}_n = \delta_c \cup D'_n \cup \sigma_n \cup \rho_\cup \cup \mathbf{gc}$ is strongly normalizing over the subset of IS_n in which the ranges of ρ-sets are restricted to values.*

Finally, as intended by the designers of Algol, the λ_n-R-calculus is indeed a conservative extension of the λ-calculus.

Theorem 3.8 $\boldsymbol{\lambda} = \boldsymbol{\lambda}_n\text{-}\mathbf{R}|\Lambda$

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

[7] As above, changing the semantics would fix the problem, too: this time *thunks* would have to be valid answers. But again, existing languages are *strict* at the top level.

Syntax, IS_b**:**

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

Semantics:

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

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

FIG. 3: Reference Cell Language

3.5 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 C, ML and Scheme do not use pass-by-reference, but instead have a new class of values, namely, *pointers*, *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. When a cell occurs as the argument to a procedure, the application binds the formal parameter to the cell because a cell is a value. In this way, an assignment to the formal parameter becomes visible in the context of the procedure application, as in pass-by-reference. Although the reference cell concept is not a proper parameter-passing technique, we consider it here because of its relation to both pass-by-worth and pass-by-reference parameter-passing.

Semantics

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 creating, assigning and dereferencing cells. Adopting Chez Scheme terminology [7], we use the primitives **box**, **setbox!**, and **unbox** to perform these operations in IS_b. The first part of Figure 3 contains the specification 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.

The evaluation contexts and the transformation function are also defined in Figure 3. 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.

Calculus

We derive the λ_v-B-calculus for reasoning with reference cells directly from the semantics by generating an equivalence relation from the transformation function and the garbage collection reduction as in previous sections. The resulting theory, $\boldsymbol{\lambda}_v$-**B**, corresponds to the semantics, $eval_{bx}$:

Theorem 3.9 $\boldsymbol{\lambda}_v$-**B** $\models \mathrm{Corr}(eval_{bx})$

Furthermore, the imperative fragment of this calculus, $\boldsymbol{\rho}_b$, is decidable:

Theorem 3.10 *Let* $\mathbf{s}_b = \delta_c \cup D_b \cup \sigma_b \cup \rho_\cup \cup \mathbf{gc}$*. Then,* $\mathbf{s}_b \models \mathrm{SN}$.

Finally, 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 3.11 $\boldsymbol{\lambda}_v = \boldsymbol{\lambda}_v$-$\mathbf{B}|\Lambda$

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, it is possible to show that IS_b and IS_v are equivalent in the formal sense of programming language expressiveness [8]. For the pass-by-reference language, the connection is not as clear, and we have no formal expressibility results. However, reference cells in the pass-by-worth language provide the ability to abstract over assignments to formal arguments (cells in this case), 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.

4 Related Work

Our equational systems were motivated by the work of Plotkin [20], who studied equational reasoning systems for call-by-name and call-by-value in functional languages, and Felleisen et al [9, 10] who developed calculi for call-by-value *imperative* languages. Demers and Donahue [5] give an equational logic for reasoning about a higher-order language that uses call-by-value parameter-passing and has memory objects similar to reference cells. The equational theory contains several dozen axioms for which they present no formal results. Mason and Talcott [16] present a conditional deduction system for observational equivalences of Lisp expressions with side-effects, a language similar to the imperative fragment of our reference cell language. Their logic is complete for the recursion-free, first-order fragment of Lisp with side-effects.

In the area of denotational semantics, several authors [11, 22] have given denotational descriptions of the different parameter-passing techniques. Such descriptions specify precise set-theoretic interpretations of parameter-passing but do not provide an intuitive and simple set of equations nor other axiomatic theories.

Finally, a number of researchers have studied parameter-passing in the context of Hoare-like axiomatic semantics [14, 12, 6]. 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 [19] eliminates the restrictions on procedures, but his Hoare-like calculi explicitly require operational specifications of procedure calls using the copy rule.

5 Conclusions

For several common parameter-passing techniques we presented a program rewriting semantics and derived a corresponding calculus. The calculi of pass-by-value languages are simply the equational theories over the rewriting semantics but the calculi for pass-by-reference languages are weaker. In the latter cases, the rewriting rule for dereferencing variables is not sound with respect to observational equivalence. As a result, the correspondence between the calculi and the semantics is weaker for pass-by-reference languages than for pass-by-value languages. All of the calculi are sound but the calculi for pass-by-reference are too weak to serve as a substitute for an evaluator. The underlying reduction systems of the calculi are always Church-Rosser and have the standard reduction property. Finally, the imperative fragments of most calculi are strongly normalizing and therefore decidable.

Table 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. Although this study verifies the folklore that Algol's call-by-name/pass-by-reference parameter-passing technique satisfies the full β axiom, it also reveals that this comes at the expense of a weak correspondence relationship and of strange restrictions on the axioms for the imperative fragment of the language. Indeed, call-by-value/pass-by-reference as well as copy-in-copy-out share these problems. Of the two remaining combinations, call-by-value/pass-by-worth with boxes has the better characteristics:

1. the calculus is the equational closure of a reduction semantics;
2. because of (1), there is a strong correspondence between the semantics and the calculus;
3. the axioms are simple, without complicated restrictions;
4. *and* the calculus is a conservative extension of the functional core language.

Because of the above properties, using call-by-value with boxes and reasoning about programs in such language seems the most attractive choice.

Evaluation Strategy	Binding Strategy	Correspondence	Beta Axiom	Param	Arg	Decidable Fragment
call-by-value	worth	exact	β_v	non-assign	value	$\boldsymbol{\rho}_v$
	reference	weak	β_r	any	variable	$\boldsymbol{\rho}'_v ; \beta_r$
	value-result	weak	β_c	–	–	$\boldsymbol{\rho}'_v \cup seq$
call-by-name	reference	very weak	β	any	any	restricted $\boldsymbol{\rho}_n$
call-by-value & reference cells		exact	β_v	any	value	$\boldsymbol{\rho}_b$

Table 1: Summary

Acknowledgement. We thank Amr Sabry and Rebecca Selke for comments on an early draft.

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. Crank, E. Parameter-passing and the lambda-calculus. Master's Thesis, Rice University, August 1990.
5. 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.
6. Donahue, J.E. *Complementary Definitions of Programming Language Semantics.* Lecture Notes in Computer Science 42, Springer-Verlag, Heidelberg, 1980.
7. Dybvig, R. K. *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
8. 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. Springer Verlag, Berlin, 1990, 134–151.
9. 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.
10. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989.
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. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.
16. 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, 284–293. An extended version will appear in *Theor. Comput. Sci.*
17. Muller, R. The operational semantics and equational logic of eval and fexprs. Unpublished manuscript. Harvard University, 1990.
18. Naur, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM* **6**(1), 1963, 1–17.
19. Olderog, E. Sound and complete Hoare-like calculi based on copy rules. *Acta Inf.* **16**, 1981, 161–197.
20. Plotkin, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.
21. Rees, J. and W. Clinger (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.
22. Schmidt, D.A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, Mass., 1986.
23. US Department of Defense. *The Programming Language Ada—Reference Manual*, Lecture Notes in Computer Science 106, Springer-Verlag, 1981.

24. Wirth N. and C.A.R. Hoare. A contribution to the development of ALGOL. *Commun. ACM* **9**(6), 1966, 413–432.