A Lazy SECD Machine

by

Arthur F. Lent

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1990

Signature of Author ....
Department of Electrical Engineering and Computer Science
January 24, 1990

Certified by ....
Albert R. Meyer
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by..
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

A Lazy SECD Machine

by

Arthur F. Lent

Submitted to the Department of Electrical Engineering and Computer Science
on January 24, 1990, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

In many cases, when an operational semantics is defined for a functional language the definition is in the form of an abstract set of rewrite rules which specify how, at a high level, complex expressions can be transformed into simpler ones. These rewrite rules, however, are not practical for forming the basis of an interpreter for these languages. The SECD machine approach [2] provides a more easily- and efficiently- implementable operational semantics. This paper describes a SECD formalization for call-by-name functional languages with any (suitable) set of constants. As an example of the generality of the model, we give an SECD implementation of the simply-typed aritmetic language PCF [4].

Thesis Supervisor: Albert R. Meyer
Title: Professor of Computer Science and Engineering

Contents

Chapter 1

Introduction

Landin [2] introduced the SECD machine as a formal model for interpreting functional languages. The historical importance of Landin's definition lies in the level of abstraction—the SECD model makes explicit the control structure of evaluation at a low level, while still leaving irrelevant, machine-dependent, details unspecified. Several others have presented SECD machines since Landin, most notably Plotkin [3] and Henderson [1].

As originally presented, the SECD machine interprets functional code in a *call-by-value* manner. In call-by-value, all arguments to functions are evaluated exactly once, at function application. There are two advantages to call-by-value over *call-by-name* (in which arguments are evaluated when they are used). First, in the presence of side effects (which most practical languages allow to some extent), it is easier to reason about call-by-value than call-by-name. Second, there is a notion that the representation of evaluated objects is generally more compact than that of unevaluated objects, resulting in improved space efficiency for call-by-value. Since the SECD machine has been primarily used to suggest an implementation strategy for functional languages, and since most languages are call-by-value, it is only natural that the primary work done with SECD machines has been for call-by-value languages.

Call-by-value does extra work if a function never uses its argument. In a call-by-name interpretation, arguments are evaluated only if they are used. Although efficiency may be important, the most significant distinction can be discovered when the evaluation of an argument M either does not halt, or results in an error. Consider a function FOO

which does not use its argument. Now suppose we apply *FOO* to *M*. In a call-by-name implementation, *FOO* applied to *M* can still evaluate to a value, whereas in a call-by-value implementation it will not. So the key difference between these two schemes is that all side-effect-free functional programs that terminate with a value in the call-by-value framework will also terminate with an equivalent value in the call-by-name framework, but not vice versa.

This thesis presents a call-by-name analog to the SECD machine of Plotkin [3]. In addition, we show that the operational behavior of the machine presented (with an appropriate selection of constants) corresponds with the operational definition of PCF via rewrite rules [4]. Both definitions are useful in understanding the operational behavior or a language. The rewrite rules are typically easier for a person to reason about; however, it is not at all clear how to build an interpreter based upon a set of rewrite rules. On the other hand, it is quite easy to build an interpreter based upon a SECD machine description, but quite difficult to reason directly about the description. Consequently, it is useful to provide both a description of a language via the SECD model and via rewrite rules, and then to prove that reasoning in one system appropriately mirrors reasoning in the other.

Chapter 2

Definitions

These definitions are very similar to those presented by Plotkin [3].

2.1 Terms in λ-calculus

The set of λ-calculus terms is determined by a set of variables $x, y, z, \ldots$ and a set of constants $a, b, \ldots$, is defined inductively by:

- Any variable is a term.
- Any constant is a term.
- If x is a variable and M is a term then $(\lambda x M)$ is a term.
- If M and N are terms, so is (MN).

A term of the form $(\lambda x M)$ is an *abstraction*; one of the form (MN) is a *combination*. A term is a *value* iff it is a constant or an abstraction. We establish the convention that $M = N$ means that M and N are identical terms.

In general the set of variables will be infinite, although the set of constants need not be. We establish the following naming conventions which will be used throughout:

- Lowercase letters (at the end of the alphabet) x, y, z—always are variables.
- Lowercase letters (at the beginning of the alphabet) a, b, c—always are constants.

- Capital letters (in the middle of the alphabet) L, M, N—always are terms.

The *free variables* of M, $FV(M)$, is defined inductively by:

$$FV(x) = \{x\};\ FV((MN)) = FV(M) \cup FV(N);\ FV((\lambda x M)) = FV(M) \backslash \{x\}.$$

The *bound variables* of M, $BV(M)$ is defined similarly.

A term is *closed* iff $FV(M) = \emptyset$, otherwise it is *open*. The *substitution prefix* is defined by:

$$[M/x]x = M;\ [M/x]y = y \text{ (if } x \neq y);$$
$$[M/x]a = a;$$
$$[M/x](NN') = ([M/x]N)\,([M/x]N');$$
$$[M/x](\lambda x N) = (\lambda x N);\ [M/x](\lambda y N) = \lambda z[M/x][z/y]N,\ \text{(if } x \neq y);$$

where z is a "fresh" variable not appearing in M or N.

2.2 Other notational conveniences

The α-equivalence relation, $=_\alpha$, of is defined inductively by:

- $x =_\alpha x$ and $a =_\alpha a$.
- If $M =_\alpha M'$ and $N =_\alpha N'$ then $(MN) =_\alpha (M'N')$.
- If $M =_\alpha [x/y]M'$, where either $x = y$ or $x \notin FV(M')$ then $(\lambda x M) =_\alpha (\lambda y M')$.

It captures the notion of syntactic equality of terms up to the renaming of bound variables.

The symbol *nil* will be used for the empty sequence and : for concatenation. X^* is the set of sequences of members of a set X.

Given sets X and Y, $(X \xrightarrow{P} Y)$ is the set of partial functions from X to Y. If $f \in (X \xrightarrow{P} Y)$, $Dom(f)$ is its domain, that is,

$$Dom(f) = \{x \in X \mid [x, y] \in f, \text{ for some } y \in Y\}$$

Expressions using partial functions are defined iff the functions are defined at their given arguments. They are equal ($=$) iff they are both undefined, or are both defined and have the same value. They are alpha equivalent ($=_\alpha$) under similar conditions.

Given a relation $\rightarrow$, $\xrightarrow{n}$ is its n th power ($n \geq 0$), $\xrightarrow{+}$ its transitive closure, and $\xrightarrow{*}$ its reflexive, transitive closure.

2.3 Closures and Environments

We do not consider the operation of syntactically substituting terms for several occurrences of a variable a primitive operation. Through the use of closures and environments, however, we can implement substitution "symbolically", using pointer manipulations rather than really doing the syntactic manipulation. We now define closures, environments and their depths inductively by:

1. $\emptyset$ denotes the empty environment, and has depth 0.
2. If $x_1, \ldots, x_n$ are distinct variables and $Cl_i (i = 1 \ldots n)$ are closures whose maximum depth is d, then $\{[x_i, Cl_i] | i = 1 \ldots n\}$ is an environment of depth $d + 1$.
3. If E is an environment with depth d, and M is a term such that $FV(M) \subseteq Dom(E)$, then $[M, E]$ is a closure of depth d.

$E\{Cl/x\}$ is the unique environment E' such that $E'(y) = E(y)$, if $y \neq x$ and $E'(x) = Cl (Cl \in \text{Closures})$. A closure $[M, E]$ is a *value closure* iff M is a value term (note: in this framework we do **not** consider a variable to be a value.

The function Real: Closure $\rightarrow$ Terms is defined inductively by:

$$\text{Real}([M, E]) = [Real(E(x_1))/x_1] \ldots [Real(E(x_n))/x_n]M,$$

where

$$FV(M) = \{x_1, \ldots, x_n\}.$$

This yields the term "represented" by a closure. So, in summary, if M is a term such that $FV(M) = \{x\}$, and Cl is a closure representing N we can represent $[N/x]M$ by the closure $[M, \emptyset\{Cl/x\}]$.

2.4 The Language Defined, Constants, and Constapply

We will use our SECD machine to specify an evaluation function Eval: Programs $\xrightarrow{P}$ Programs. When fully specified, Eval provides an operational method of giving meaning to all of the terms in a programming language. Part of the specification of Eval is given relative to an interpretation of the constants. The function

$$\text{constapply: Constants} \times \text{Closed Values} \xrightarrow{P} \text{Closed Terms.}$$

gives this interpretation. We wish some operators to work on λ-abstractions, so we do not choose constapply in: (Constants $\times$ Constants) $\xrightarrow{P}$ Closed Values as Plotkin does.

Unfortunately, since constapply ranges over λ-terms, our machine may be in a state where it has a representation for a perfectly good λ-term for constapply, but it has stored it in the form of a closure. Consequently, we need a Constapply for the machine which is determined to within $=_\alpha$ by constapply.[1]

$$\text{Constapply: Constants} \times \text{Value Closures} \xrightarrow{P} \text{Closures,}$$

where Constapply must obey the restriction:

$$\text{Real}(\text{Constapply}(a, Cl)) =_\alpha \text{constapply}(a, \text{Real}(Cl))$$

To remain in the SECD spirit, Constapply should be chosen in such a way as to be implementable using a bounded number of pointer manipulations based upon looking a bounded depth into its arguments.

We stated that we were defining a machine that was *call-by-name*, yet in this setup arguments to constant operators are **always** evaluated, hence constant applications are **call-by-value.** Because some constant applications **must** be call-by-value (such as +1), we claim that this is a reasonable decision. The only alternative is to partition the set of constants into a group of "non-strict" constants which never have their arguments evaluated, and a group of "strict" constants which always have their arguments evaluated. In fact, it would be a fairly straightforward modification to the machine to allow both strict and

---

[1] Note the distinction between constapply and Constapply. The function constapplyis that used at the level of rewrite rules, whereas the function Constapply will be used for the SECD machine defined in the next chapter.

non-strict constant operators, resulting in minor changes in the definition of Eval and in the proofs of Lemmas 2 and 3. Our restriction to strict constant operators does not limit the constructs definable in any given language. One can see this by considering the kinds of constant operators that need to be strict. In every case there is a simple alternative which operates in the desired way. Consider the following examples:

1. An operator that ignored its first argument and then did M. But this is definable by $(\lambda x M)$ (for an $x \notin FV(M)$).

2. An operator that might or might not evaluate its first argument depending on its later arguments. But this is definable by something of the form $M_1 = (\lambda x(\lambda y((My)x)))$. An expression such as $((M_1 M_2)M_3)$ will reduce to something of the form $M' M_2$ where $M' = (M M_3)$. Based upon the value of M_3, M can construct a λ-term that either might use M_2 (a term of the form $(\lambda x M'')$, where x is free in M'') or one that will definitely not use M_2, where x is *not* free in M''. Since our language is call-by-name this will work even if the evaluation of M_2 does not terminate (M_2 *diverges*).

More complex operators can be constructed analogously. For more details examine the implementation of $\supset$ in the following example.

Example 1. Consider what the constants and constapply would be for PCF proper—we will basically ignore issues of type-checking and assume all programs are type correct. We will, however, assign types to all of our constants in the language. Our base (ground) types are o (for boolean values) and ι (for natural number values). Our higher types are of the form $(\sigma \rightarrow \tau)$, where σ and τ are types and $(\sigma \rightarrow \tau)$ represents an appropriate subset of the functions form objects of type σ to objects of type τ.

The constants, together with their types are:

$tt : o$,

$ff : o$,

$\supset_\iota : (o \rightarrow \iota \rightarrow \iota \rightarrow \iota)$,

$\supset_o : (o \rightarrow o \rightarrow o \rightarrow o)$,

$Y_\sigma : ((\sigma \rightarrow \sigma) \rightarrow \sigma)$ (one for each σ),

$k_n : \iota$ (one for each integer $n \geq 0$),

$(+1) : (\iota \rightarrow \iota)$

$(-1) : (\iota \rightarrow \iota)$

$(Z) : (\iota \rightarrow o)$

The partial function constapply is defined as follows:

$\supset_\sigma$(σ **ground**)	constapply($\supset_\sigma, tt$)	=	$(\lambda x^\sigma \lambda y^\sigma x)$	
	constapply(($\supset_\sigma, ff$)	=	$(\lambda x^\sigma \lambda y^\sigma y)$	
(+1)	constapply($(+1), k_m$)	=	k_{m+1}	$(m \geq 0)$
(−1)	constapply($(-1), k_{m+1}$)	=	k_m	$(m \geq 0)$
Z	constapply(Z, k_0)	=	tt	
	constapply(Z, k_{m+1})	=	ff	
$\mathbf{Y}_\sigma$	constapply(Y_σ, V)	=	$(V(Y_\sigma V))$	V is a value

An acceptable definition of Constapply that fits with both the restriction that

$$\text{Real}(\text{Constapply}(a, Cl)) =_\alpha \text{constapply}(a, \text{Real}(CL))$$

and the restriction that it be "easily implementable" is as follows:

$\supset_\sigma$(σ **ground**)	Constapply($\supset_\sigma, [tt, E]$)	=	$[(\lambda x^\sigma \lambda y^\sigma x), \emptyset]$	
	Constapply(($\supset_\sigma, ff$)	=	$[(\lambda x^\sigma \lambda y^\sigma y), \emptyset]$	
(+1)	Constapply($(+1), [k_m, E]$)	=	$[k_{m+1}, \emptyset]$	$(m \geq 0)$
(−1)	Constapply($(-1), [k_{m+1}, E]$)	=	$[k_m, \emptyset]$	$(m \geq 0)$
Z	Constapply($Z, [k_0, E]$)	=	$[tt, \emptyset]$	
	Constapply($Z, [k_{m+1}, E]$)	=	$[ff, \emptyset]$	
$\mathbf{Y}_\sigma$	Constapply($Y_\sigma, [V, E]$)	=	$[(V(Y_\sigma V)), E]$	([V,E] is a value closure)

Note that $\supset_\sigma$ occurs in a curried form rather than that originally presented for PCF by Plotkin [4]. It is an easy task, however, to show that this does not alter the language defined. A proof that this selection of Constants and constapply together define an Eval and eval such that Eval(M) = eval(M) = PCF(M) [2] for all programs [3] M appears at the end of the paper in Chapter 4.

2.5 Stacks, Controlstrings, and SEC state

Controlstrings = $(Terms \cup \{ap, ct\})^*$, where $ap \notin Terms$, and Stacks = Closures*.

The function FV is extended to Controlstrings by:

$$FV(ap) = \emptyset; \quad FV(C_1, \ldots, C_n) = \bigcup_{i=1}^{n} FV(C_i) \quad (n \geq 0).$$

A state Q of the SEC machine is a triple $[S, E, C]$ with S a Stack, E an environment and C a controlstring such that $FV(C) \subseteq Dom(E)$.

[2]We write PCF(M) to denote the evaluation function defined with respect to the rewrite rules for PCF. It is written as $\text{Eval}_{\mathcal{C}}$ by Plotkin.

[3]A closed term of gound type is called a *program*.

Chapter 3

The SEC Machine

Landin, Plotkin and Henderson all presented call-by-value SECD machines. As the name implies the all used a stack, environment, a controlstring, and a dump. In this thesis we have given technical definitions of stacks, environments and controlstrings. We have not, however, defined dumps. This is unnecessary for us, as it turns out that in the call-by-name case the dump is unnecessary—thus an "SEC" machine results. Before we give the full definition of the automaton we discuss how the stack, environment, and dump are used:

Stack The stack is used to store intermediate results during evaluation.

Environment The environment is the environment in which the top item on the controlstring is being evaluated.

Controlstring The controlstring contains whatever instructions are needed in order to complete the evaluation. The primary branch in deciding which rule to apply is based upon the top item on the controlstring.

The primary method of operation of the SEC machine is as follows:

$$[S, E, M : C] \overset{*}{\Rightarrow} [Cl : S, E', C]$$

for all stacks S, environments E such that $FV(M) \subseteq Domain(E)$, and controlstrings C. The point is that the term represented by Cl is the result of evaluating the term represented by $[M, E]$. This notion of evaluation will be made precise by the definition of eval in section 3.2. The main result of this chapter will be to prove the following theorem:

Theorem 1. For all closed terms M there is a value closure Cl such that:

$$[nil, nil, N] \overset{*}{\Rightarrow} [Cl, E, nil]$$

iff eval(Real$[M, \emptyset]$) exists, moreover, Real$(Cl) =_\alpha$ eval(Real$[M, \emptyset]$).

3.1 The Transition Function $\Rightarrow$ and Eval

The transition function $\Rightarrow$, in States $\xrightarrow{P}$ States, is defined by:

(1)	$[S, E, a : C]$	$\Rightarrow$	$[[a, \emptyset] : S, E, C]$
(2)	$[S, E, x : C]$	$\Rightarrow$	$[S, E', M : C]$
			where $E(x) = [M, E']$
(3)	$[S, E, \lambda x M : C]$	$\Rightarrow$	$[[\lambda x M, E] : S, E, C]$
(4)	$[[(\lambda x M), E'] : Cl : S, E, ap : C]$	$\Rightarrow$	$[S, E'\{Cl/x\}, M : C]$
(5)	$[[a, E''] : [V, E'''] : S, E, ct : C]$	$\Rightarrow$	$[S, E', M' : C]$
			(where Constapply$(a, [V, E''']) = [M', E']$)
			(and V is a value)
(6)	$[[a, E'] : [N, E''] : S, E, ap : C]$	$\Rightarrow$	$[S, E'', N : a : ct : C]$
			(where N is not a value)
(7)	$[S, E, (MN) : C]$	$\Rightarrow$	$[[N, E] : S, E, M : ap : C]$

We now justify each of the rules:

1. A constant should evaluate to itself, so it is put on the top of the stack in an appropriate closure.

2. A variable x in environment E should evaluate to whatever $E(x)$ evaluates to. Install $E(x)$ and let it evaluate.

3. A λ-abstraction evaluates to itself, but, since its body might have free variables we need to keep the same environment.

4. The object we are in the process of evaluating was a combination. Since this is call-by-name, the argument was placed on the stack unevaluated. The operator has evaluated to a λ-abstraction. Perform the appropriate β-reduction using environments and closures to do the substitution.

5. The object we are in the process of evaluating was a combination. The operator has evaluated to a constant. Since constant applications are call-by-value, we also need to evaluate its argument—which has already been done. We use Constapply to do the application, then we need to evaluate the result.

6. The object we are in the process of evaluating was a combination. The operator has evaluated to a constant. Since constant applications are call-by-value, we need to evaluate the argument before we can use Constapply.

7. In order to evaluate an application in the call-by-name framework, evaluate the operator, leaving the argument unevaluated. After the operator is evaluated, if it is a λ-abstraction, do a β-reduction without evaluating the argument. If the operator is a constant, evaluate the argument, use Constapply, then evaluate that result.

We will use Load to initialize the SECD machine to evaluate a closed term, and use Unload to extract the result from a "halted" SECD state. They are defined as follows:

$$\text{Load}(M) = [nil, \emptyset, M]$$

$$\text{Unload}([Cl, E, nil]) = \text{Real}(Cl)$$

We can now define the automaton's evaluation function by:

$$\text{Eval}(M) = N \text{ iff } \text{Load}(M) \overset{*}{\Rightarrow} Q, \text{ and } N = \text{Unload}(Q) \text{ for some state } Q.$$

Now that we have defined our automaton, it is important to show that it really has the properties which we were looking for in a model. We wanted each step to be realizable by a bounded number of pointer manipulations, based upon looking a constant depth into the expression being evaluated.

The parse tree can be constructed in such a way that any subterm can be fully represented by a pointer to a node in the tree. All decisions for our machine (which rule to apply, what is the value of Constapply) can be based upon examining the parse tree with a small, constant number of pointer manipulations. We now consider the "boundedness" of the actions of the machine.

1. Copying of terms is done by sharing objects that were created at the beginning of program execution (small, constant number of operations).

2. Copying of environments is done by sharing objects that were created during execution (single operation).

3. Modifying environments is done by appending a pair (the variable name, paired with the closure to which it is being bound) on the front of the environment to be modified, and providing a pointer to the new environment with the added pair in front (small, constant number of operations).

4. Variables may be looked up via a sequential search through the environment, but the maximum number of variables in an environment is a syntactic property, detectable by the parser. (It is the maximum nesting depth of a term in the program—the number of syntactically enclosing λ's).

It is easy to see that the first three of the preceding statements are true. The fourth, however, requires additional justification. In particular, it is necessary to examine the claim that the maximum number of variables in an environment is dependent only on the structure of the term initially loaded into the machine, and is independent of the number of steps needed to evaluate M. The key observation is that a subterm N_i of $M = (N_1 N_2)$ will always be evaluated with respect to the environment in which M was first encountered (due to rule 7). Why? We can have 3 cases:

1. N_1 is the subterm, trivial from rule 7.

2. N_2 is the subterm, then a closure $[N_2, E]$ is created—where E is the environment in which $(N_1 N_2)$ resides.

There is now no way to evaluate N outside of E unless Constapply does some syntactic rearranging in its arguments that "buries" terms inside λ's (e.g. turn N into $(\lambda x N)M$). So for any definition of Constapply, it is important to check that the constant operators do not cause any problems. In the case of the example given in Section 2.4, Y is the only operator that may cause a problem. It does not: consider (YV) in environment E (the closure $[(YV), E]$). We then get $(V(YV))$ in environment E, which is V in environment E and $[(YV), E]$ which is the closure at which we started.

For this scheme, all actions of the machine can be implemented by a constant number of pointer manipulations—except variable lookup. The scheme proposed is one of many implementation strategies, and is presented mainly to illustrate feasibility. There are many alternative strategies, some of which could improve the performance of variable lookup at the expense of other operations.

Henderson [1] uses an alternate approach to the SECD machine. In his model the original code is "compiled" into SECD "machine" code, so the analysis of the structure of a term is done only once—even if the term is evaluated several times (although this provides only marginal benefit over using a good parse tree). In addition, an analysis can be done at compile time to determine at what offset in the environment the value of each variable can be found. This presentation, however, serves the purpose of building a functioning interpreter/compiler for a subset of Lisp. Consequently, Henderson's interest in the implementation results in a model in which some clarity has been sacrificed for implementability.

3.2 eval

Our definition of Eval is quite cumbersome and can be very difficult to reason about formally. We therefore introduce a much more manageable function, eval, which provides an inductive characterization of our language. In the case of PCF, eval can be thought of as a recursive characterization of the rewrite rules. We define eval by first defining the binary relation on closed terms eval_Γ. The relation eval_Γ is defined inductively as follows:

- $\text{eval}_\Gamma(c, c)$ for c a constant.
- $\text{eval}_\Gamma(\lambda x.\ M, \lambda x.\ M)$.
- if $\text{eval}_\Gamma(M_1, \lambda x.M)$ and $\text{eval}_\Gamma([N_2/x]M, L)$, then $\text{eval}_\Gamma((M_1 N_1), L)$
- if $\text{eval}_\Gamma(M_1, c)$, $\text{eval}_\Gamma(N_1, N_2)$ and $\text{eval}_\Gamma(\text{constapply}(c, N_2), L)$, then $\text{eval}_\Gamma((M_1 N_1), L)$

The following two facts about eval_Γ can be proven by induction on its definition:

(a) eval_Γ is the graph of a function.

(b) If $\text{eval}_\Gamma(M, N)$ then N is a value.

Thus we can make the following definition of partial function eval:

$$\text{eval}(M) \stackrel{\text{def}}{=} \text{ the unique } V\text{, if any such that } \text{eval}_\Gamma(M, V)$$

Although we have rigorously defined eval above, one can gain additional insight into the operation of eval by thinking of it in terms of being a function which satisfies the following:

$$\text{eval}(a) = a; \quad \text{eval}(\lambda x M) = \lambda x M$$

$$\text{eval}(MN) = \begin{cases} \text{eval}([N/x]M') & (\text{if eval}(M) = \lambda x M') \\ \text{eval}(\text{constapply}(a, N')) & (\text{if eval}(M) = \text{a and eval}(N) = N') \end{cases}$$

In our proofs we will often find it necessary to do an induction on the proof that $\text{eval}_\Gamma(M, V)$ holds, which we will call induction on the definition of eval_Γ. For convenience we may also say "M evals to V" to mean $\text{eval}_\Gamma(M, V)$. In addition we will use the following fact about how eval and eval_Γ "obey" our intuitions about α-equivalence.

Fact 1. If $M_1 =_\alpha M_2$ then

1. $\text{eval}_\Gamma(M_1, N_1)$ implies that there is an $N_2 =_\alpha M_2$ such that $\text{eval}_\Gamma(M_2, N_2)$.
2. $\text{eval}_\Gamma(N_1, M_1)$ implies that there is an $N_2 =_\alpha M_2$ such that $\text{eval}_\Gamma(N_2, M_2)$.

3.3 Result.

The main result of this thesis is to prove that the Eval function of the SEC machine properly captures the operational semantics of call-by-name. This is done by proving it equivalent to eval—a general recursive characterization of call-by-name without side effects, abstracted away from the specific constants in the language. Given the following theorem, proving the correctness of a given SEC implementation for a specific language becomes merely a matter of showing that eval, with proper constants and definition of Constapply, captures the operational semantics of the target language. Chapter 4 presents the proof for the target language PCF. Proving that the recursive characterization eval correctly captures the target language is much simpler than directly proving the equivalence with the SEC machine itself.

The precise statement of our main theorem is as follows:

Theorem 1. Eval$=_\alpha$eval.

The proof of this theorem is an immediate consequence of the following two Lemmas. The first, Lemma 2, says that if eval$(M)=_\alpha N$ then Eval$(M)=_\alpha N$. The second, Lemma 3 says that if Eval$(M)=_\alpha N$ then eval$(M)=_\alpha N$. Hence, given our notion of equality between partial functions eval$(M)=_\alpha$Eval(M). In other words eval(M) is undefined iff Eval(M) is undefined, and eval(M) is defined and has value N iff Eval(M) is defined and has value within $=_\alpha$ of N.

But, before we can prove these two Lemmas, we need to observe that a variable lookup in an environment takes a number of steps less than the depth of the environment. This is formally expressed by the following Fact:

Fact 2. If E has depth d then $[S, E, x : C] \stackrel{d'}{\Rightarrow} [S, E', M : C]$ where $d' \leq d$ and Real$([M, E'])=_\alpha$Real$([x, E])$ and M is not a variable.

In addition we need a Lemma which demonstrates the correctness of how the SEC machine uses environments and closures to represent the term that results from the substitution of the term N for the variable y inside the term M (which we need in order to capture β-reduction efficiently).

Lemma 1. Suppose $[\lambda y M, E]$ and $[N, E']$ are closures and Real$([\lambda y M, E])=_\alpha(\lambda x M')$ and Real$([N, E'])=_\alpha N'$. Then Real$([M, E\{[N, E']/y\}])=_\alpha[N'/x]M'$.

The proof follows from the observation that if $\lambda y M =_\alpha \lambda x M'$ then $M =_\alpha [y/x]M'$ hence $[N/y]M =_\alpha [N/x]M'$.

We would like to show that eval$(M)=_\alpha N$ implies Eval$(M)=_\alpha N$. This is proven by induction on the definition of eval$_\Gamma$, with the inductive hypothesis strengthened as follows:

Lemma 2. Suppose $[M, E]$ is a closure and eval$_\Gamma$(Real$([M, E], M'')$), then $\forall S, E, C$ with FV(C) $\subseteq$ Dom(E) and some E'' we have:

$$[S, E, M : C] \stackrel{*}{\Rightarrow} [[M', E'] : S, E'', C]$$

where $[M', E']$ is a closure Real$([M', E'])=_\alpha M''$.

Proof: By induction on the proof that $\text{eval}_\Gamma(M'', \text{Real}([M,E]))$. This proof follows very closely along the lines presented by Plotkin[3]. There are 4 main cases:

1. M is a constant c. Here $\text{Real}([M,E]) = M = M'' = c$. Thus $\text{eval}_\Gamma(c,c)$. As

$$[S,E,M:C] \Rightarrow [[M,\emptyset]:S,E,C]$$

we can take $[M',E'] = [M,\emptyset]$ and $E'' = E$.

2. M is an abstraction $(\lambda x.N)$. Here $M'' = \text{Real}([M,E]) = \lambda x.N$, and thus we have $\text{eval}_\Gamma(\lambda x.N, \lambda x.N)$. As

$$[S,E,M:C] \Rightarrow [[M,E]:S,E,C]$$

we can take $[M',E'] = [M,E]$ and $E'' = E$.

3. M is a variable, call it x. By Fact 2,

$$[S,E,x:C] \overset{d'}{\Rightarrow} [S,E',M':C]$$

where M' is not a variable. The proof then breaks down to a case analysis on the structure of M' exactly like this theorem, except the case of M' is a variable cannot occur.

4. $M = (M_1M_2)$ is a combination. Then

$$\text{Real}([M_1M_2,E]) = \text{Real}([M_1,E])\text{Real}([M_2,E]) = N_1N_2 \text{ say.}$$

(a) Suppose $\text{eval}_\Gamma(N_1, \lambda x N_3)$. Then to get $\text{eval}_\Gamma((N_1N_2), M'')$ we must also have $\text{eval}_\Gamma([N_2/x]N_3, M'')$. Then by the induction hypothesis [1] there are E_{d_1}, and $[M_1', E_1']$ such that

$$\begin{aligned}[S,E,(M_1M_2):C] &\Rightarrow [[M_2,E]:S,E,M_1:ap:C] \\ &\overset{*}{\Rightarrow} [[M_1',E_1']:[M_2,E]:S,E_{d_1},ap:C]\end{aligned}$$

[1] When the induction hypothesis is applied to a SEC machine state Q to yield a state of the form $[Cl,E,C]$ and we do not care what E is (*e.g.* we are about to discard it or use it as E'' in proving the induction hypothesis for the next step, where it may be arbitrary) we will write E as E_{d_i} where i is used to distinguish environments when the induction hypothesis is applied several times. When we do care about E having certain properties, it will not be labeled in the form E_{d_i}.

where $\text{Real}([M_1', E_1']) =_\alpha \lambda x N_3$ and $[M_1', E_1']$ is a closure. Here $M_1' = \lambda y M_3'$ for some M_3' and

$$\text{Real}([M_3', E_1'\{[M_2, E]/y\}]) =_\alpha [N_2/x]N_3 \text{ (Lemma 1).}$$

Now,

$$\begin{aligned}
[[M_1', E_1'] : [M_2, E] : S, E_{d_1}, ap : C] &\Rightarrow [S, E_1'\{[M_2, E]/y\}, M_3' : C] \\
&\stackrel{*}{\Rightarrow} [[M', E'] : S, E_{d_2}, C]
\end{aligned}$$

where, by the induction hypothesis, $\text{Real}([M', E'])$ is to within α-equivalence of the value to which $\text{Real}([M_3', E_1'\{[M_2, E]/y\}])$ evals, E_{d_2} is an environment, and $[M', E']$ is closure.

(b) If there is no term of the form $\lambda x.N_3$ such that $\text{eval}_\Gamma(N_1, \lambda x.\, N_3)$ then there must be a constant a such that $\text{eval}_\Gamma(N_1, a)$. In addition, there must be an N such that $\text{eval}_\Gamma(N_2, N)$. Then, by the induction hypothesis (applied to $\text{eval}_\Gamma(N_1, a)$ and $\text{eval}_\Gamma(N_2, N)$), there are environments $E_i'(i = 1, 2)$, E_{d_1} and E_{d_2} and a term M_2' such that

$$\begin{aligned}
[S, E, (M_1\ M_2) : C] &\Rightarrow [[M_2, E] : S, E, M_1 : ap : C] \\
&\stackrel{*}{\Rightarrow} [[a, E_1'] : [M_2, E] : S, E_{d_1}, ap : C] \\
&\Rightarrow [S, E, M_2 : a : ct : C] \\
&\stackrel{*}{\Rightarrow} [[M_2', E_2'] : S, E_{d_2}, a : ct : C] \\
&\Rightarrow [[a, \emptyset] : [M_2', E_2'] : S, E_{d_2}, ct : C] \\
&\Rightarrow [S, E'', M'' : C]
\end{aligned}$$

Then, in order to have $\text{eval}_\Gamma((N_1 N_2), M'')$, $\text{eval}_\Gamma(\text{constapply}(a, N), M'')$ must hold. Furthermore there is a closure $[L, E'']$ such that $\text{Real}([M_2', E_2']) =_\alpha N$, and $\text{Constapply}(a, [M_2', E_2']) = [L, E'']$.

We also know that $\text{eval}_\Gamma(\text{Real}([M'', E'']), N')$ for some N'. By the induction hypothesis there are Cl and E_{d_3} such that

$$[S, E'', M'' : C] \stackrel{*}{\Rightarrow} [Cl : S, E_{d_3}, C]$$

and $\text{Real}(Cl) =_\alpha N'$. Taking $[M', E'] = Cl$ concludes the proof of the lemma.

■

In order to prove the last part we need to observe the following property of the SEC machine:

Property 1. In one transition only the top item of the controlstring may be removed or altered, *i.e.*, $w : C$ may turn into $w' : C$ or $w'' : w' : C$, but the C must remain intact.

We now show that $\text{Eval}(M){=_\alpha}N$ implies $\text{eval}(M) = N$ by induction on the number of SEC steps in computing Eval, with the induction hypothesis strengthened as follows:

Lemma 3. For all closures $[M, E]$, stacks S, and controlstrings C, if there is an S' and E' such that

$$[S, E, M : C] \overset{*}{\Rightarrow} [S', E', C]$$

then $S' = Cl : S$ for some value closure Cl. Moreover there is an $M'{=_\alpha}\text{Real}(Cl)$ such that $\text{eval}_\Gamma(\text{Real}[M, E], M')$.

Proof: By induction on t the number of state transitions, and cases according to the structure of M.

Basis. $t = 1$. M must be a constant or a λ-abstraction. The result holds immediately.

Induction step. $t > 1$. We have two cases:

1. $M = x$. Suppose $E(x) = [M'', E'']$. Then

$$\begin{aligned} [S, E, M : C] &\Rightarrow [S, E'', M'' : C] \\ &\overset{*}{\Rightarrow} [S', E', C] \end{aligned}$$

But $S' = Cl : S$ by the induction hypothesis. Moreover, there is an $M'{=_\alpha}\text{Real}(CL)$ such that $\text{eval}_\Gamma(\text{Real}([M'', E'']), M')$. But $\text{Real}([x, E]) = \text{Real}([M'', E''])$, consequently $\text{eval}_\Gamma(\text{Real}([M, E]), M')$.

2. $M = (M_1\ M_2)$. We now have:

$$\begin{aligned} [S, E, (M_1\ M_2)] &\Rightarrow [[M_2, E] : S, E, M_1 : ap : C] \\ &\overset{*}{\Rightarrow} [S_1, E'', ap : C] \\ &\overset{*}{\Rightarrow} [S', E', C] \end{aligned}$$

We know that we must pass through intermediate states of the above forms in order to reach the desired last state. But we can apply the induction hypothesis to the second transition, taking $S_1 = [N_1, E_1] : [M_2, E] : S$, where $\text{eval}_{\Gamma}(\text{Real}([M_1, E]), M')$ for some $M' =_{\alpha} \text{Real}([N_1, E_1])$. In order show that S' is of the appropriate form, we do a case analysis on M', noting that M' is either an abstraction or a constant.

(a) $M' = \lambda x.N$. Then we have:

$$\begin{aligned}[][N_1, E_1] : [M_2, E] : S, E', ap : C] &\Rightarrow [S, E_1\{[M_2, E]/x\}, N_1 : C] \\ &\stackrel{*}{\Rightarrow} [S', E', C]\end{aligned}$$

Applying the induction hypothesis to the last transition we find that $S' = Cl : S$, where there is an $M'' =_{\alpha} \text{Real}(Cl)$ such that $\text{eval}_{\Gamma}(\text{Real}([N_1, E_1\{[M_2, E]/x\}]), M'')$. By an application of Lemma 1, several applications of Fact 1, and the definition of eval_{Γ}, we get: there is an $M''' =_{\alpha} \text{Real}(Cl)$ such that $\text{eval}_{\Gamma}(\text{Real}([M, E]), M''')$.

(b) $M' = c$. Then we have:

$$\begin{aligned}[][c, E_1] : [M_2, E] : S, E'', ap : C] &\Rightarrow [S, E, M_2 : c : ct : C] \\ &\stackrel{*}{\Rightarrow} [Cl' : S, E''', c : ct : C] \\ &\Rightarrow [[c, \emptyset] : Cl' : S, E', ct : C]\end{aligned}$$

The second intermediate state is justified by the induction hypothesis. Also note that there is an $M'' =_{\alpha} \text{Real}(Cl')$ such that $\text{eval}_{\Gamma}(\text{Real}([M_2, E]), M'')$. Finally, let $\text{Constapply}(c, Cl') = [N, F]$, so then we have

$$\begin{aligned}[][c, \emptyset] : Cl' : S, E''', ct : C] &\Rightarrow [S, F, N : C] \\ &\stackrel{*}{\Rightarrow} [Cl'' : S, F', C]\end{aligned}$$

where there is an $M''' =_{\alpha} \text{Real}(Cl'')$ such that $\text{eval}_{\Gamma}(\text{Real}([N, F]), M''')$ by the induction hypothesis. Through several applications of Fact 1, and the requirement

$$\text{Real}(\text{Constapply}(a, Cl)) =_{\alpha} \text{constapply}(a, \text{Real}(Cl))$$

we can conclude that there is an $M'''' =_{\alpha} \text{Real}(Cl'')$ such that

$$\text{eval}_{\Gamma}(\text{Real}([M, E]), M'''')$$

■

Chapter 4

Equivalence of eval and PCF

A specification of the set Constants and the function Constapply is given in the example in Section 2.4. We will demonstrate that Eval(M) = $PCF(M)$ when M is a program of ground type. This in turn will show that the SEC machine presented will properly evaluate well-typed programs in PCF (provided they are fully parenthesized, *e.g.*($\supset$ $f\!f$ M N) should be written as $(((\supset f\!f)M)N)$).

4.1 Preliminaries: The Language PCF

Consider the following set of rewrite rules for PCF:

1. (a) $(\supset_\sigma tt) \rightarrow_{\mathcal{L}} (\lambda x^\sigma \lambda y^\sigma x), (\supset_\sigma f\!f) \rightarrow_{\mathcal{L}} (\lambda x^\sigma \lambda y^\sigma y)$ (σ ground)

   (b) $(Y_\sigma M) \rightarrow_{\mathcal{L}} (M(Y_\sigma M))$

   (c) $(\lambda x M)N \rightarrow_{\mathcal{L}} [N/x]M$

   (d) $(+1 k_m) \rightarrow_{\mathcal{L}} k_{m+1} (m \geq 0)$

   (e) $(-1 k_{m+1}) \rightarrow_{\mathcal{L}} k_m (m \geq 0)$

   (f) $(Z k_0) \rightarrow_{\mathcal{L}} tt, (Z k_{m+1}) \rightarrow_{\mathcal{L}} f\!f$

2. (a) If $M \rightarrow_{\mathcal{L}} M'$ then $(MN) \rightarrow_{\mathcal{L}} (M'N)$

   (b) If $M \rightarrow_{\mathcal{L}} M'$ then $(aM) \rightarrow_{\mathcal{L}} (aM')$ (if $a \neq Y_\sigma$)

Note that these rules use a curried form of $\supset$. However, it is a simple task to show that $(((\supset_\sigma tt)M)N) \xrightarrow{3}_{\mathcal{L}} M$ and that $(((\supset_\sigma ff)M)N) \xrightarrow{3}_{\mathcal{L}} N)$.

We are not going to show directly that these rewrite rules for PCF are equivalent to eval. Instead we will work with PCF with a strict Y. We change rules 1b and 2b to:

1b′ $(Y_\sigma V) \rightarrow_{\mathcal{L}} (V(Y_\sigma V))$ (where V is a Value)

2b′ If $M \rightarrow_{\mathcal{L}} M'$ then $(aM) \rightarrow_{\mathcal{L}} (aM')$ (even if $a = Y_\sigma$)

It is easy to see that these new rules are equivalent to the old rules in the sense that a term M diverges in the old rules iff it diverges in the new rules. If M does not diverge in the old rules there must be a term (call it N) to which M reduces that cannot be further reduced, then in the new rules there must be a term (call it N') to which M reduces that cannot be further reduced. Furthermore N and N' are observationally equivalent[1] with respect to the old rules (and by induction on the structure of N, the new rules).

Consider the claim on the term (YM): If M diverges then (YM) diverges in either framework. If $M \rightarrow^*_{\mathcal{L}} V$ (V a value), then in the old framework we get $(V(YM))$, in the new we get $(V(YV))$ but since $M \rightarrow^*_{\mathcal{L}} V$ the two are observationally congruent in the old framework. By an induction on the length of the reduction one can show the desired result.

4.2 The Actual Proof

Theorem 2. For all well-typed, closed terms M with constants in Constants (as specified in the example in section 2.4) and constapply, also as defined in that example, then $M \rightarrow^*_{\mathcal{L}} M'$ (M' a value) iff $\text{eval}_\Gamma(M, M'')$.

But first we need several facts:

Fact 3. $\rightarrow_{\mathcal{L}}$ is deterministic. That is: if $M \rightarrow_{\mathcal{L}} M'$ then $\not\exists M'' \neq M'$ such that $M \rightarrow_{\mathcal{L}} M''$ Thus if $M \xrightarrow{n}_{\mathcal{L}} M''$, $M \xrightarrow{m}_{\mathcal{L}} M'$ and $m \leq n$ then $M' \xrightarrow{n-m}_{\mathcal{L}} M''$.

Fact 4. If $M_1 \xrightarrow{n}_{\mathcal{L}} M_1'$ then $(M_1 M_2) \xrightarrow{n}_{\mathcal{L}} (M_1' M_2)$ and $(aM_1) \xrightarrow{n}_{\mathcal{L}} (aM_1')$

[1] In order to define operational equivalence it is first necessary to introduce program contexts. A program context $C[\cdot]$ is simply a "closed" term of base type with a "hole." $C[T]$ is simply the term represented by C with the hole filled by the term T. M and M' are said to be operationally equivalent (to each other) iff for any program context $C[\cdot]$, $PCF(C[M])$ and $PCF(C[N])$ are both undefined, or are both defined and equal.

Fact 5. If M is a closed value, then $(cM) \rightarrow_{\mathcal{L}}$ constapply(c, M) which is to say that if constapply(c,M) is defined then (cM) reduces to it, and if constapply(c,M) is not defined then $\nexists M' : (cM) \rightarrow_{\mathcal{L}} M'$.

Proof: $(M \xrightarrow{n}_{\mathcal{L}} M' \Rightarrow \text{eval}(M) = M')$. By induction on n.

Basis. $n = 0$. M is a constant c, or M is an abstraction $(\lambda x N)$ in either case $M = M'$ and M has value M' at time 1.

Inductive Step. M is a combination, say $(M_1 M_2)$. For $(M_1 M_2) \rightarrow^{*}_{\mathcal{L}} M'$, a value, then it must be the case that $M_1 \xrightarrow{n_1}_{\mathcal{L}} M_1'$, where M_1' is a value. By Fact 4, $(M_1 M_2) \xrightarrow{n_1}_{\mathcal{L}} (M_1' M_2)$. The proof now breaks down into two cases depending on what kind of value M_1' is.

1. $M_1' = \lambda x N$. Then

$$(M_1 M_2) \xrightarrow{n_1}_{\mathcal{L}} ((\lambda x N) M_2) \rightarrow_{\mathcal{L}} ([M_2/x]N)^{n-(n_1+1)} \rightarrow_{\mathcal{L}} M'.$$

By the inductive hypothesis then $\text{eval}(M_1) = \lambda x N$ and $\text{eval}([M_2/x]N) = M'$. Thus:

$$\text{eval}(M_1 M_2) = \text{eval}([M_2/x]N) = M'$$

2. $M_1' = c$. Then it must be the case that $M_2 \xrightarrow{n_2}_{\mathcal{L}} M_2'$ where M_2' is a value. By Fact 4:

$$(M_1 M_2) \xrightarrow{n_1}_{\mathcal{L}} (cM_2) \xrightarrow{n_2}_{\mathcal{L}} (cM_2') \rightarrow_{\mathcal{L}} \text{constapply}(c, M_2')^{n-(n_1+n_2+1)} \rightarrow_{\mathcal{L}} M'$$

By the inductive hypothesis:

$$\text{eval}(M_1) = c,\ \text{eval}(M_2) = M_2',\ \text{and eval(constapply}(c, M_2')) = M'.$$

Thus:

$$\text{eval}(M_1 M_2) = \text{eval(constapply}(c, M_2')) = M'$$

■

Proof: $\text{eval}_r(M, M') \Rightarrow M \rightarrow^{*}_{\mathcal{L}} M'$). By induction on the definition of eval_r

Basis. $M = M'$, and is either a constant or an abstraction. In either case $M \xrightarrow{0}_{\mathcal{L}} N$ and we are done.

Inductive Step. M is neither a constant nor an abstraction so it must be an application, say $(M_1 M_2)$. In order for $\text{eval}_\Gamma((M_1 M_2), M')$ to hold it must be the case that there is an M_1' such that $\text{eval}_\Gamma(M_1, M_1')$ holds. Then, by the induction hypothesis, $M_1 \rightarrow^*_{\mathcal{L}} M_1'$, and then by rule 2a $(M_1 M_2) \rightarrow^*_{\mathcal{L}} (M_1' M_2)$. The analysis now breaks down into 2 cases based upon M_1'.

1. $M_1' = \lambda x N$. In this case we must have $\text{eval}_\Gamma([M_2/x]N, M')$. But then

$$\begin{aligned} M = (M_1 M_2) &\rightarrow^*_{\mathcal{L}} (\lambda x N) M_2 \\ &\rightarrow_{\mathcal{L}} [M_2/x]N \end{aligned}$$

   and by the inductive hypothesis $[M_2/x]N' \rightarrow^*_{\mathcal{L}} M'$ and so $M \rightarrow^*_{\mathcal{L}} M'$.

2. M_1' is a constant. Thus there must be an M_2' such that both $\text{eval}_\Gamma(M_2, M_2')$ and $\text{eval}_\Gamma(\text{constapply}(M_1', M_2'), M')$ hold. By the induction hypothesis $M_2 \rightarrow^*_{\mathcal{L}} M_2'$. Hence by rule 2$b'$ $(M_1' M_2) \rightarrow^*_{\mathcal{L}} (M_1' M_2')$. Let $N = \text{constapply}(M_1', M_2')$. By fact 5 we know that $(M_1 M_2') \rightarrow_{\mathcal{L}} N$. Finally, since $N = \text{constapply}(M_1', M_2')$, $\text{eval}_\Gamma(N, M')$ holds; thus by the induction hypothesis $N \rightarrow^*_{\mathcal{L}} M'$ and more importantly, $M \rightarrow^*_{\mathcal{L}} M'$.

■

Chapter 5

Open problems

There is a variant on "true" call-by-name which tries to balance the greater expressiveness of call-by-name with the efficiency of call-by-value. In call-by-need, the arguments are only evaluated if they are used, but if they are used their values are "memoized" so that if they are used again their values are immediately available. Due to the additional bookkeeping needed to determine whether or not an argument has been evaluated, there is a slight degradation of performance over call-by-value. For a language without side effects, the semantics of call-by-name is the same as call-by-need. Consequently, in this limited context, call-by-need can be considered a particular implementation strategy for call-by-name—a strategy that is generally more efficient than the straightforward one.

Acknowledgements

I would like to thank my advisor, Albert R. Meyer, for giving me a chance and for suggesting this project, and his many hours discussing this research. I would especially like to thank Jon Riecke for his excellent feedback, and for being there to help clarify my every confusion. Both Jon and Albert have been invaluable assets in the production of this Thesis. I would also like to thank Trevor Jim and Mike Ernst for helpful comments on an earlier draft of this Thesis, and to additionally thank Trevor for implementing the SEC machine in ML. Finally, I would like to thank my fiancé, Nicole Skinner, for providing support which has made this process much easier.

Bibliography

[1] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice Hall, 1980.

[2] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[3] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[4] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–257, 1977.