A Calculus for Assignments In Higher-Order Languages

By

Matthias Felleisen & Daniel P. Friedman
Department of Computer Science
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 202

A Calculus for Assignments
In Higher-Order Languages

by

Matthias Felleisen & Daniel P. Friedman

October, 1986

This report is based on work supported in part an IBM Fellowship Award to Matthias Felleisen, and by the National Science Foundation grants numbered DCR 83-03325 and DCR 85-01277.

This report also appeared in the *Conf. Record of the 14th Annual ACM Sym. on Principles of Programming Languages*, Munich, W. Germany (Jan. 1987), 314–345.

A Calculus for Assignments in Higher-Order Languages

Matthias Felleisen, Daniel P. Friedman

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405, USA

Abstract. Imperative assignments are abstractions of recurring programming patterns in purely functional programming languages. When added to higher-order functional languages, they provide a higher-level of modularity and security but invalidate the simple substitution semantics. We show that, given an operational interpretation of a denotational semantics for such a language, it is possible to design a two-level extension of the λ_v-calculus. This calculus provides a location-free rewriting semantics of the language and offers new possibilities for reasoning with assignments. The upper level of the calculus factors out all the steps in a reduction sequence which must be in a linear order; the lower level allows a partial ordering of reduction steps.

1. Pros and cons of assignment statements

The assignment statement is an abstraction commonly found in algorithmic programming languages. It represents and expresses the concept of a state change. Like any other mental abstraction it is associated with intellectual costs and benefits. Here the question about its usefulness centers around the dichotomy between modularity and security on one hand and ease-of-understanding and simplicity of the language semantics on the other.

Programs without assignment statements can be understood in terms of a simple substitution model. Assignments can be imitated in a higher-order functional programming language, *e.g.*, the λ-calculus, by passing a functional abstraction of the current store to all functions and by returning an updated store abstraction. In practice this solution is implemented by passing all relevant state variables around and by returning a compound structure which reflects the new state. Clearly, the more state variables, the longer the parameter lists. Even worse, more functions than necessary have access to the state variables and every one of these functions can corrupt the current state by altering the value of a state variable. These arguments obviously call for a more modular and secure solution. The assignment statement is the traditional answer.

In a higher-order language with assignments, a state variable can be hidden in the scope of those functions which deal with that particular part of the state [8, 10]. However, the introduction of assignment statements has a price: the simple program rewriting or substitution model for the evaluation of functional programs no longer works. In order to reason about programs one now has to rely on a denotational or operational store semantics.

The introduction of a store means that variables are placeholders for locations which contain a value and whose contents may be altered. Although this defines a mathematical meaning of state and assignments, the model has some problems. To begin with, the store model does not abstract enough from traditional machine architectures. Given a store semantics it is relatively easy to implement this language on a store machine, but it does not provide any insight on how to implement an imperative language directly on a non-von Neumann machine.

Another problem with these models is that they cannot be used in a straightforward manner in proofs of program properties. The standard versions fail to identify intuitively equivalent program pieces. A classical example is the block

$$\textbf{begin var } x = 0; \textbf{ skip end}.$$

It is impossible to prove this block equal to the statement **skip** unless the semantics explicitly deallocates locations upon exit from a lexical scope. The freedom to declare variables in an arbitrary order provides another example.

To appear in the 1987 ACM Symposium on Principles of Programming Languages.

Even though the sequence of declarations in

begin var $x = 0$, $y = 1$; $\langle cmd \rangle$ **end**

should not matter, an ordinary store semantics cannot prove this block equivalent to

begin var $y = 1$, $x = 0$; $\langle cmd \rangle$ **end**.

For restricted programming languages one can get by with a location-free store semantics [1, 3, 5] that has the desired properties. But these solutions do not generalize to higher-order languages. A denotational store semantics with appropriate attributes is also available [7], but it is rather complicated. What is missing is a simple, syntactic calculus for reasoning about these and other equivalences.

The same general problem emerges when non-functional control operators are introduced into functional languages. We have shown [6] that, given an operational semantics of a thus extended λ-calculus programming language, we can *design* a two-level axiomatic extension of the calculus which corresponds to these additional operations. A similar derivation also produces a solution to the problem at hand. The result is λ_s, a two-level extension of the λ_v-calculus [12] which incorporates assignment expressions. It makes it possible to manipulate programs with assignments in an algebraic manner. In particular, it is trivial to show that the program pieces of the above examples are equivalent. The standard reduction sequence of the calculus defines a location-free operational semantics of higher-order programming languages with assignments which could be used for a rewriting machine.

In the next section we present a λ-calculus based programming language with an expression-oriented assignment construct. We assume some familiarity with the terminology and notation of the traditional λ-calculus [2]. The semantics of this language is defined via an abstract store machine. We systematically transform this machine into a program-store rewriting system. In Sections 3 and 4 we design the extended calculus on the basis of this rewriting system. Section 4 also contains the key theorems about the λ_s-calculus: a variation of the Church-Rosser Theorem, a Standardization Theorem à la Curry-Feys, and two correspondence theorems. The fifth section is devoted to examples. Section 6 summarizes our development and discusses related work.

2. Λ with assignment abstractions

Adding an assignment statement to an expression-oriented language like Λ confronts the language designer with a problem: if added naïvely, the language is divided into two major syntactic categories, namely statements and expressions. In an expression-oriented language there is almost no restriction on how expressions may be joined to form a larger expression; in a divided language a user is required to know where he can use one sentence category, but not the other. In order to avoid this difficulty we have added σ-abstractions to Λ. Their syntax is $(\sigma x.M)$ where x is a variable and M is an expression. A σ-abstraction does not bind the variable, but it abstracts the right to assign a variable a new value. When a σ-abstraction is applied to a value, it assigns x that value and M is evaluated to yield the result of the application. The meaning of the remaining constructs should be adapted accordingly: variables are re-assignable placeholders for values, abstractions correspond to call-by-value procedures, and applications invoke the result of the function part on the value of the argument part. The syntax is summarized in Definition 1.

The set of free and bound variables of a term M, $FV(M)$ and $BV(M)$, is defined as usual; the only binding construct in the language is the λ-abstraction. The set of assigniable variables in M, $AV(M)$, contains all variables that occur in the variable position of a σ-abstraction:

$$\begin{aligned} AV(x) &= \emptyset, \\ AV(\lambda x.M) &= AV(M), \\ AV(MN) &= AV(M) \cup AV(N), \\ AV(\sigma x.M) &= \{x\} \cup AV(M). \end{aligned}$$

Definition 1: The language Λ_σ

The improper symbols are λ, (,), ., and σ. *Vars* is a countable set of variables. The symbols $x, \ldots$ range over *Vars* as meta-variables but are also used as if they were elements of *Vars*. The *term set* Λ_σ contains

— *variables:* x if $x \in$ *Vars*;

— *λ-abstractions:* $(\lambda x.M)$ if $M \in \Lambda_\sigma$ and $x \in$ *Vars*;

— *applications:* (MN) if $M, N \in \Lambda_\sigma$;

— *σ-abstractions* $(\sigma x.M)$ if $M \in \Lambda_\sigma$ and $x \in$ *Vars*.

The union of variables and abstractions is referred to as the set of *values*. Λ, the term set of the traditional λ-calculus, stands for Λ_σ restricted to variables, applications, and abstractions.

Terms with no free variables are called *closed terms* or *programs*. Since we want to avoid syntactic issues, we adopt Barendregt's convention of identifying ($\equiv_\alpha$ or just $\equiv$) terms that are equal modulo some renaming of bound variables and his hygiene condition which says that *in a discussion, free variables are assumed to be distinct from bound ones*. Substitution is extended in the natural way and we use the notation $M[x := N]$ to denote the result of substituting all free variables x in M by N.

The semantics of Λ_σ-programs is defined via an abstract machine. The machine manipulates quadruples of control strings, environments, stores, and continuations. A *control string* is either the symbol ‡ or a Λ_σ-expression. *Environments*, denoted by ρ, are finite maps from variables to natural numbers (or locations); *stores*, denoted by θ, are finite maps from natural numbers to λ- and σ-closures. If f is a function, then $f[x := y]$ is like f except at the place x where it is y. A *closure* or *semantic value* is an ordered pair $\langle M, \rho\rangle$ composed of an abstraction M and an environment ρ whose domain covers the free variables of the abstraction, *i.e.*, $FV(M) \subseteq Dom(\rho)$. Depending on the kind of embedded abstraction a closure is called λ- or σ-closure. A *continuation code* represents the remainder of the computation, *i.e.*, it encodes what the machine has left to do when the current control string is evaluated. The representation is defined in two stages. If N is a Λ_σ-term, ρ is an environment such that $FV(N) \subseteq Dom(\rho)$, and V is a semantic value, then a p-continuation has one of the following forms:

$$(\mathbf{stop}),\ (\kappa\,\mathbf{arg}\ N\rho),\ (\kappa\,\mathbf{fun}\ V)$$

where κ is a p-continuation. A ret-continuation is of the form

$$(\kappa\,\mathbf{ret}\ V)$$

where κ is a p-continuation and V is a semantic value.

A CESK-*machine state* is either a quadruple of the form $\langle ‡, \emptyset, \theta, \kappa\rangle$ where κ is a ret-continuation or a quadruple of the form $\langle M, \rho, \theta, \kappa\rangle$ where M is a Λ_σ-term, ρ is an environment with $FV(M) \subseteq Dom(\rho)$, θ is a store with $Ran(\rho) \subseteq Dom(\theta)$, and κ is a p-continuation. Machine states of the form $\langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle$ are the *initial* states; if V is a closure and θ is defined on all locations which are used by V then $\langle ‡, \emptyset, \theta, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle$ is a *terminal* state. The state transition function is displayed in Table 1. We use $\overset{CESK^+}{\longmapsto}$ and $\overset{CESK^*}{\longmapsto}$ to denote the transitive and transitive-reflexive closure, respectively.

In order to evaluate a program M, the machine is started in the initial state $\langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle$. When the machine reaches a terminal state, it stops and returns the value on the stack together with the store as the answer. The evaluation function is defined on closed terms M by:

$$eval_{CESK}(M) = \langle V, \theta\rangle, \text{ if}$$

$$\langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle \overset{CESK^+}{\longmapsto} \langle ‡, \emptyset, \theta, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle.$$

Since the transition function is clearly defined on all legal states except for terminal ones, the machine, when started in an initial state, either halts in a terminal state or never terminates.

The CESK-machine is an operational interpretation of a denotational semantics for Λ_σ. This machine yields a mechanism for the evaluation of Λ_σ-programs. A programmer, however, prefers to reason more directly about programs. The environment and continuation components can be eliminated by simple transformations [6]. An environment is a machine-oriented representation of substitutions. By carrying out these substitutions at the right time, environments become superfluous. Since environments map identifiers to natural numbers, substitutions replace free identifiers by numbers: the language of control strings must be extended appropriately.

The continuation component of an abstract machine remembers the context of the next executable subexpression while the machine is taking the program apart on its search for this *redex*. The CESK-machine recognizes two kinds of redexes: an identifier (or a location) and an

Table 1: The CESK-machine

$$\begin{aligned}
\langle x, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle ‡, \emptyset, \theta, (\kappa\,\mathbf{ret}\ \theta(\rho(x)))\rangle && (1)\\
\langle \lambda x.M, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle ‡, \emptyset, \theta, (\kappa\,\mathbf{ret}\ \langle \lambda x.M, \rho\rangle)\rangle && (2)\\
\langle MN, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle M, \rho, \theta, (\kappa\,\mathbf{arg}\ N\rho)\rangle && (3)\\
\langle ‡, \emptyset, \theta, ((\kappa\,\mathbf{arg}\ N\rho)\ \mathbf{ret}\ F)\rangle &\overset{CESK}{\longmapsto} \langle N, \rho, \theta, (\kappa\,\mathbf{fun}\ F)\rangle && (4)\\
\langle ‡, \emptyset, \theta, ((\kappa\,\mathbf{fun}\ \langle \lambda x.M, \rho\rangle)\ \mathbf{ret}\ V)\rangle &\overset{CESK}{\longmapsto} \langle M, \rho[x := n], \theta[n := V], \kappa\rangle && (5)\\
&\qquad \text{where } n \notin Dom(\theta)\\
\langle \sigma x.M, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle ‡, \emptyset, \theta, (\kappa\,\mathbf{ret}\ \langle \sigma x.M, \rho\rangle)\rangle && (6)\\
\langle ‡, \emptyset, \theta, ((\kappa\,\mathbf{fun}\ \langle \sigma x.M, \rho\rangle)\ \mathbf{ret}\ V)\rangle &\overset{CESK}{\longmapsto} \langle M, \rho, \theta[\rho(x) := V], \kappa\rangle && (7)
\end{aligned}$$

application with semantic values in both positions. All other machine transitions are simply moves within the context of a redex to find the redex. They determine that the search proceeds through applications in a preorder traversal. We therefore define the set of sk-contexts and *use them* to represent continuations:

(skC1) $[\ ]$ is a context,

(skC2) $VC[\ ]$ is a context where V is a value, and

(skC3) $C[\ ]M$ is a context, where M is arbitrary, and

in the last two clauses $C[\ ]$ stands for a context. If $C[\ ]$ is a context, then $C[M]$ is the term where the hole is filled with the term M.

Given a context representation of continuations, we simplify the machine by combining the continuation and control string components. The combination consists of two steps: first, the context holes are filled with the control string, ‡'s are simply dropped; second, all search rules are now identity steps and may be eliminated. Putting these three transformations together we arrive at the control string-store machine whose transition function is given in Table 2. The evaluation function is adapted in the obvious way:

$$eval_{CS}(M) = \langle V,\theta\rangle \text{ iff } \langle M,\emptyset\rangle \overset{CS}{\longmapsto}{}^{*} \langle V,\theta\rangle.$$

The translation of closures in the CESK-machine to abstractions in the CS-machine is accomplished by a function $\mathcal{R}$ which replaces free variables by locations:

$$\mathcal{R}(\langle M,\rho\rangle) = M[x_1 := \rho x_1]\ldots[x_n := \rho x_n] \quad \text{where } FV(M) = \{x_1,\ldots,x_n\}.$$

With $\mathcal{R}$ and its natural extension $\overline{\mathcal{R}}$ on stores we can describe in what sense the two machines correspond:

Theorem 2.1 (CS-Simulation). *For any program M,*

$$eval_{CS}(M) = \langle \mathcal{R}(V), \overline{\mathcal{R}}(\theta)\rangle \textit{ iff } eval_{CESK}(M) = \langle V,\theta\rangle.$$

Proof. The proof proceeds in three stages according to the three transformations. For each transformation step the resulting machine is constructed and shown to be equivalent to the original one. Two auxiliary morphisms are needed in order to prove that sk-contexts correctly represent continuations and that they can be eliminated by combining the control-string component with the continuation component. We omit the details because the formalization is lengthy, but straightforward. □

The CS-machine is close in spirit to a Plotkin-style [11] operational semantics of Λ_σ. With the above derivation and theorem we have shown that the system reflects a more conventional operational interpretation of a denotational store semantics. In the next section we transform the machine into a storeless program rewriting system. We then proceed to develop a true calculus of assignments.

3. Designing a control string rewriting system

Before we can combine the store component of the CS-machine with the control string part, we need to clarify what the store accomplishes. According to rule (CS2), the store remembers which positions in the body of a λ-abstraction were labeled by the same variable. In other words, the store retains the *bound variable equivalence relation* among subterms beyond the point of application. In rule (CS3) the machine utilizes this information. It assigns a new value to all the positions which are descendents of the same bound variable, thus implementing the assignment application.

The static relationship among the bound variables of a λ-abstraction is, for example, captured in de Bruijn's nameless Λ-notation [4]. In this representation of an abstraction all variable names are replaced by a (static distance) pointer to the binding λ-symbol. If numbers are used for these pointers, they generally count the number of λ's between the occurrence and the binding λ-symbol, *e.g.*, $\lambda x.\lambda y.xy$ becomes $\lambda.\lambda.10$. Drawing arrows from the occurrence to the λ-symbol is an equivalent, but less machine-oriented way.

In terms of the arrow notation the store in the CS-machine retains the connecting arrows after the abstraction was applied and the variable occurrences were replaced by values. One way to express this equivalence relation in a textual form is to mark all these value occurrences with the same label. This would roughly correspond to a replacement of locations in the control strings of the CS-machine by *labeled values*[1] of the form V^n where $\theta(n) = V$. Since every (CS2)-step retains an arrow that is

Table 2: The CS-machine

$$\langle C[n],\theta\rangle \overset{CS}{\longmapsto} \langle C[\theta(n)],\theta\rangle \quad (1)$$
$$\langle C[(\lambda x.M)V],\theta\rangle \overset{CS}{\longmapsto} \langle C[M[x:=n]],\theta[n:=V]\rangle \quad \text{where } n \notin Dom(\theta) \quad (2)$$
$$\langle C[(\sigma n.M)V],\theta\rangle \overset{CS}{\longmapsto} \langle C[M],\theta[n:=V]\rangle \quad (3)$$

[1] Labeled terms have also been used for the investigation of the regular λ-calculus [2, p.353]. Although our problem at hand is unrelated to these investigations we have chosen to adopt the notation in order to avoid new terminology.

different from all others, the label must be unique. In the framework of our store semantics we expressed this as $n \notin Dom(\theta)$, *i.e.*, picking an unused location. For the rewriting system we say that the label must be *fresh*. For the discussions and proofs below we assume that *whenever a new label is introduced for a transition or derivation step, this label is distinct from all others used in the same textual context*. Keeping this condition in mind reasoning with labels is similar to reasoning with substitution in conjunction with the hygiene condition for bound parameters. Accordingly we refer to this condition as the *hygiene condition for labels*.

The identity of fresh labels is of no importance. They only name an equivalence class which is already implicitly there. In order to avoid naming problems in this context, we consider two terms equivalent if[2] they are α-equivalent except for their labels and if there is a bijection between their label structure. We call this equivalence label-equivalence and denote it with $\equiv_{lab}$. We shall furthermore extend the notation $\equiv$ to mean the union of α- and label-equivalence.

These considerations nearly lead to a combination of the store and control string components of the CS-rewriting system. Store values which contain a direct or indirect reference to their own location cause a minor complication. On traditional machines this corresponds to a circular structure. Infinite terms would directly represent these structures but would also unnecessarily burden the mathematical treatment. A simpler solution ignores self-referential locations for an assignment of the location n to a value V and relies on the dereferencing transition to provide a proper value which contains the current value at these positions.

The transliteration of the three rules is easy. The substitution step (C2) performs the unique labeling of the values so that assignment expressions may trace the origin of their variable part. The assignment, in turn, is implemented by a replacement algorithm which changes all subterms with the same label to a new value. Our notation for this operation is $L[n := V]$; it is defined in Table 3. The algorithm does not replace n-labeled values inside of V. As motivated earlier the dereferencing rule maintains the invariant that all values with label n inside of V^n are updated to the *current value*, V^n. Hence, dereferencing means to step from V^n to $V[n := V]$ in an sk-context. The pure control string rewriting system is summarized in Table 3.

A translation of CS-states into labeled Λ_σ-terms is impossible. By the above argument self-referential values in assignments always contain a part or all of their "assignment history." Hence, for a proof of the simulation theorem we need to formulate a variant of the CS-machine whose store component keeps around the store history. In other words, a store is now a finite map from locations to value-history pairs—*val* and *his* are the respective selectors—where the *his* component represents the history of the location contents. The machine transition function becomes:

$$\begin{aligned}
\langle C[n], \theta\rangle &\xmapsto{CS} \langle C[val(\theta(n))], \theta\rangle\\
\langle C[(\lambda x.M)V], \theta\rangle &\xmapsto{CS} \langle C[M[x := n]], \theta[n := \langle V, *\rangle]\rangle\\
&\qquad\text{where } n \notin Dom(\theta)\\
\langle C[(\sigma n.M)V], \theta\rangle &\xmapsto{CS} \langle C[M], \theta[n := \langle V, \theta(n)\rangle]\rangle.
\end{aligned}$$

This modified machine is clearly in accord with the original one. Its history mechanism permits a translation of location expressions to labeled terms:

$$\begin{aligned}
\mathcal{S}(x, \theta) &= x,\\
\mathcal{S}(n, \theta) &= \mathcal{S}(val(\theta(n)), \theta[n := his(\theta(n))])^n,\\
\mathcal{S}(\lambda x.M, \theta) &= \lambda x.\mathcal{S}(M, \theta),\\
\mathcal{S}(MN, \theta) &= \mathcal{S}(M, \theta)\mathcal{S}(N, \theta),\\
\mathcal{S}(\sigma X.M, \theta) &= (\sigma \mathcal{S}(X, \theta).\mathcal{S}(M, \theta)).
\end{aligned}$$

Table 3: The C-rewriting system

$$C[V^n] \xmapsto{C} C[V[n := V]] \qquad (1)$$

$$C[(\lambda x.M)V] \xmapsto{C} C[M[x := V^n]] \quad \text{where } n \text{ is fresh} \qquad (2)$$

$$C[(\sigma U^n.M)V] \xmapsto{C} C[M][n := V] \qquad (3)$$

The labeled term substitution $L[n := V]$ is defined by:

$$x[n := V] = x,\ U^n[n := V] = V^n,\ U^m[n := V] = U[n := V]^m \text{ if } n \neq m,$$
$$(\lambda x.M)[n := V] = \lambda x.M[n := V],$$
$$(MN)[n := V] = (M[n := V]N[n := V]),$$
$$(\sigma X.M)[n := V] = (\sigma X[n := V].M[n := V]).$$

[2] For the representation of circular structures (see below) we are even more generous and say that the contents of embedded self-references does not matter.

With these technical modifications we can now show that the new machine still computes the same values:

Theorem 3.1 (C-Simulation). *For any program M,*

$$eval_{CS}(M) = \langle V, \theta\rangle \text{ iff } M \overset{C}{\longmapsto}{}^{*} \mathcal{S}(V, \theta).$$

Proof. The proof verifies that every step of the (history) CS-transition function is reflected by the C-rewriting system. □

The C-rewriting system could be a reasonable basis for the design of an assignment calculus. The transition function specifies the standard reduction function of the calculus and, hence, determines the basic notions of reduction. Indeed, these notions of reduction must be context-independent versions of the C-transition rules. This dependency, however, is somewhat hidden in our current rules and we therefore construct an intermediate rewriting system which bridges the gap to the next section.

The rule for assignment applications (C3) is evidently context-sensitive, but, contrary to appearances, the delabeling rule depends on its context as well. The unique partitioning of a program into an sk-context and a C-redex implicitly ensures that the delabeling of a labeled value happens at the right time and produces the correct value. If $C[\ ]$ were not an sk-context, the delabeling might undermine the effect of an assignment to that very variable[3]. The application rule is free of this problem. Both parts are values and cannot be altered; only embedded references to labeled values may be changed, but this is, as we show below, irrelevant.

The application transition *per se* is not context-sensitive, but it is troubled by a different problem. In the assignment-free sublanguage Λ, a variable always stands for a value and $(\lambda x.M)y$ is a valid axiom in the corresponding λ_v-calculus. It is safe to say that variables *are* values.

Variables cannot be treated as values in the full language. The Λ_σ-semantics replaces all bound variables by labeled values. Labeled values must be delabeled at the correct time in order to become values. This is equally true for assignable and non-assignable variables. Reasoning with variables has become impossible. If nothing is changed, the λ_v-calculus cannot be a subcalculus of a reasoning system for Λ_σ. This is clearly undesirable.

The root of our problem is the equalizing treatment of assignable and non-assignable variables. Put differently, although non-assignable variables can never change their associated value, we impose the same intellectual cost on them and assignable variables. We can improve on this by partitioning the set of variables, *Vars*, into a subset of assignable variables, Var_σ, and non-assignable ones, Var_λ. In order to emphasize the higher cost we change the syntax and introduce a delabeling application in which assignable variables must be embedded. This not only underlines their different character, but also makes clear that the delabeling must happen at a particular point in time. Although this does not quite solve the above delabeling-timing problem, it at least reminds the user of its existence.

A natural form of the delabeling application could be $(\mathcal{D}\, x)$ and $(\mathcal{D}\, V^n)$, respectively, where $\mathcal{D}$ is a terminal symbol. However, in anticipation of the next section, we introduce the form $(\mathcal{D}\, X M)$ [4] where M is an arbitrary expression which is to consume the delabeled value. The modified transition rule (C1) becomes:

$$C[(\mathcal{D}\, V^n M)] \overset{mC}{\longmapsto} C[MV[n := V]].$$

The transition rule for application is split according to the partitioning of the set *Vars*:

$$C[(\lambda x.M)V] \overset{mC}{\longmapsto} C[M[x := V]] \text{ where } x \in Var_\lambda,$$
$$C[(\lambda x.M)V] \overset{mC}{\longmapsto} C[M[x := V^n]]$$
$$\text{where } x \in Var_\sigma \text{ and } n \text{ is fresh.}$$

The new language Λ_S is displayed in Definition 2. In

Definition 2: The language Λ_S

The improper symbols are λ, (,), ., σ, and $\mathcal{D}$. $Vars = Var_\lambda \cup Var_\sigma$ is a countable set of variables. We use the set of natural numbers, ω, as labels. The set of values contains variables, abstractions, and σ-abstractions. V ranges over values, $n \in \omega$ over labels, and X over Var_σ and labeled values. Λ_S contains

— *variables:* x if $x \in Var_\lambda$;

— *λ-abstractions:* $(\lambda x.M)$ if $M \in \Lambda_S$ and $x \in Vars$;

— *applications:* (MN) if $M, N \in \Lambda_S$;

— *σ-abstractions:* $(\sigma x.M)$ and $(\sigma V^n.M)$ if $M, N \in \Lambda_S$ and $x \in Var_\sigma$;

— *$\mathcal{D}$-applications:* $(\mathcal{D}\, x M)$ and $(\mathcal{D}\, V^n M)$ if $M \in \Lambda_S$ and $x \in Var_\sigma$.

[3] This is a different characterization of the fact that assignment statements enforce a precise sequencing of events in a machine.

[4] This could not have been done if we had not partitioned the set of variables: we could not have written a consumer M in this case.

Λ_S expressions like $((\bar{\mathcal{D}}\mathrm{I}^1\mathrm{I})(\mathcal{D}^{\prime}0^{11}\mathrm{I}))$ are legal and have a well-defined meaning, but they do not represent a stage in the evaluation of some Λ_σ-program. This is another indication that labels are an abstraction of locations. It is only on von Neumann machines that labels are most conveniently implemented as locations.

The splitting of the β-transition step complicates the comparison of the two transition systems. Since the modified C-rewriting system omits labels from non-assignable values, we need to keep around this information for the morphism from Λ_σ to Λ_S. One way to achieve this is to tag the values of a β-step with a truth value indicating the set membership of the original placeholder:

$$C[(\lambda x.M)V] \overset{C}{\longmapsto} C[M[x := V^{n,\tau}]],$$
$$\text{where } \tau \iff x \in AV(M) \setminus BV(M).$$

For the following theorem and proof we use the self-explanatory symbols β and σ for τ. The rest of the transition rules are adapted accordingly.

This slightly modified C-rewriting system clearly computes the same value as the original one, if one simply removes the tag information at the end. Given these technical changes we can define a simulation morphism and prove a simulation theorem. A labeled Λ_σ-expression M is translated to a Λ_S-expression $\overline{M}$ according to the following definition:

$$\overline{x} = x \text{ if } x \in Var_\lambda,\ \overline{x} = (\mathcal{D}\, x\mathrm{I}) \text{ if } x \in Var_\sigma,$$
$$\overline{V^{n,\beta}} = V,\ \overline{V^{n,\sigma}} = (\mathcal{D}\, V^n\mathrm{I}),$$
$$\overline{\lambda y.M} = \lambda y.\overline{M},$$
$$\overline{MN} = \overline{M}\,\overline{N},$$
$$\overline{\sigma x.M} = \sigma x.\overline{M},\ \overline{\sigma V^{n,\sigma}.M} = \sigma V^n.\overline{M}.$$

The translation assumes that the nature of a variable is known beforehand. Without that the translation would become more complex, but not more instructive. The simulation theorem becomes:

Theorem 3.2 (mC-Simulation). *For any program M (in Λ_σ) and value V,*

$$M \overset{C}{\longmapsto}{}^{*} V \text{ iff } \overline{M} \overset{mC}{\longmapsto}{}^{*} V.$$

Proof. As above one can directly prove that every C-step is mirrored by some mC-transitions. □

From the modified C-rewriting system it is only a short step to the λ_s-calculus. As mentioned above we accept the transition function as the specification of the standard reduction function and derive the basic notions of reduction and computation relations, *i.e.*, we eliminate the context dependency of assignment and delabeling rules as much as possible.

4. The λ_s-calculus

The rewriting rules for regular applications are already context independent. They induce the well-known β_v-notion [12] and the β_σ-variation:

$$(\lambda x.M)V \xrightarrow{\beta_v} M[x := V] \qquad (\beta_v)$$
where $x \in Var_\lambda$ and V is a value,
$$(\lambda x.M)V \xrightarrow{\beta_\sigma} M[x := V^n] \qquad (\beta_\sigma)$$
where $x \in Var_\sigma$, V is a value, and n is fresh.

The transitions for the other redexes depend on their context. In order to obtain notions of reductions we analyze the possible cases for the construction of sk-contexts. From the case of the empty sk-context, we get relations which work when the redex is at the root of the term:

$$(\sigma U^n.M)V \rhd_\sigma M[n := V] \text{ where } V \text{ is a value} \qquad (\rhd_\sigma)$$
$$(\mathcal{D}\, V^n\, M) \rhd_{\mathcal{D}} MV[n := V]. \qquad (\rhd_{\mathcal{D}})$$

We call them *computation relations* and emphasize this by using a $\rhd$ instead of the customary $\longrightarrow$ for a notion of reduction. When the redex is nested inside a proper sk-context, it must proceed to the root so that it can use the computation relation and complete the assignment or delabeling step. There are two cases: namely, when a redex is to the right of a value and when it is to the left of an arbitrary expression. In both instances the redex must incorporate the next piece of the context and then do the same to the rest of the context. Incorporating a piece of the context means extending the continuation part of the redex in the appropriate way. For a σ-abstraction the body is the first part of the continuation and a σ-redex can make the next context subexpression a part of its body:

$$N((\sigma X.M)V) \xrightarrow{\sigma_R} (\sigma X.(NM))V \qquad (\sigma_R)$$
where N and V are values
$$((\sigma X.M)V)N \xrightarrow{\sigma_L} (\sigma X.(MN))V \qquad (\sigma_L)$$
where V is a value.

In a $\mathcal{D}$-application the consuming function corresponds to the first piece of the rest of the computation. Therefore a $\mathcal{D}$-redex introduces a new function which, when applied to a value, will channel the value to the original consumer:

$$N(\mathcal{D}\, X\, M) \xrightarrow{\mathcal{D}_R} (\mathcal{D}\, X\, (\lambda v.N(Mv))) \qquad (\mathcal{D}_R)$$
where N is a value
$$(\mathcal{D}\, X\, M)N \xrightarrow{\mathcal{D}_L} (\mathcal{D}\, X\, (\lambda v.MvN)). \qquad (\mathcal{D}_L)$$

If these four notions of reductions are applicable to *deeply* nested redexes, they clearly move any C-rewriting redex to the root of an sk-context where the computation relations can finish the simulation of a C-transition step. To this end, we form the compatible closure of the notions of reductions. Because of the β_σ-rule this construction differs slightly from the usual one. A β_σ-step requires a unique labeling of its argument part, hence, if we want to perform such a step inside of an application, for example, we pick a representative of the new term which satisfies this need. In principle, this construction is unnecessary if we simply observe the hygiene condition for labels; it merely shows that the condition could be explicitly enforced. Otherwise the definition of the one-step reduction is quite standard and is formalized in Definition 3. The natural extensions of this one-step reduction are the transitive-reflexive closure and the respective congruence relation.

Since the computation relations are only applicable at the root of a term, they cannot be treated like notions of reduction. In particular, it is impossible to form the compatible closure over them. Doing so would obviously introduce inconsistencies. On the other hand, these relations are needed to simulate the entire C-rewriting system. We therefore add these two rules to the transitive-reflexive closure of the reductions where they can do no harm. This yields the computation relation and the computational equivalence relation. We sometimes refer to this construction as the upper level of the calculus. From a different perspective the upper level of the calculus factors out all the steps in a rewriting sequence that must be in a linear sequence whereas the lower level allows a partial ordering of reduction steps. The two levels of the calculus may be understood as a characteristic of the imperative nature of the computations involved.

The first result about the λ_s-calculus is a substitution theorem:

Theorem 4.1 (Substitution).

(i) *If $V =_m U$, both are values, and $x \in Var_\lambda$ then $M[x := V] =_m M[x := U]$.*

(ii) *If $V =_m U$, both are values, $x \in Var_\sigma$, and n is a label (that may occur in M), then $M[x := V^n] =_m M[x := U^n]$.*

Proof. The proof is a straightforward induction on the structure of M. The hygiene condition for labels plays an important role. Without the knowledge that labels introduced by the $=_m$-step do not occur in M, the proof is invalid. □

The next question to be addressed is whether the λ_s-calculus is consistent. In other words, do the relations $\twoheadrightarrow_m$ and $\rhd_s$ satisfy the diamond property?

Theorem 4.2 (Church-Rosser).

(i) *The relation $\xrightarrow{m}$ is Church-Rosser.*

(ii) *The computation relation $\rhd_s$ satisfies the diamond property, i.e., if $M \rhd_s N$ and $M \rhd_s L$ then there exists a K such that $N \rhd_s K$ and $L \rhd_s K$.*

(iii) *If $M =_m N$ then there exists an L such that $M \twoheadrightarrow_m L$ and $N \twoheadrightarrow_m L$.*

(iv) *If $M =_s N$ then there exists an L such that $M \rhd_s^* L$ and $N \rhd_s^* L$.*

Proof. The proof of point (i) is an extension of the corresponding proof for the traditional λ-calculus [2]. For (ii) we prove that $\twoheadrightarrow_m$ commutes with $\rhd_\sigma$ and $\rhd_{\mathcal{D}}$. Both parts are tedious and not instructive. The last two points are consequences of the first two. □

Another question of interest about a calculus like λ_s is whether it defines an operational semantics, *i.e.*, if there is a standardization theorem. For the proof of this theorem we follow Plotkin's strategy and define a *standard reduction function (of type M)*, $\longmapsto_{sm}$, which always picks the leftmost-outermost redex and reduces it:

$$C[M] \longmapsto_{sm} C[N], \text{ if}$$
$$C[\ ] \text{ is an sk-context and } M \xrightarrow{m} N.$$

Definition 3: The λ_s-calculus

Let $\xrightarrow{m} = \xrightarrow{\beta_v} \cup \xrightarrow{\beta_e} \cup \xrightarrow{\sigma_L} \cup \xrightarrow{\sigma_R} \cup \xrightarrow{\mathcal{D}_L} \cup \xrightarrow{\mathcal{D}_R}$. Then define the *one-step M-reduction* $\longrightarrow_m$ as the compatible closure of $\xrightarrow{m}$:

$$\begin{aligned}
M \xrightarrow{m} N &\Rightarrow M \longrightarrow_m N;\\
M \longrightarrow_m N &\Rightarrow \lambda x.M \longrightarrow_m \lambda x.N;\\
M \longrightarrow_m N &\Rightarrow LM \longrightarrow_m LN', ML \longrightarrow_m N'L;\\
M \longrightarrow_m N &\Rightarrow (\sigma X.M) \longrightarrow_m (\sigma X.N');\\
M \longrightarrow_m N &\Rightarrow (\sigma M^n.L) \longrightarrow_m (\sigma N'^n.L) \text{ if } M \text{ and } N \text{ are values;}\\
M \longrightarrow_m N &\Rightarrow (\mathcal{D}\, X\, M) \longrightarrow_m (\mathcal{D}\, X\, N');\\
M \longrightarrow_m N &\Rightarrow (\mathcal{D}\, M^n\, L) \longrightarrow_m (\mathcal{D}\, N'^n\, L) \text{ if } M \text{ and } N \text{ are values;}
\end{aligned}$$

where L is an arbitrary term, X is a variable in Var_σ or a labeled value, and N' is a relabeled representative for N so that a (possible) β_σ argument in N is uniquely labeled. The *M-reduction* is denoted by $\twoheadrightarrow_m$ and is the transitive-reflexive closure of $\longrightarrow_m$. We denote the smallest congruence relation generated by $\twoheadrightarrow_m$ with $=_m$.
The *computation* $\rhd_s$ is defined by: $\rhd_s = \rhd_\sigma \cup \rhd_{\mathcal{D}} \cup \twoheadrightarrow_m$. The relation $=_s$ is the smallest equivalence relation generated by $\rhd_s$.

Standard reduction sequences (of type M) are then something like compatible closures of reduction sequences according to $\longmapsto_{sm}$. The computation relations are added on top of this construction:

$$\longmapsto_{ss} = \triangleright_{\sigma} \cup \triangleright_{\mathcal{D}} \cup \longmapsto_{sm} .$$

and they yield *standard reduction sequences (of type S)* for the entire calculus. Definition 4 formalizes these notions. They imply:

Theorem 4.3 (Standardization).

(i) $M \triangleright_s^* N$ *iff there exists an* S-SRS $L_1, \ldots, L_n$ *with* $M \equiv L_1$ and $L_n \equiv N$.

(ii) $M \longrightarrow\!\!\!\!\!\rightarrow_m N$ *iff there exists an* M-SRS $L_1, \ldots, L_n$ *with* $M \equiv L_1$ and $L_n \equiv N$.

Proof. The proof of the standardization theorem is a modification of the corresponding proof for the control calculus λ_c [6]. □

An important consequence is the following:

Corollary 4.4. $M \triangleright_s^* N$ *for some value* N *iff* $M \longmapsto_{ss}^* N'$ *for some value* N'.

This corollary says that the standard reduction function defines an operational semantics for the calculus or, in other words, the calculus is an abstract machine. We can prove the equivalence of the original CESK-machine and the standard reduction semantics of λ_s:

Theorem 4.5 (Simulation). *For any program* M *and value* V, $M \longmapsto_{ss}^* V$ *iff* $M \overset{mC}{\longmapsto}{}^* V$.

This first correspondence theorem is to some degree built into the calculus since we have derived it from the modified C-rewriting system. The theorem assures us that our calculus is correct with respect to the original semantics of assignments. An important consequence is that the mC-transitions are available within the calculus. This, in conjunction with the second correspondence theorem, greatly facilitates reasoning about programs.

The second correspondence theorem establishes a connection between the calculus and the equivalence of behaviors of program pieces [12, 6]. Space does not permit the development of all the necessary technical details. We appeal to the reader's intuition for an understanding of the notion of operational equivalence. Roughly speaking, two terms M and N are operationally equivalent, notation $M \approx N$, if there does not exist a (n arbitrary) program context—any program with a hole, not necessarily an sk-context—which can distinguish them hygienically, *i.e.*, without violating the hygiene condition for labels in M and N. Given this we state

Definition 4: Standard Reduction Sequences

The set of *standard reduction sequences of type M*, abbreviated M-SRS, contains all variables:

$$x \in Var_\lambda \Rightarrow x \text{ is an M-SRS;}$$

furthermore, it is closed under syntactic compatibility:

$$M_1, \ldots, M_k \text{ is an M-SRS} \Rightarrow$$
$$(\lambda x.M_1), \ldots, (\lambda x.M_k), (\sigma x.M_1), \ldots, (\sigma x.M_k), \text{ and } (\mathcal{D}xM_1), \ldots, (\mathcal{D}xM_k)$$
$$\text{are M-SRS's for appropriate variables } x;$$

and

$$M_1, \ldots, M_j \text{ and } N_1, \ldots, N_k \text{ are M-SRS's} \Rightarrow$$
$$M_1N_1, \ldots, M_jN_1, \ldots, M_jN_k \text{ is an M-SRS;}$$

and

$$M_1, \ldots, M_j \text{ and } V_1, \ldots, V_k \text{ are M-SRS's and the } V\text{'s are values,} \Rightarrow$$
$$(\sigma V_1^n.M_1), \ldots, (\sigma V_k^n.M_1), \ldots, (\sigma V_k^n.M_j) \text{ and}$$
$$(\mathcal{D}V_1^n M_1), \ldots, (\mathcal{D}V_k^n M_1), \ldots, (\mathcal{D}V_k^n M_j) \text{ are M-SRS's;}$$

and, finally, the set is closed under standard reductions:

$$M \longmapsto_{sm} M_1, \text{ and } M_1, \ldots, M_k \text{ is a M-SRS} \Rightarrow M, M_1, \ldots, M_k \text{ is an M-SRS.}$$

A *standard reduction sequence of type S*, S-SRS, is defined by:

$$M_1, \ldots, M_k \text{ is an M-SRS} \Rightarrow M_1, \ldots, M_k \text{ is an S-SRS;}$$
$$M \longmapsto_{ss} M_1 \text{ and } M_1, \ldots, M_k \text{ is an S-SRS} \Rightarrow M, M_1, \ldots, M_k \text{ is an S-SRS.}$$

Theorem 4.6 (Operational Equivalence). *Let M and N be in Λ_S. If $C[M] =_s C[N]$ for all sk-contexts $C[\ ]$, then $M \approx N$. Also, if $M =_m N$, then $M \approx N$.*

The two correspondence theorems play a key role in the proof of program properties. Some examples in the next section demonstrate how the two theorems interact and what kind of conclusions they allow.

5. Examples

Our first example illustrates the treatment of *circular* structures in our calculus—a topic that we have only alluded to formally. For an example we investigate the expression

$$(\lambda f.(\lambda g.(\sigma g.\mathcal{D}g\mathrm{I})(\lambda x.f(\mathcal{D}g\mathrm{I})x))\ulcorner 0\urcorner)F \text{ where } F \text{ is a value.}$$

An application of the first correspondence theorem shows that the evaluation in an arbitrary sk-context proceeds as follows:

$$\begin{aligned} &C[(\lambda g.(\sigma g.(\mathcal{D}g\mathrm{I}))(\lambda x.F(\mathcal{D}g\mathrm{I})x))\ulcorner 0\urcorner)] \\ =_s &C[(\sigma\ulcorner 0\urcorner^m.(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I}))(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I})x)] \\ =_s &C[\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I})x)^m\mathrm{I}]. \end{aligned}$$

At this point we have created a circular structure. The value $(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I})x)$ is labeled with m and also contains an m-labeled value (whose actual *contents* is irrelevant). We now perform the delabeling step and obtain

$$\ldots =_s C[\lambda x.F(\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I})x)^m\mathrm{I})x]. \qquad (*)$$

If the inner delabeling is ever performed, it will produce the same value. In some sense we have constructed an assignment-delabeling-by-need system.

Beyond a demonstration of how circular structures are maintained in the calculus the first example points to an interesting connection between circular structures and the implementation of recursion. Let us introduce the abbreviation

$$Z \equiv \lambda f.(\lambda g.(\sigma g.\mathcal{D}g\mathrm{I})(\lambda x.f(\mathcal{D}g\mathrm{I})x))\ulcorner 0\urcorner.$$

Given Z, we have shown in the preceding paragraph that according to the second correspondence theorem

$$ZF \approx \mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I})x)^m\mathrm{I}.$$

By the definition of operational equivalence we can replace two indistinguishable terms in any context. In particular, we can replace $\mathcal{D}(\lambda x.F(\mathcal{D}\ulcorner 0\urcorner^m\mathrm{I})x)^m\mathrm{I}$ in (*) by ZF and by transitivity we get

$$ZF \approx \lambda x.F(ZF)x.$$

This is the same operational equation that the by-value recursion operator $\mathrm{Y}_v \equiv \lambda fx.(\lambda\alpha.\alpha\alpha)(\lambda\alpha.f(\lambda x.\alpha\alpha x))x$ for functionals satisfies. Although this does not imply that the two recursion operators are equivalent, we feel more justified using Z for the implementation of recursive function definitions.

Next we return to the examples of the introduction. For the transliteration of these program pieces into Λ_S we assume that all *relevant* variables are in Var_σ. As usual, blocks are translated into applications of λ-abstractions to the initial values [8]. Thus the program piece

$$\textbf{begin var } x = 0;\ \textbf{skip end}$$

becomes

$$(\lambda x.\mathrm{I})\ulcorner 0\urcorner$$

where the function I is an arbitrary representation of the command **skip**. It follows that this block is equivalent to I, *i.e.*, **skip**:

$$(\lambda x.\mathrm{I})\ulcorner 0\urcorner =_m \mathrm{I}[x := \ulcorner 0\urcorner^l] \equiv \mathrm{I}.$$

The labeled value ($\ulcorner 0\urcorner^l$) simply disappears.

It is similarly trivial to show that the sequence of declarations in a block header plays no role. With the natural transliteration for

$$\textbf{begin var } x = 0,\ y = 1;\ \langle cmd\rangle\ \textbf{end}$$

we get

$$\begin{aligned} (\lambda xy.M)\ulcorner 0\urcorner\ulcorner 1\urcorner &=_m M[x := \ulcorner 0\urcorner^n][y := \ulcorner 1\urcorner^m] \\ &\equiv M[y := \ulcorner 1\urcorner^k][x := \ulcorner 0\urcorner^l] \\ &=_m (\lambda yx.M)\ulcorner 1\urcorner\ulcorner 0\urcorner. \end{aligned}$$

The last expression represents

$$\textbf{begin var } y = 1,\ x = 0;\ \langle cmd\rangle\ \textbf{end.}$$

The important point is that each β_σ-step introduces a *fresh* label and the identity of these labels does not matter.

The last example is derived from a problem posed by Halpern *et al*[7]. Suppose the expression

$$\textbf{let } x = 0 \textbf{ in let } d = p(0) \textbf{ in } x$$

is given and the additional information that it is always used in a context where the free variable p is bound to a terminating function. The result of this block should be 0; the side-effects are those of the invocation of p. The question is whether this is provable for all possible cases. The answer is yes:

$$\begin{aligned} &C[(\lambda x.(\lambda d.(\mathcal{D}x\mathrm{I}))(p\ulcorner 0\urcorner))\ulcorner 0\urcorner] \\ =_s &C[(\lambda d.(\mathcal{D}\ulcorner 0\urcorner^l\mathrm{I}))(p\ulcorner 0\urcorner)] \text{ where } l \text{ is fresh} \\ =_s &C'[(\mathcal{D}\ulcorner 0\urcorner^l\mathrm{I})] \\ &\quad \text{since } p \text{ always terminates, possibly with side-effects} \\ =_s &C'[\ulcorner 0\urcorner] \\ =_s &C[(\lambda d.\ulcorner 0\urcorner)(p\ulcorner 0\urcorner)]. \end{aligned}$$

In other words, the above expression is operationally indistinguishable from

$$\textbf{let } d = p(0) \textbf{ in } 0.$$

The hygiene condition for labels plays a crucial role in this proof. No matter what expression is substituted for the free variable p, it cannot contain a reference to x and, hence, it cannot access the label l.

The preceding proof is also possible in the framework of denotational semantics. The notion of a cover as introduced by Halpern *et al* [7] supports this proof in a nice way. The *cover* of a command is roughly the set of all accessible locations. It is used to describe the correct way to allocate a new location. Their central condition is then that new locations which are allocated at the entry of a block must be from outside of the cover of the block body. This corresponds to our own requirement that a label is fresh and that in any context we need to make sure that there is no interference with labels from other terms. For example, in the above block these two conditions enforce that whatever we substitute for p does not invalidate the β_σ step. The freshness and the hygiene condition are simpler because we can always decide about them by inspection of some finite terms. Another factor which contributes to this simplicity is that programs in Λ_S have no control over locations. We feel that this is justified. Locations are a relic of the time when language design was driven by a particular machine architecture.

Although important, the hygiene condition is not the central point of our development. In the next section we discuss research efforts that are directly related to the central objectives of our work.

6. Summary and related work

In the preceding sections we have shown that

- we can derive a calculus of assignments from a control-string-store rewriting system,
- this calculus is correct, and
- its standard reduction function defines a location-free rewriting semantics for a higher-order programming language with assignments.

This development has two central consequences. On the practical side we have demonstrated that there is a semantics of assignment languages which can be directly implemented on rewriting machines. The existence of the calculus and of standard reduction sequences shows that even in an assignment language events are not ordered in a linear sequence. The two levels of the calculus cleanly separate the classes of actions which must happen in totally or in partially ordered sequences. On the theoretical side we have an improved understanding of assignment in programming languages. Potential uses are in the area of program proofs and transformational programming.

Mason's work [9] comes closest in spirit to the theoretical aspect of our work. He is interested in first-order Lisp programs which destructively change the store. His work is based on a Plotkin-style computational theory with control string-store configurations—memory objects in his terminology. Since the language is restricted to the class of first-order Lisp-programs, the theory is—except for access to the addresses of locations—a special case of the CS-machine in Section 2. An operational equivalence relation is then defined based on this operational semantics, *i.e.*, a relation which compares Lisp programs relative to their *results*. Several extensions of this base relation compare various extensional and intensional aspects of destructive and purely functional programs. The memory object isomorphism equivalence captures the spirit of our $\equiv_{lab}$-relation; indeed, on this level our own operational equivalence relation à la Theorem 4.6 and Mason's relation have comparable power. Our primary advantage is our choice of the primitive assignment and a higher-order language as the starting point. The memory object isomorphism cannot deal with the examples of the preceding section. Furthermore, with the lower level, *i.e.*, $=_m$, we have a more fine grained equivalence relation, although we do not yet know if this yields any additional possibilities for the understanding of assignments. It is not clear how the other relations fit into our system since they explicitly refer to locations and garbage collection. The paper does not include a calculus for reasoning about these equivalences, nor is there any attempt to eliminate the explicit store.

Landin's idea of a sharing machine [8] anticipated our perception of the store as an equivalence relation. The sharing machine is an extended SECD-machine. The additional component is an equivalence relation which designates certain parts of the state as equivalent. When an update occurs, all equivalent components of the state are changed to contain the new value. This is clearly the same idea except that it is applied to a 4-tuple rewriting system or abstract machine. Landin himself did not foresee the possibility of applying this idea to terms. He feared that "the meaning of an IAE [imperative applicative expression], ... is completely dependent on an abstract machine."[5]

Other efforts to eliminate the concept of locations from semantics have concentrated on imperative first-order languages [1, 3, 5]. The basic approach models the store directly with the environment and uses auxiliary means such as equivalence relations or update continuations in order to realize more complicated features like aliasing and call-by-name procedures. We have not yet seen any generalizations to the case of higher-order languages.

[5] See [8], page 92.

Beyond these developments in store semantics there is of course a bulk of work on the Floyd-Hoare logic of programs. The relationship to this work is not clear, but certainly merits consideration.

Our investigations have demonstrated that assignments in higher-order languages can be understood with a rather simple model. There are many loose ends to be explored, but we feel that this calculus offers new opportunities to reassess the role of assignments in programming languages.

Acknowledgement. Sussman and Steele's paper "Scheme: an interpreter for *extended* λ-calculus"[6] stimulated us to consider how such a calculus for assignments in higher-order languages could be constructed. Bruce Duba, Carolyn Talcott, and Mitch Wand provided helpful comments on earlier drafts of the paper. Matthias Felleisen is supported by an IBM Graduate Research Fellowship. The work is also supported in part by the NSF grants DCR 85 01277 and DCR 83 03325.

References

1. ABDALI, S.K., D.S. WISE. Storeless semantics for ALGOL-style block structure, *Proc. Conf. Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1985, to appear.

2. BARENDREGT, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

3. BROOKES, S.D. A fully abstract semantics and a proof system for an Algol-like language with sharing, *Proc. Conf. Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1985, to appear.

4. DE BRUIJN, N.G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the Church-Rosser theorem, *Indagationes Mathematics* 37, 1972, 381–392.

5. DONAHUE, J.E. Locations considered unnecessary, *Acta Informatica* 8, 1977, 221–242.

6. FELLEISEN, M., D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus, *Formal Description of Programming Concepts III*, North-Holland, Amsterdam, 1986, to appear.

7. HALPERN, J.Y., A.R. MEYER, B.A. TRAKHTENBROT. The semantics of local storage, or What makes the free-list free?, *Proc. 11th ACM Symp. Principles of Programming Languages*, 1984, 245–257.

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

9. MASON, I. A. Equivalences of first-order Lisp programs, *Proc. First Symp. Logic in Computer Science*, 1986, 105–117.

10. MORRIS, J.H. Protection in programming languages, *Comm. ACM* 16(8), 1973, 15–21.

11. PLOTKIN, G.D. A structural approach to operational semantics, Tech. Rpt. DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

12. PLOTKIN, G. D. Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science* 1, 1975, 125–159.

[6] Memo 349, MIT AI-Lab, 1975.