A Reduction Semantics For Imperative Higher-Order Languages

By

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

TECHNICAL REPORT NO. 218

A Reduction Semantics
For Imperative Higher-Order Languages

by

Matthias Felleisen & Daniel P. Friedman

March, 1987

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 *Parallel Architectures and Languages Europe*, (de Bakker, Nijman & Treleaven, eds.), Lecture Notes in Computer Science, Vol. 259, Springer, Berlin, (1987) 206–223.

A Reduction Semantics for Imperative Higher-Order Languages

Matthias Felleisen, Daniel P. Friedman

Computer Science Department, Indiana University, Bloomington, IN 47405, USA

Abstract

Imperative higher-order languages are a highly expressive programming medium. Compared to functional programming languages, they permit the construction of safe and modular programs. However, functional languages have a simple reduction semantics, which makes it easy to evaluate program pieces in parallel. In order to overcome this dilemma, we construct a conservative extension of the λ-calculus that can deal with control and assignment facilities. This calculus simultaneously provides an algebraic reasoning system and an elegant reduction semantics of higher-order imperative languages. We show that the evaluation of applications can still take advantage of parallelism and that the major cost of these evaluations stems from the necessary communication for substitutions. Since this is also true for functional languages, we conjecture that if a successful parallel evaluation scheme for functional languages is possible, then the same strategy will also solve the problem of parallel evaluations for imperative programming languages.

1. *Pro* Imperative Higher-Order Languages

A programming language is a medium for expressing thoughts about problems and their solutions. This statement is folk wisdom, yet, it has been ignored since programming became an activity performed on real machines. In the beginning, programming languages were considered as command languages for computers. This view grew out of the popular imperative programming languages for the early computing machines and the necessity for maximal utilization of scarce resources.

Another phase in programming language research was determined by the advent of non-von-Neumann computer architectures. The realization was that the traditional way of processing programs had a bottleneck and that this bottleneck should be eliminated in favor of as much parallel processing as possible. But, instead of implementing redesigned traditional languages on these modern machines, new languages were invented. The prevailing opinion was [4, 17] that "increasing performance by exploiting parallelism must go hand in hand with making the programming of *these* machines easier."[1] This argument, together with a trend for more mathematical languages, ignited interest in applicative languages [1].

Applicative languages are easy to implement on non-traditional reduction architectures. They have a simple operational model based on reduction semantics. However, these languages lack abstractions for expressing evaluation control and state change because these facilities invalidate ordinary reduction semantics and thus complicate parallelization of program evaluations. Programmers must simulate these imperative effects in applicative languages by using and optimizing tricks of denotational semantics, *e.g.*, accumulators, auxiliary functions, or the clumsy passing around of state variables. Again, the burden is borne by the programmer.

Our basic premise is that the language user should not feel any restrictions caused by the underlying implementation machine. We agree that a language must have a clean, mathematical reasoning system, but, we also insist that a language must include control and assignment facilities as fundamental abstractions for expressing evaluation control and state change. They are necessary ingredients for secure, modular, and compositional programming.

The starting point of our development is the λ-calculus [2], more precisely, the λ-value-calculus [20], which is simultaneously a language and a reasoning system. Its advantage is that all objects, including functions, are first-class, *i.e.*, there is no restriction on their use. The programming language contains two additional facilities which preserve this property. One gives full access and control over

[1] [17, p.350], emphasis ours.

the current continuation of an expression; the other abstracts the right to reassign a value to a variable. Programming paradigms such as logic, functional, or object-oriented programming are easily emulated in this language by implementing the respective constructs as syntactic abstractions [3, 13].

Recently [6, 7] we presented two extensions of the λ_v-calculus which independently model control and assignment facilities. Here we demonstrate that the two extensions can be unified. The new calculus automatically yields a reduction semantics for imperative higher-order languages. The standard reduction strategy of the calculus reveals considerable potential for parallelism in the evaluation of programs. A minor result is the introduction of a new control construct. While simplifying the reductions of the control calculus [6], it syntactically subsumes all traditional control constructs and permits the design of an entirely new class of programs [5].

The emphasis here is on the development of the reduction semantics since it is crucial for the parallel implementation of programs. The reader who is more interested in the proof system is referred to the earlier reports,[2] but the relevant material is repeated here for completeness. In the next section we formalize the syntax and semantics of the programming language and demonstrate with a few examples how some commonly found facilities of other languages are simple syntactic abstractions. The third section contains the transformation of our abstract machine semantics into a program rewriting system. The derivation method is new; some of the intermediate stages are remotely related to Plotkin-style operational semantics [19]. In Section 4 we reformulate the rewriting system as a set of freely applicable notions of reduction. Section 5 addresses implementation issues. An alternative definition of standard evaluation reveals that the reduction system for the programming language offers ample opportunity for evaluating programs in parallel.

2. The Programming Language

The programming language is an idealized version of such imperative higher-order programming languages as ISWIM [14], GEDANKEN [22], and Scheme [23]. Its term set $\Lambda_{\mathcal{F}\sigma}$ is an extension of the term set Λ of the λ-calculus. The two new kinds of expressions are $\mathcal{F}$-applications and σ-abstractions. An $\mathcal{F}$-application is of the form $(\mathcal{F}M)$ where M is an arbitrary expression; when evaluated, it applies M to a functional abstraction of its current continuation, *i.e.*, a functional representation of the rest of the computation. These abstractions have the same status and behavior as functions created by λ-abstractions. The syntax of a σ-abstraction is $(\sigma x.M)$ for a variable x and a term M. The abstraction does not bind the variable, but abstracts the right to reassign a value to the variable. When invoked on a value, a σ-abstraction performs the reassignment and then continues to evaluate its body, which yields a result for the entire application. The meaning of the remaining constructs should be adapted accordingly: variables are 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 2.1.

The sets of free and bound variables of a term M, $FV(M)$ and $BV(M)$, are defined as usual; the only binding construct in the language is the λ-abstraction. The set of assignable variables in M, $AV(M)$, contains all variables that occur in the variable position of a σ-abstraction. Terms with no free variables are called *closed terms* or *programs*. To avoid syntactic issues, we adopt Barendregt's *α-congruence convention* of identifying ($\equiv_\alpha$ or just $\equiv$) terms that are equal modulo some renaming of bound variables and his *hygiene convention* 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

[2] C. Talcott [24] and I. Mason [18] also have designed reasoning systems for control and [first-order] assignment abstractions, respectively. However, their systems are not extensions of λ-calculi, but are equational theories based on rewriting machines similar to the ones we present in Section 3. Neither addresses the issue of reduction systems; work on a unification of the two systems is in progress [personal communication].

Definition 2.1: The programming language $\Lambda_{\mathcal{F}\sigma}$

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

— *variables:* x if $x \in Vars$;

— *abstractions:* $(\lambda x.M)$ if $M \in \Lambda_{\mathcal{F}\sigma}$ and $x \in Vars$;

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

— *$\mathcal{F}$-applications:* $(\mathcal{F}M)$ if $M \in \Lambda_{\mathcal{F}\sigma}$;

— *σ-abstractions:* $(\sigma x.M)$ if $x \in Vars$ and $M \in \Lambda_{\mathcal{F}\sigma}$.

The union of variables, abstractions, and σ-abstractions is referred to as the set of *(syntactic) values.*

Definition 2.2: The CESK-transition function

$$\begin{aligned}
\langle x, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\,\mathbf{ret}\ \theta(\rho(x)))\rangle && (1)\\
\langle \lambda x.M, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\,\mathbf{ret}\ \langle\lambda x.M, \rho\rangle)\rangle && (2)\\
\langle MN, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle M, \rho, \theta, (\kappa\,\mathbf{arg}\ N\,\rho)\rangle && (3)\\
\langle \ddagger, \emptyset, \theta, ((\kappa\,\mathbf{arg}\ N\,\rho)\,\mathbf{ret}\ F)\rangle &\overset{CESK}{\longmapsto} \langle N, \rho, \theta, (\kappa\,\mathbf{fun}\ F)\rangle && (4)\\
\langle \ddagger, \emptyset, \theta, ((\kappa\,\mathbf{fun}\ \langle\lambda x.M, \rho\rangle)\,\mathbf{ret}\ V)\rangle &\overset{CESK}{\longmapsto} \langle M, \rho[x := n], \theta[n := V], \kappa\rangle && (5)\\
&\qquad\text{where } n \notin Dom(\theta)\\
\langle \mathcal{F}M, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle M\gamma, \rho[\gamma := \langle \mathrm{p}, \kappa\rangle], \theta, (\mathbf{stop})\rangle && (6)\\
\langle \ddagger, \emptyset, \theta, ((\kappa\,\mathbf{fun}\ \langle \mathrm{p}, \kappa_0\rangle)\,\mathbf{ret}\ V)\rangle &\overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa \otimes \kappa_0\,\mathbf{ret}\ V)\rangle && (7)\\
\langle \sigma x.M, \rho, \theta, \kappa\rangle &\overset{CESK}{\longmapsto} \langle \ddagger, \emptyset, \theta, (\kappa\,\mathbf{ret}\ \langle\sigma x.M, \rho\rangle)\rangle && (8)\\
\langle \ddagger, \emptyset, \theta, ((\kappa\,\mathbf{fun}\ \langle\sigma x.M, \rho\rangle)\,\mathbf{ret}\ V)\rangle &\overset{CESK}{\longmapsto} \langle M, \rho, \theta[\rho(x) := V], \kappa\rangle. && (9)
\end{aligned}$$

denote the result of substituting all free variables x in M by N.

The semantics of $\Lambda_{\mathcal{F}\sigma}$-programs is defined via an abstract state-transition machine. The machine manipulates quadruples of control strings, environments, stores, and continuations. A *control string* is either the symbol $\ddagger$ or a $\Lambda_{\mathcal{F}\sigma}$-expression. A variable γ is reserved for exclusive use in machine transitions. *Environments*, denoted by ρ, are finite maps from variables to semantic values and locations. *Stores*, denoted by θ, are finite maps from locations and semantic values to semantic values; stores are the identity map on values. We use the set of natural numbers as locations. If f is an environment or store, then $f[x := y]$ is like f except at the place x where it is y. The set of *semantic values* is the union of closures and program points. A *closure* 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. Depending on the kind of abstraction, a closure is called λ- or σ-closure. A *program point* is a p-tagged continuation code.

A *continuation code* remembers the remainder of the computation during the evaluation of the current control string. The domain of continuations consists of two subdomains: p- and **ret**-continuations. A **ret**-continuation $(\kappa\,\mathbf{ret}\ V)$ consists of a p-continuation code κ and a semantic value V. The value is supplied to the p-continuation so that it can finish whatever is left to do. p-Continuations are defined inductively and have the following intuitive function with respect to an evaluation:

— $(\mathbf{stop})$ stands for the initial continuation, specifying that nothing is left to do;

— $(\kappa\,\mathbf{arg}\;N\rho)$ indicates that N is the argument part of an application, that ρ is the environment of the application, and that κ is its continuation;

— $(\kappa\,\mathbf{fun}\;V)$ represents the case where the evaluation of a function part yielded V as a value, and κ is the continuation of the application.

Machine states of the form $\langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle$ are *initial*; if V is a closure and θ is defined on all locations which are used by V then $\langle \ddagger, \emptyset, \theta, ((\mathbf{stop})\,\mathbf{ret}\;V)\rangle$ is a *terminal* state.

The CESK-transition function is displayed in Definition 2.2. The first five rules evaluate the functional subset of $\Lambda_{\mathcal{F}\sigma}$. They correspond to the transition rules of the SECD-machine [16]. The store component is superfluous with respect to this subset and could be merged with the environment. (CESK6) and (CESK7) describe the evaluation of an $\mathcal{F}$-application and the application of a continuation to a value; (CESK8) and (CESK9) define the effect of an assignment application.

The evaluation of an $\mathcal{F}$-application directly corresponds to the informal description of $\mathcal{F}$. The $\mathcal{F}$-argument is applied to γ, which stands for the current continuation. The current continuation is transferred out of the register into a p-closure and this gives the program total control over its use. In particular, the decision about when to use the continuation is left to the program.

The application of a continuation in a $\Lambda_{\mathcal{F}\sigma}$-program results in the concatenation of the applied continuation to the current one with the auxiliary function $\otimes$:

$$\begin{aligned} \kappa \otimes (\mathbf{stop}) &= \kappa \\ \kappa \otimes (\kappa'\,\mathbf{arg}\;N\;\rho) &= (\kappa \otimes \kappa'\,\mathbf{arg}\;N\;\rho) \\ \kappa \otimes (\kappa'\,\mathbf{fun}\;V) &= (\kappa \otimes \kappa'\,\mathbf{fun}\;V) \end{aligned}$$

This causes the machine to evaluate the applied continuation as if it were a function. If the applied continuation does not affect its continuation during the evaluation, the result is returned to the point of application.

The effect of an assignment application depends on three different factors. First, occurrences of variables are disambiguated via the environment, but the store contains the associated current value. Second, the store component of a CESK-evaluation is always present. The only operations on the store are updates and extensions. Unlike the environment, it never shrinks nor is it removed from its register. Third, the rules (CESK8) and (CESK9) jointly manipulate the location-value association in the store according to an intuitive understanding of assignment. (CESK8) produces a σ-closure from a σ-abstraction and the current environment. Thus the transition rule (CESK9) changes the value of the variable that was lexically visible at definition time. Because of the constant presence of the store, every subsequent occurrence of this variable refers to the new value.

In order to abstract from the machine details, we define an evaluation function that maps programs to value-store pairs:

$$eval_{CESK}(M) = \langle V, \theta\rangle \text{ iff } \langle M, \emptyset, \emptyset, (\mathbf{stop})\rangle \overset{CESK^+}{\longmapsto} \langle \ddagger, \emptyset, \theta, ((\mathbf{stop})\,\mathbf{ret}\;V)\rangle.$$

When we refer to the extensional semantics of $\Lambda_{\mathcal{F}\sigma}$, we mean this $eval_{CESK}$-function. The intension behind this function with respect to the CESK-machine is the CESK-transition function. In other words, $eval_{CESK}$ defines *which* value-store pair is the result of a program, and the transition function says *how* this result is computed.

The distinction between extensional and intensional semantics is extremely important for our development. From the extensional point of view, which is that of a programmer, the programming language is—exactly like the λ-calculus—an entirely sequential language. To an outside observer this means that events in an evaluation are totally ordered. No function in our language nor in the λ-calculus can thus compute the mathematical (symmetric) or-function. However, the sequentiality of our intensional semantics depends on the granularity of the event representation. We shall see a number of different machine architectures, each of which is extensionally equivalent to, but intensionally rather different from the CESK-machine.

At this point, we clarify the formal semantics with a few programming examples. Beyond providing insight into the semantics, these examples also illustrate some of the expressiveness of imperative higher-order languages. Lack of space prohibits broader treatment of this issue. The interested reader is referred to the literature.

$\mathcal{F}$-applications are closely related to the call/cc-function in Scheme [3]. There are two essential differences. First, when call/cc is applied to a function, it provides this function access to the current continuation, but it also leaves the continuation in its register. This effect can be achieved by an $\mathcal{F}$-application if the $\mathcal{F}$-argument immediately invokes the continuation. Second and more important, call/cc applies its argument to a continuation object, which, when applied in turn, discards the current continuation; an $\mathcal{F}$-application, however, simply provides its argument with a function that upon invocation performs the same action as the current continuation. Hence, a simulation of call/cc must pass a function to the call/cc argument which ignores its current continuation, or, in terms of $\mathcal{F}$-applications, the object must grab the continuation without using it. Putting all this together produces the following equivalence:

$$\mathsf{call/cc} \equiv \lambda f.\mathcal{F}(\lambda k.k(f(\lambda v.\mathcal{F}(\lambda d.kv)))).$$

Landin's J-operator [15] and Reynolds's **escape**-construct [21] are also closely related language facilities. Both are syntactic variations on call/cc [8] and hence, we omit detailed treatment. None of these facilities can implement an $\mathcal{F}$-application as a syntactic abstraction because of the abortive effect of their continuation objects.

The assignment abstraction is more conventional. In traditional expression-oriented languages, statements usually come together with a block like **begin** $\langle stmt \rangle$ **result** $\langle exp \rangle$. This is a block that first performs the statement-part and then evaluates the expression-part to return a result. Together with ordinary assignment this block can express an assignment abstraction as a function:

$$\sigma x.M \equiv (\lambda v.(\textbf{begin}\ x := v\ \textbf{result}\ M))$$
$$\text{where } v \text{ is a fresh variable.}$$

The inverse relationship is expressed by

$$\textbf{begin}\ x := N\ \textbf{result}\ M \equiv (\sigma x.M)N.$$

The choice of σ-abstractions over assignment statements is for syntactic and proof-technical reasons only.

The implementation of call/cc and ordinary assignment as syntactic abstraction makes clear that all programs that were written with these facilities can also be written in $\Lambda_{\mathcal{F}\sigma}$. However, there are control constructs that are more easily defined with $\mathcal{F}$. For example, the operation halt, which is implicitly used in the definition of call/cc, is realized by grabbing and ignoring the current continuation:

$$\mathsf{halt} \overset{df}{\equiv} \lambda x.\mathcal{F}(\lambda d.x).$$

An exit-facility in a function-body is an equally simple use of $\mathcal{F}$. Suppose we want an abstraction

$$(\textbf{function}\ x\ \mathit{Body})$$

which is like an ordinary function, except that its function body may contain **exit**-expressions of the form

$$(\textbf{exit } Exp).$$

The effect of an **exit**-expression is an immediate return to the function caller with the value of Exp. When cast into continuation terminology, this description leads to the obvious implementation of **function**- and **exit**-expressions: an **exit**-expression resumes the continuation of the **function**-application. With call/cc we get:

$$(\textbf{function } x \; Body) \overset{df}{\equiv} \lambda x.\text{call/cc}(\lambda \epsilon. Body_{\epsilon}),$$
$$(\textbf{exit } Exp) \overset{df}{\equiv} (\epsilon \; Exp).$$

We hereby assume that **exit**'s ϵ and the ϵ in the call/cc-expression are the same. Although all of these programming constructs look functional, the reader should be aware that the presence of $\mathcal{F}$ makes them imperative.

Assignment abstractions are also imperative in nature, but they give rise to a different class of programs. In conjunction with higher-order functions it is easy to program reference cells with σ-abstractions. The three operations on a cell are: mk-cell, which creates a new cell with given contents; deref, which looks up the current contents of a cell; and set-cell, which changes the contents of a cell to a new value and returns this value upon completion. An implementation of this set of functions is:

$$\text{mk-cell} \overset{df}{\equiv} \lambda x.\lambda m.mx(\sigma x.x),$$
$$\text{deref} \overset{df}{\equiv} \lambda c.c(\lambda xs.x),$$
$$\text{set-cell} \overset{df}{\equiv} \lambda c.c(\lambda xs.s).$$

The definition of set-cell clarifies why we call σ-abstraction "an abstraction of the right to reassign a value to x." When set-cell is applied to a cell, it returns a σ-abstraction, which upon application changes the content of the cell.

The implementation strategy for single-value cells generalizes to full Lisp-cons cells and Smalltalk-objects in a straightforward manner. Furthermore, with mutable cells a program can create circular structures to model self-referential dependencies in the real-world. A use of state variables for remembering the current control state of a computation leads to simple implementations of such facilities as coroutines [10], backtrack points [9], and relations [11].

In summary, we have demonstrated that the programming language (together with an appropriate syntax preprocessor [13]) is interesting and expressive. Its semantics is defined via an abstract machine that is derived from a denotational specification via a well-explored technique [12, 21, 25]. An implementation of the machine on stock hardware is straightforward and reasonably efficient. Given its character as a rewriting system, a reduction or a rewriting machine could also serve as an implementation vehicle, but, such an implementation could not utilize the inherent parallelism in these machines. The intensional specification of the semantics is sequential in nature and any attempt to exploit parallelism on the basis of these specifications remains *ad hoc*. The underlying reason is that the structure of the machine states and the state representations are too closely tied to conventional architectures. However, we doubt that any of the state components besides the control string is *natural* in the sense that it is unavoidable for the specification of the language semantics. An elimination of these extra components can clearly improve the comprehensibility of the machine activities and should give more insight into the intensional aspects of the language semantics.

3. From the CESK-Machine to a Control String-Rewriting System

An evaluation of a program on the CESK-machine is a sophisticated interplay between the control string and the other three auxiliary components. In the course of program evaluation, parts of the control string are mapped to values in the environment-store system, others become a part of the continuation code. At other points control strings are shuffled back from the auxiliary components into the control string register. Our objective in this section is to integrate the extra state components into the control string by extending the programming language.

3.1. Eliminating Environments

Environments map free variables of a control string to their meaning. They directly correspond to stacks on traditional machines. For machines which only simulate stacks, quasi-substitution generally improves the performance. Instead of storing away the meaning of free variables, quasi-substitution replaces all free variables in a control string by their meaning. For the CESK-machine this means that applications of λ-closures to values now place values and locations in the control string. The $\Lambda_{\mathcal{F}\sigma}$-language is modified accordingly. Otherwise, the system remains the same.

3.2. Eliminating Continuations

An inspection of some sample evaluations on the CESK-machine reveals that the continuation code is like a control string memory. Those pieces of the control string that are of no interest to the current computation phase are shifted to the continuation stack. This is equally true for (syntactic) values in the control string register. Such a step causes the machine to look in the continuation code for what to do next. In case the two parts of an application have been identified as proper values, the appropriate action, *i.e.*, a function, continuation, or assigner invocation, is performed; otherwise, the search is continued.

From the perspective of the program as a whole, the machine searches through the term until it finds a value in the left part of an application. Then it backs up and continues the search in the right part of the application. Once an application with two values—a *redex*—has been found, a computation step changes the corresponding subterm. During the search phase the *context* of the redex is moved to the continuation. In other words, the p-continuation encodes the textual context of a redex, and it is therefore quite natural to represent p-continuation codes as contexts. Informally, an applicative context—or sometimes context for short—is a term with a hole in it such that the path from the root to the hole goes through applications only. Let $C[\ ]$ stand for a context, then the inductive definition is:[3]

— $[\ ]$ is the empty context or hole,

— $VC[\ ]$ is a context where V is a value, and

— $C[\ ]M$ is a context where M is arbitrary.

If $C[\ ]$ is a context, then $C[M]$ is the term where the hole is filled with the term M.

The definition of applicative contexts captures the dynamic character of the search phase in the correct way. The context hole corresponds to the current search area. If a value is in the left part of an application, the search moves to the right; otherwise, the search space is narrowed down to the left part.

Once the continuation representation uses contexts, the control string and the continuation component can be merged. First, the holes of the contexts are filled with the respective term, $\ddagger$'s are simply dropped. Then all the search rules are eliminated. These two steps together yield a control string-store machine whose transition function is displayed in Definition 3.1. All other concepts are modified appropriately.

3.3. The Control String-Rewriting System

After the elimination of environments and continuations we are left with a system that is solely based on control strings and stores. The role of the store is characterized by the three transition rules (CS1), (CS2), and (CS5): they use, extend, and modify the store. The crucial rule is (CS2). It replaces bound variables of a λ-abstraction by a new, distinct location and thus gradually builds the store. Future

[3] Generally, contexts are defined in a broader sense, *i.e.*, as a term with at least one hole and no restriction on the location of the hole. Since we only use applicative contexts in this paper, the terminology should not cause any confusion.

Definition 3.1: The CS-machine

$$\langle C[n],\theta\rangle \overset{CS}{\longmapsto} \langle C[\theta(n)],\theta\rangle \tag{1}$$
$$\langle C[(\lambda x.M)V],\theta\rangle \overset{CS}{\longmapsto} \langle C[M[x:=n]],\theta[n:=V]\rangle \quad \text{where } n \notin Dom(\theta) \tag{2}$$
$$\langle C[\mathcal{F}M],\theta\rangle \overset{CS}{\longmapsto} \langle M\langle \mathrm{p}, C[\;\;]\rangle,\theta\rangle \tag{3}$$
$$\langle C[\langle \mathrm{p}, C_0[\;\;]\rangle V],\theta\rangle \overset{CS}{\longmapsto} \langle C[C_0[V]],\theta\rangle \tag{4}$$
$$\langle C[(\sigma n.M)V],\theta\rangle \overset{CS}{\longmapsto} \langle C[M],\theta[n:=V]\rangle \tag{5}$$

references to a bound variable are resolved via the always-present store. A request for the current value of a variable causes a store lookup; an instruction to change the current value results in a modified store. It is therefore important to understand the nature of bound variables.

At this point we must recall the α-congruence convention about bound variables in terms. According to this convention, the name of a bound variable is irrelevant (up to uniqueness). Abstractions like $\lambda x.x$ and $\lambda y.y$ are considered to be the same. This actually means that the programming language is the quotient of Λ over $\equiv_\alpha$. From this perspective, a λ-abstraction is an expression together with a relation that determines which parts of the expression are equivalent. The relationship is manifested by occurrences of the bound variable.

A unification of our ideas on the role of the CS-store and the nature of bound variables directly leads to an abstracted view of the store. The intention behind the replacement of bound variables by unique locations in the bodies of λ-abstractions is to retain the *static* α-equivalence for the rest of the computation, *i.e.*, as a *dynamic* relation, even after the λx.-part has disappeared. Hence, if we want to integrate the store into the control string language, we must find a way to express this dynamic relationship.

The most natural possibility is a labeling scheme. Instead of placing a location into the program text and the value into the store, the value could be labeled with the location and placed into the text as a *labeled value*.[4] With respect to the transition relation, we replace

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

by

$$C[(\lambda x.M)V] \longmapsto C[M[x:=V^n]] \text{ where } n \text{ is not used in the rest of the program.}$$

V^n is the labeled version of the value V. With this labeling technique it is indeed possible to re-interpret lookups and assignments as term manipulations.

The effect of an assignment relies on the introduction of a new substitution operation. In the extended term language, a σ-application looks like

$$(\sigma U^n.M)V$$

when it is about to be evaluated. The assignable variable has been replaced by a labeled value; all other related variable positions carry the same label. When it is time to perform the above σ-application, all these occurrences of an n-labeled value must be replaced by n-labeled values V. To implement this, we introduce the labeled value substitution $M[n:=V]$. The result of $M[n:=V]$ is a term like M except

[4] Labeled terms have also been used for the investigation of the regular λ-calculus [2, p.353]. Although our problem is unrelated to these investigations we have adopted the notation in order to avoid the introduction of an entirely new concept.

Definition 3.2: The C-rewriting system

$$C[V^n] \overset{C}{\longmapsto} C[V[n := V]] \tag{1}$$

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

$$C[\mathcal{F}M] \overset{C}{\longmapsto} M\langle \mathrm{p}, C[\ ]\rangle \tag{3}$$

$$C[\langle \mathrm{p}, C_0[\ ]\rangle V] \overset{C}{\longmapsto} C[C_0[V]] \tag{4}$$

$$C[(\sigma U^n.M)V] \overset{C}{\longmapsto} C[M][n := V] \tag{5}$$

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

$x[n := V] = x$, $U^n[n := V] = V^n$, $U^m[n := V] = U[n := V]^m$ if $n \neq m$,
$(\lambda x.M)[n := V] = \lambda x.M[n := V]$,
$(MN)[n := V] = (M[n := V]N[n := V])$,
$(\mathcal{F}M)[n := V] = (\mathcal{F}M[n := V])$,
$(\sigma X.M)[n := V] = (\sigma X[n := V].M[n := V])$,
$\langle \mathrm{p}, C[\ ]\rangle[n := V] = \langle \mathrm{p}, C[\ ][n := V]\rangle$, $[\ ][n := V] = [\ ]$.

that all subterms labeled n are replaced by V^n. With this new operation, the assignment transition is definable as

$$C[(\sigma U^n.M)V] \longmapsto C[M][n := V].$$

At first glance, a variable lookup merely strips off the label from the labeled value which is now already sitting in the right position. Yet, a closer look reveals that this strategy has a minor flaw: it cannot deal with circular or self-referential assignments. When the label n appears not only in $C[M]$ but also in V, the equivalence-positions in V are not affected by the labeled value substitution. There are two obvious ways out of this dilemma. First, we can blame the assignment statement and require that self-referential assignment builds a circular or infinite term. Second, we can find a more intelligent lookup transition that is knowledgeable about circularities. In other words, the assignment transition only updates the equivalence classes within the program and leaves the equivalence positions within the new value alone. The complimentary lookup transition is then something like a by-need continuation of the latest assignment transition or their equivalence class:

$$C[V^n] \longmapsto C[V[n := V]].$$

This version of the lookup transition simultaneously strips off a label and updates all equivalence positions within the value.

Definition 3.2 contains the formalization of the transition function. With respect to changes in the language $\Lambda_{\mathcal{F}\sigma}$, we leave it at the above informal description. The *eval*-function for the rewriting semantics maps programs directly to values:

$$eval_C(M) = V \text{ iff } M \overset{C}{\longmapsto}{}^{*} V.$$

The correspondence between the CESK- and the C-rewriting semantics is captured in the following

Theorem 1 (C-Simulation). *There is a morphism* Unload *such that for all programs $M \in \Lambda_{\mathcal{F}\sigma}$*

$$eval_C(M) = \mathsf{Unload}(eval_{CESK}(M)).$$

Proof. The proof has four parts, each establishing the correctness of one of the transformations in this section. For each part, we define a principal morphism that models the crucial idea of the corresponding transformation step. Based on this morphism, an induction on the number of evaluation steps shows that two corresponding states in two evaluation sequences are related by this morphism. The four morphisms are:

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

for eliminating environments;

$$\left.\begin{aligned} \mathsf{S}((\mathbf{stop})) &= [\ \ ] \\ \mathsf{S}((\kappa\,\mathbf{arg}\,N)) &= C[[\ \ ]\mathsf{S}(N)] \\ \mathsf{S}((\kappa\,\mathbf{fun}\,F)) &= C[\mathsf{S}(F)[\ \ ]] \end{aligned}\right\} \text{ where } \mathsf{S}(\kappa) = C[\ \ ]$$

for representing p-continuations as contexts;

$$\begin{aligned} \mathsf{T}(\ddagger,(C[\ \ ]\,\mathbf{ret}\ V)) &= C[V] \\ \mathsf{T}(M,C[\ \ ]) &= C[M] \end{aligned}$$

for merging continuations and control strings; and

$$\begin{aligned} \mathsf{U}(x,\theta) &= x, \\ \mathsf{U}(n,\theta) &= \mathsf{U}(\theta(n),\theta[n := (\lambda x.x)])^n, \\ \mathsf{U}(\lambda x.M,\theta) &= \lambda x.\mathsf{U}(M,\theta), \\ \mathsf{U}(MN,\theta) &= \mathsf{U}(M,\theta)\mathsf{U}(N,\theta), \\ \mathsf{U}(\mathcal{F}M,\theta) &= \mathcal{F}\mathsf{U}(M,\theta), \\ \mathsf{U}(\langle \mathrm{p},C[\ \ ]\rangle,\theta) &= \langle \mathrm{p},\mathsf{U}(C[\ \ ],\theta)\rangle, \\ \mathsf{U}([\ \ ],\theta) &= [\ \ ], \\ \mathsf{U}(\sigma X.M,\theta) &= (\sigma\mathsf{U}(X,\theta).\mathsf{U}(M,\theta)) \end{aligned}$$

for folding the store into the control string. The rest of the proof is tedious, but routine. The required morphism Unload is the composition of U, S, and R. □

Theorem 1 stipulates a redefinition of $eval_{CESK}$:

$$eval_{CESK}(M) = \mathsf{Unload}(V,\theta) \text{ iff } \langle M,\emptyset,\emptyset,(\mathbf{stop})\rangle \overset{CESK^+}{\longmapsto} \langle \ddagger,\emptyset,\theta,((\mathbf{stop})\,\mathbf{ret}\ V)\rangle$$

since the user is only interested in the final values and their relationship to the rest of the store. Given this, the claim of the theorem is that $eval_C$ and $eval_{CESK}$ extensionally define the same semantics.

The advantages of the C-rewriting system are obvious. The domain of the transition function is a minor variant of $\Lambda_{\mathcal{F}\sigma}$; the function itself only consists of five rules. Continuations have become something more intuitive, namely, the textual context of an evaluation redex. Labeled values, though complex at first glance, also have a number of advantages. For example, a vacuously abstracted variable causes the CESK-machine to enlarge the domain of the store; in the C-rewriting system the labeled value simply disappears and with it the useless label. Unfortunately, the transition rules are context dependent. It is therefore impossible to freely apply these rewriting rules to subterms. In the next section, we derive a reduction calculus that is equivalent to the C-rewriting system with the desired properties.

4. The λ_v-CS-Calculus

The context dependency of the C-rewriting transition rules has two entirely different aspects. First, the requirement that $C[\ \ ]$ is an applicative context enforces the correct timing of rewriting steps. Since there is exactly one partitioning of a program into a redex and a context, only one action is possible at any given time. Second, the context of such a pair represents the current continuation and can thus become a part of the program in the form of a program point. The latter dependency can apparently be eliminated by encoding a context as a term, but the former is inherent: by their very nature, imperative actions must happen in a certain order.

Because of the dichotomy between timing and context dependency there is no hope of finding a set of entirely context-free reduction relations. The closest we can get is a system where the dependency is confined to a particular point. The root of the term offers itself for this purpose. If there is no context to a redex, the entire extent of the reduction step is known. Therefore, the obvious solution is to distribute the transition work to two sets of relations: ordinary reductions, whose task it is to move a C-redex-like term to the root, and special relations, which perform the imperative action at the root of a program. We refer to these special steps as computation rules or steps and mark them with $\triangleright$ instead of the customary $\rightarrow$ for notions of reduction.

The computation rule for a λ-application $(\lambda x.M)V$ has the same form as the C-transition rule (C2) without context:

$$(\lambda x.M)V \triangleright_{\beta_\sigma} M[x := V^n]$$

where n neither occurs in M nor in V. Given that this rule is applicable when the redex is at the root of the term or, put differently, in the empty context, the inductive definition of applicative contexts requires us to consider two more cases: the embedding of a λ-application to the left of an arbitrary term N and to the right of a value U. In the first case, the C-rewriting semantics says that after the completion of the β-step the modified λ-body $M[x := V^n]$ and N form an application $M[x := V^n]N$. Since the second case is symmetric to the first, we suggest the two λ-reductions:

$$((\lambda x.M)V)N \longrightarrow (\lambda x.MN)V,$$
$$U((\lambda x.M)V) \longrightarrow (\lambda x.UM)V.$$

Like the β-rule itself, these reductions rely on the hygiene convention and assume that the set of free variables in N and U, respectively, is disjoint from the set of bound variables in M.

Although the proposed λ-reductions and computations obviously simulate (C2) in the right way, it is disappointing to realize that the ordinary β_v-reduction

$$(\lambda x.M)V \longrightarrow M[x := V] \qquad (\beta_v)$$

is lost. One would expect that an extension of the Λ-language as a programming language would be orthogonal with respect to the original constructs and that therefore the new calculus would be an extension of the λ_v-calculus. The flaw in our development is the convention that *assignable* and *non-assignable* variables are the same kind of objects. The solution is to split the variable set into two disjoint subsets, one for each category. Accordingly, the β-rewriting rule is split into two different transitions and the set of reductions is extended with the β_v-relation. This decision simultaneously makes our calculus-extension conservative and clarifies why a λ-application for an assignable variable must *bubble up* to the top of a term. While an ordinary β-reduction discards the sharing relation of the λ-abstraction, a $\triangleright_{\beta_\sigma}$-computation retains it for the rest of the computation and hence, if these transitions are in accord with the original semantics, they establish these dynamic sharing relations at the appropriate time.

The simulation of assignment with reductions and computation rules is now straightforward. A σ-application can work its way to the top of a term with rules similar to those for a λ-application. The computation rule becomes

$$(\sigma U^n.M)V \triangleright_\sigma M[n := V].$$

For the delabeling step (C1) this technique does not work since a labeled value V^n does not contain a natural continuation part. Our solution is to replace all assignable variables by a delabel-application of the form $(\mathcal{D}\, x\, M)$ where M is expected to consume the proper value of x after the delabeling transition. A $\mathcal{D}$-application at the top simply delabels the labeled value and applies M to the value:

$$(\mathcal{D}\, V^n M) \rhd_{\mathcal{D}} MV[n := V].$$

If $(\mathcal{D}\, V^n M)$ is to the left of some expression N, the $\mathcal{D}$-application must somehow incorporate the argument N into the consumer expression in order to move closer to the top. According to the intuitive semantics of $\mathcal{D}$-applications, the entire application would first delabel V^n, then apply M to the resulting value, and finally, the result of this application would consume N. Abstracting from the value, this informal description is captured in the λ-abstraction $\lambda v.MvN$ and hence, the two necessary reductions for $\mathcal{D}$-applications are:

$$(\mathcal{D}\, V^n M)N \rightarrow (\mathcal{D}\, V^n(\lambda x.MxN)),$$
$$U(\mathcal{D}\, V^n M) \rightarrow (\mathcal{D}\, V^n(\lambda x.U(Mx))).$$

Finally, we turn to the simulation of $\mathcal{F}$-applications. The crucial idea is that a context $C[\;]$ can be perceived as a function of its hole. Based on this observation, the C-transition rules (C3) and (C4) could be replaced by

$$C[\mathcal{F}M] \longrightarrow M(\lambda x.C[x])$$

without changing the extensional semantics of the C-rewriting system. For a reduction simulation of this rule this means that $\mathcal{F}$-applications must encode their context and apply this to their argument. For the top-level computation rule this is rather simple:

$$\mathcal{F}M \rhd_{\mathcal{F}} M(\lambda x.x)$$

is the required relation. For the other cases, we must again consider the two inductive cases of a context definition. Consider the expression $(\mathcal{F}M)N$ and suppose it is embedded in a context $C[\;]$. Clearly, the function $\lambda f.fN$ is a correct encoding of the immediate context of the $\mathcal{F}$-application. The rest of the context, *i.e.*, $C[\;]$, can be encoded by another $\mathcal{F}$-application and by building this part of the continuation into the expression $\lambda f.fN$. Putting these ideas together, the reductions for $\mathcal{F}$-applications should be

$$(\mathcal{F}M)N \longrightarrow \mathcal{F}\lambda\kappa.M(\lambda f.\kappa(fN)),$$
$$U(\mathcal{F}M) \longrightarrow \mathcal{F}\lambda\kappa.M(\lambda v.\kappa(Uv)).$$

The new language and its reduction and computation relations are summarized in Definition 4.1.[5] Constructing the rest of the calculus is almost standard [2, 6, 7] and we omit it. With respect to the capabilities of the λ_v-CS-calculus as a reasoning system the first important question is whether the calculus is consistent. The answer is expressed in the form of a (generalized) Church-Rosser Theorem:

Theorem 2 (Church-Rosser). *The computation relation, that is, the union of the computation relations and the transitive-reflexive and compatible closure of the reduction relations, satisfies the diamond property.*

Proof. The proof is a generalization of the Martin-Löf proof for the Church-Rosser Theorem for the traditional λ-calculus [2]. □

Equally important, but more relevant for our investigation of evaluation functions is the question whether the system defines an evaluation function that is extensionally equivalent to $eval_{CESK}$. For

[5] The original description of the assignment facet [7] contains the erroneous β_σ-reduction: $(\lambda x.M)V \longrightarrow M[x := V^n]$. As pointed out above, a reduction like this could establish the dynamic sharing relationships at the wrong time. However, in the standard computation function this rule is valid and hence, can be used as a shortcut: see also Section 5 below.

Definition 4.1: The λ_v-CS-calculus

The improper symbols are λ, (,), ., $\mathcal{F}$, σ, and $\mathcal{D}$. $Vars = Var_\lambda \cup Var_\sigma$ is a countable set of variables. The set of labels is an arbitrary, infinite set. The set of values contains variables and abstractions. U and V range over values, n over labels, and X over Var_σ and labeled values. Λ_{CS} contains

- *variables:* x if $x \in Var_\lambda$;
- *λ-abstractions:* $(\lambda x.M)$ if $M \in \Lambda_{CS}$ and $x \in Vars$;
- *applications:* (MN) if $M, N \in \Lambda_{CS}$;
- *$\mathcal{F}$-applications:* $(\mathcal{F}M)$ if $M \in \Lambda_{CS}$;
- *σ-abstractions:* $(\sigma x.M)$ and $(\sigma V^n.M)$ if $M, N \in \Lambda_{CS}$ and $x \in Var_\sigma$;
- *$\mathcal{D}$-applications:* $(\mathcal{D}\, x\, M)$ and $(\mathcal{D}\, V^n\, M)$ if $M \in \Lambda_{CS}$ and $x \in Var_\sigma$.

All variables in the following notions of reductions are in Var_λ unless explicitly stated otherwise:

$$(\lambda x.M)V \xrightarrow{\beta_v} M[x := V] \text{ where } V \text{ is a value,} \qquad (\beta_v)$$

$$U((\lambda x.M)V) \xrightarrow{\beta_R} (\lambda x.(UM))V \text{ where } x \in Var_\sigma,\ U \text{ and } V \text{ are values} \qquad (\beta_R)$$

$$((\lambda x.M)V)N \xrightarrow{\beta_L} (\lambda x.(MN))V \text{ where } x \in Var_\sigma,\ V \text{ is a value} \qquad (\beta_L)$$

$$U(\mathcal{F}M) \xrightarrow{\mathcal{F}_R} \mathcal{F}\lambda\kappa.M(\lambda v.\kappa(Uv)) \text{ where } U \text{ is a value} \qquad (\mathcal{F}_R)$$

$$(\mathcal{F}M)N \xrightarrow{\mathcal{F}_L} \mathcal{F}\lambda\kappa.M(\lambda f.\kappa(fN)) \qquad (\mathcal{F}_L)$$

$$U((\sigma X.M)V) \xrightarrow{\sigma_R} (\sigma X.(UM))V \text{ where } U \text{ and } V \text{ are values} \qquad (\sigma_R)$$

$$((\sigma X.M)V)N \xrightarrow{\sigma_L} (\sigma X.(MN))V \text{ where } V \text{ is a value} \qquad (\sigma_L)$$

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

$$(\mathcal{D}\, X\, M)N \xrightarrow{\mathcal{D}_L} (\mathcal{D}\, X\, (\lambda v.MvN)) \qquad (\mathcal{D}_L)$$

The top-level computation rules are

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

$$(\mathcal{F}M) \triangleright_{\mathcal{F}} M(\lambda x.x) \qquad (\mathcal{F}_T)$$

$$(\sigma U^n.M)V \triangleright_\sigma M[n := V] \text{ where } V \text{ is a value} \qquad (\sigma_T)$$

$$(\mathcal{D}\, V^n\, M) \triangleright_{\mathcal{D}} MV[n := V] \qquad (\mathcal{D}_T)$$

the traditional λ-calculus, this is the standard reduction function which always reduces the leftmost-outermost redex first. With the notion of an applicative context, the definition of a standard reduction function becomes

$$M \longmapsto_{scsv} N \text{ iff there are } P,\ Q, \text{ and } C[\ ] \text{ such that}$$
$$P \text{ reduces to } Q, M \equiv C[P], \text{ and } N \equiv C[Q].$$

Since we have generalized the notion of a calculus to a system that may contain computation relations, we must also provide for these rules in the evaluation function. We call the resulting function standard computation function, denote it by $\overset{\triangleright}{\longmapsto}_{scsv}$, and define it as:

$$\overset{\triangleright}{\longmapsto}_{scsv} = \longmapsto_{scsv} \cup \triangleright_{\beta_\sigma} \cup \triangleright_{\mathcal{F}} \cup \triangleright_\sigma \cup \triangleright_{\mathcal{D}}\ .$$

The evaluation function for programs is simply the transitive-reflexive closure of this standard computation function:

$$eval_{csv}(M) = V \text{ iff } M \overset{\triangleright}{\longmapsto}^{*}_{scsv} V.$$

Its correctness is captured in

Theorem 3 (Standard Simulation). *There is a (n injective) morphism* D *such that for all programs* $M \in \Lambda_{\mathcal{F}\sigma}$,

$$eval_{CESK}(M) = V \textit{ iff } eval_{csv}(\mathsf{D}(M)) = \mathsf{D}(V).$$

Proof. The morphism D is a transformation that replaces all assignable variables and labeled values by $(\mathcal{D}\ X\ \lambda x.x)$. Now, it is easy to show by induction on the structure of the context $C[\ ]$ that

$$C[(\mathcal{D}\ V^n\ M)] \overset{\triangleright}{\longmapsto}^{*}_{scsv} C[MV[n := V]], \text{ and}$$
$$C[\sigma U^n.M)V] \overset{\triangleright}{\longmapsto}^{*}_{scsv} C[M][n := V].$$

A similar result for $C[\mathcal{F}M]$ can also be obtained, but the details are rather technical because the standard computation relation actually produces an encoding of $\lambda x.C[x]$ and not the function itself. We refer the interested reader to our earlier report [6]. Given Theorem 1, these equivalences are sufficient to prove the theorem. □

The crucial consequence of this simulation theorem is that the $eval_{csv}$-function is yet another semantics for $\Lambda_{\mathcal{F}\sigma}$, but unlike the previous evaluation function the standard computation function is *not sequential* in nature. In the next section we compare the two semantic frameworks in more depth and discuss the relevance of this result with respect to implementation.

5. Parallelism for Imperative Languages

The potential for parallel evaluation of $\Lambda_{\mathcal{F}\sigma}$ programs can best be seen from the definition of the function *Eval* in Definition 5.1, which is an (extensionally) equivalent reformulation of the $eval_{csv}$-function. Because of the two-level definition of the calculus, the *Eval*-function needs an auxiliary function *seval*, which returns the value or semi-value of an expression. A semi-value is a top-level redex and causes another cycle of the *Eval*-function; a value is the final result. On one hand, the auxiliary function is actually superfluous. An obvious way to merge *seval* with *Eval* is by introducing a top-level symbol (by a loader/compiler) that *syntactically* marks the root of the term. On the other hand, factoring out this definition facilitates a discussion about a realization of this semantics, because *seval* performs nearly all of the work.

The first interesting point about *seval* is that a functional program is evaluated with almost no loss of efficiency compared to an evaluator for the functional subset of our language. Second and more important, the two parts of an application can still be evaluated in parallel. The intuitive reason behind this parallelism is that imperative effects do not take place immediately, but cause the bubbling-up of the respective semi-value. Thus, if an assignment is to happen, this is indicated by a σ-application in the corresponding branch of the application. Although this may cause a momentary suspension of other evaluations, the coordination is natural and built into the system so that there is no real need for artificial communication.

Unfortunately, this scheme has two major problems. First, the *seval*-function apparently does not require the (semi-) value of the argument part of an application when the function part yields a semi-value. Second, even though the bubbling-up of imperative redexes is a natural means of coordinating imperative effects, it implies that one step of the sequential rewriting machine is replaced by a possibly long sequence of reduction steps. Indeed, the bubbling-up movement may be considered as the counterpart of the von-Neumann bottleneck. The speed of the evaluation function greatly depends on its optimal implementation.

Definition 5.1: The *Eval*-function

Let U and V range over values; S over semi-values, that is, $\mathcal{F}$-, $\mathcal{D}$-, σ-, and λ-applications (whose variables are in Var_σ); M, N, P, and Q over arbitrary terms; and X over labeled values. Then the *Eval*-function is defined as:

$$Eval(M) \equiv \begin{cases} Eval(N[x := V^n]) & \text{if } seval(M) \equiv (\lambda x.N)V \quad (*) \\ Eval(N(\lambda x.x)) & \text{if } seval(M) \equiv (\mathcal{F}N) \\ Eval(N[n := V]) & \text{if } seval(M) \equiv (\sigma U^n.N)V \\ Eval(NV[n := V]) & \text{if } seval(M) \equiv (\mathcal{D}\,V^n N) \\ seval(M) & \text{otherwise.} \end{cases}$$

where *seval* is defined in four inductive clauses:

$$seval(V) \equiv V \text{ and } seval(S) \equiv S;$$

and, if $seval(M) \equiv U$ and $seval(N) \equiv V$, then

$$seval(MN) \equiv \begin{cases} seval(P[x := V]) & \text{if } U \equiv \lambda x.P,\ x \in Var_\lambda \\ UV & \text{otherwise;} \end{cases}$$

and, if $seval(M) \equiv U$ and $seval(N) \equiv S$ then

$$seval(MN) \equiv \begin{cases} (\lambda x.UP)V & \text{if } S \equiv (\lambda x.P)V \quad (*) \\ \mathcal{F}\lambda\kappa.P(\lambda v.\kappa(Uv)) & \text{if } S \equiv \mathcal{F}P \\ (\sigma X.UP)V & \text{if } S \equiv (\sigma X.P)V \\ \mathcal{D}\,X(\lambda v.U(Pv)) & \text{if } S \equiv \mathcal{D}\,XP; \end{cases}$$

and, finally, if $seval(M) \equiv S$, then

$$seval(MN) \equiv \begin{cases} (\lambda x.PN)V & \text{if } S \equiv (\lambda x.P)V \quad (*) \\ \mathcal{F}\lambda\kappa.P(\lambda f.\kappa(fN)) & \text{if } S \equiv \mathcal{F}P \\ (\sigma X.PN)V & \text{if } S \equiv (\sigma X.P)V \\ \mathcal{D}\,X(\lambda v.PvN) & \text{if } S \equiv \mathcal{D}\,XP. \end{cases}$$

At this point, Theorem 3 plays an important role. According to this theorem, the reduction system and the rewriting system are equivalent and hence, some of the insight gained from the rewriting system can be added to the realization of the *seval*-function. With respect to the first point, Theorem 3 clarifies that in most cases the (semi-) value of an argument part is needed independently of the value of the corresponding function part. For example, one consequence of Theorem 3 and the preceding discussion is that the evaluation function need not bubble-up a λ-application with an assignable variable to the root of a program. Once *seval* encounters such an application, it is the correct time to perform a substitution with a labeled value. The only condition is that the label be unique. This condition can be guaranteed in various ways and therefore, the three clauses (*) for bubbling-up a λ-application in Definition 5.1 are superfluous:

$$seval(MN) = seval(P[x := V^n]) \text{ if } seval(M) = \lambda x.P, x \in Var_\sigma,$$
$$\text{and } seval(N) = V.$$

Furthermore, after bubbling up a σ- or $\mathcal{D}$-application the *seval* resumes the evaluation of the applications all the way down to the place where the imperative redex started. Based on these arguments, *seval* may as well evaluate both pieces of an application when encountering it the first time. Only grabbing a continuation in the function part will ever require suspension of evaluation of an argument part. But even then, the stopping point is clearly marked by the arrival of an $\mathcal{F}$-application and, since continuations are rarely thrown away, the early evaluation may still be useful later. In other words, this special case represents the only instance of speculative parallelism in our system.

Finally, Theorem 3 also points to some major shortcuts with respect to the bubbling-up movement. Since σ-applications must resume evaluation of the σ-body after the top-level step and since $\mathcal{D}$- and $\mathcal{F}$-applications build up new β-redexes, the respective reduction sequences can benefit from an immediate evaluation of the internal redexes of their right-hand sides. For example, the ordinary evaluation of $(M(N(\mathcal{D}\ (\lambda x.x)^n\ L)))$ proceeds as follows:

$$\begin{aligned}(M(N(\mathcal{D}\ (\lambda x.x)^n\ L))) &\longrightarrow (M(\mathcal{D}\ (\lambda x.x)^n\ (\lambda x.N(Lx)))\\ &\longrightarrow (\mathcal{D}\ (\lambda x.x)^n\ (\lambda x.M((\lambda x.N(Lx))x))\\ &\longrightarrow (\lambda x.M((\lambda x.N(Lx))x)(\lambda x.x)\\ &\longrightarrow M((\lambda x.N(Lx))(\lambda x.x))\\ &\longrightarrow M(N(L(\lambda x.x)))\end{aligned}$$

but, a parallel reduction of the internal β-redexes shortens this evaluation:

$$\begin{aligned}(M(N(\mathcal{D}\ (\lambda x.x)^n\ L))) &\longrightarrow (M(\mathcal{D}\ (\lambda x.x)^n\ (\lambda x.N(Lx)))\\ &\longrightarrow (\mathcal{D}\ (\lambda x.x)^n\ (\lambda x.M(N(Lx))))\\ &\longrightarrow (\lambda x.M(N(Lx)))(\lambda x.x)\\ &\longrightarrow M(N(L(\lambda x.x))).\end{aligned}$$

The internal reductions must be aware of free variables, yet, since these variables stand for values, they can be treated as such until their "contents" is needed for the function position of a proper application. This has the advantage that the end-consumer of the value can immediately absorb the variable x and the value for x arrives sooner at its final destination, *e.g.*, *seval* can immediately begin with the evaluation of Lx in the above example.

Given the acceptance of *Eval* as a parallel evaluator and the preceding arguments about possible improvements, it seems that the really expensive part of our schema is the substitution algorithm. It imposes a necessary communication among the processors. Put differently, this argument means that the cost of evaluating imperative programs should be comparable to the cost of evaluating applicative programs since for the latter communication costs also constitute the central obstacle to parallel evaluations. In applicative languages this cost may be avoided by compiling programs into combinator form where variables do not exist and substitution is unnecessary. Based on on-going research, we conjecture that a similar compilation scheme can be developed for $\Lambda_{\mathcal{F}\sigma}$ and the λ_v-CS-calculus.

6. Summary

In the preceding sections we demonstrated how an intensionally sequential semantics for an imperative higher-order language can be transformed into an extensionally equivalent, yet intensionally parallel semantics. In Section 5 we discussed a series of improvements to this semantics. Based on this discussion, we believe that if a successful parallelization for functional languages is possible, then the same strategy will also lead to a successful parallelization for imperative programming languages.

Many of the issues raised in Section 5 call for a thorough analysis. In particular, the demand for a combinator-based target system for imperative higher-order languages is a challenging problem. Despite a lack of obvious answers to many questions, we hope that our analysis stimulates research in programming languages for programmers.

Acknowledgement. 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. BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21**(8), 613–641.
2. BARENDREGT, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.
3. CLINGER, W.D., D.P. FRIEDMAN, M. WAND. A scheme for a higher-level semantic algebra, in *Algebraic Methods in Semantics*, J. Reynolds, M. Nivat (Eds.), 1985, 237–250.
4. DENNIS, J.B. Programming generality, parallelism and computer architectures, *Information Processing* **68**, 1969, 484–492.
5. FELLEISEN, M., D.P. FRIEDMAN, B. DUBA, J. MERRILL. Beyond continuations, Technical Report No 216, Indiana University Computer Science Department, 1987.
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. FELLEISEN, M., D.P. FRIEDMAN. A calculus for assignments in higher-order languages, *Proc. 14th ACM Symp. Principles of Programming Languages*, 1987, 314-325.
8. FELLEISEN, M. Reflections on Landin's J-operator: a partly historical note, *Computer Languages*, 1987, to appear.
9. FRIEDMAN, D.P., C.T. HAYNES, E. KOHLBECKER. Programming with continuations, in *Program Transformations and Programming Environments*, P. Pepper (Ed.), Springer-Verlag, 1985, 263–274.
10. HAYNES, C.T., D.P. FRIEDMAN, M. WAND. Obtaining coroutines from continuations, *Computer Languages* **11**(3/4), 1986, 143–153.
11. HAYNES, C. T. Logic continuations, *Proc. Third International Conf. Logic Programming*, London, Springer-Verlag, 1986, 671–685; also to appear in *Journal of Logic Programming*, 1987.
12. HENSON, M.C., R. TURNER. Completion semantics and interpreter generation, *Proc. 9th ACM Symp. Principles of Programming Languages*, 1982, 242–254.
13. KOHLBECKER, E, M. WAND. Macro-by-example: deriving syntactic transformations from their specifications, *Proc. 14th ACM Symp. Principles of Programming Languages*, 1987, 77–85.
14. LANDIN, P.J. The next 700 programming languages, *Comm. ACM* 9(3), 1966, 157–166.
15. LANDIN, P.J. An abstract machine for designers of computing languages, *Proc. IFIP Congress*, 1965, 438–439.
16. LANDIN, P.J. The mechanical evaluation of expressions, *Computer Journal* **6**(4), 1964, 308–320.
17. MAGÓ, G.A. A network of microprocessors to execute reduction languages, *Int. Journal of Computer and Information Sciences* **8**, 1979, 349–385; 435–471.
18. MASON, I. A. *The Semantics of Destructive Lisp*, Ph.D. dissertation, Stanford University, 1986.
19. PLOTKIN, G.D. A structural approach to operational semantics, Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
20. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science* **1**, 1975, 125–159.
21. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages, *Proc. ACM Annual Conference*, 1972, 717–740.

22. Reynolds, J.C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept, *Comm. ACM* **13**(5), 1970, 308–319.

23. Sussman G.J., G. Steele. Scheme: An interpreter for extended lambda calculus, Memo 349, MIT AI-Lab, 1975.

24. Talcott, C. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*, Ph.D. dissertation, Stanford University, 1985.

25. Wand, M., D.P. Friedman. Compiling lambda-expressions using continuations and factorizations, *Computer Languages* **3**, 1978, 241–263.

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

259

PARLE Parallel Architectures and Languages Europe

Volume II: Parallel Languages
Eindhoven, The Netherlands, June 15–19, 1987
Proceedings

Edited by
J. W. de Bakker, A. J. Nijman and P. C. Treleaven