Control Operators, the SECD-Machine, and the λ-Calculus

by

Matthias Felleisen and Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 197

Control Operators, the SECD-Machine, and the λ-Calculus

by

Matthias Felleisen and Daniel P. Friedman
Indiana University
June, 1986

This material is based on work supported by the National Science Foundation under grant numbers DCR 85-01277 and DCR 85-03279.

To appear in *The Proceedings of the Conference on Formal Description of Programming Concepts Part III* in Ebberup, Denmark, August, 1986.

Control Operators, the SECD-Machine, and the λ-Calculus

Matthias Felleisen, Daniel P. Friedman

Indiana University, Lindley Hall 101, Bloomington, IN 47405, USA

Abstract

Control operators like J and *call/cc* are often found in implementations of the λ-calculus as a programming language. Their semantics is always defined by the evaluation function of an abstract machine. We show that, given such a machine semantics, one can derive an algebraic extension of the λ_v-calculus. The extended calculus satisfies the diamond property and contains a Church-Rosser subcalculus. This underscores that the interpretation of control operators is to a certain degree independent of a specific order of evaluation.

1. The control problem of the λ-calculus

The λ-calculus is a natural basis for a programming language. Recognition of this fact led Landin to use it as a meta-language to study programming language features [9, 10]. Since the purely functional basis was insufficient to transliterate jumps and labels directly, Landin extended the language with the non-functional control operator **J**[9]. Others who used similar languages for ordinary applications followed and also added control operators. Notably all the Scheme dialects include control facilities which are equal in power to **J**, *e.g.* *catch* and *throw* [17] and *call-with-current-continuation* (abbreviated *call/cc*) [13]. Common Lisp [16] is equipped with a less powerful version of *catch* and *throw*. Label values in GEDANKEN [14] and escape functions [15] are related to **J**, but more traditional in nature.

The general advantage of non-functional control operators is that they "provide a way of pruning unnecessary computation and allow certain computations to be expressed by more compact and conceptually manageable programs." [18, p.16] They enhance the expressive power of a language and give the programmer a conceptual handle for the specification of control. Some examples will support our case.

Suppose a function Σ_0^* is required which maps a tree of numbers to the sum of the numbers if there is no 0 in the tree and otherwise returns a 0. A tree is either an empty tree or it is a node which contains a number and two (sub-) trees. If the specification did not include the 0-exception clause, the function could be written in the ordinary, recursive style. In the λ-value-calculus it would be expressed by:

$$\Sigma^* \equiv \mathbf{Y}_v(\lambda s.\lambda t.$$
$$(\mathbf{if}\,(\mathbf{mt?}\,t)\,\ulcorner 0\urcorner$$
$$(+(\mathbf{num}\,t)(+(s(\mathbf{lson}\,t))(s(\mathbf{rson}\,t))))))$$

This material is partly based on work supported by the National Science Foundation under grants DCR 85-01277 and DCR 85-03279.

where $\mathbf{Y}_v$ is the call-by-value recursion operator, **if** is syntactic sugar for a branching construct, and **mt?**, **num**, **lson**, and **rson** are combinators to deal with number trees.

Given continuation-operators like **J** and *call/cc*, we can solve the problem by modifying the recursive function. For example, with *call/cc* we could write:

$$
\begin{aligned}
\Sigma_0^* \equiv \lambda t.\, &call/cc(\lambda\kappa.\\
&\mathbf{Y}_v(\lambda s.\lambda t.\\
&\ (\mathbf{if}\,(\mathbf{mt?}\,t)\,\ulcorner 0\urcorner\\
&\quad (\mathbf{if}\,(\mathbf{zero?}\,(\mathbf{num}\,t))\,(\kappa\ulcorner 0\urcorner)\\
&\quad\quad (+(\mathbf{num}\,t)(+(s(\mathbf{lson}\,t))(s(\mathbf{rson}\,t)))))))\\
&t).
\end{aligned}
$$

With **J** the definition becomes $\lambda t.(\lambda k.\ \ldots)(\mathbf{J}\lambda x.x)$ [15]. When Σ_0^* is applied to an argument, the operator *call/cc* applies its argument to a representation of the current continuation, *i.e.*, κ becomes bound to a function-like abstraction of the rest of the program. Then the recursive tree traversal begins. If there is a 0 in the tree, Σ_0^* sooner or later visits that node and can then invoke the continuation on 0. The program evaluation continues as if Σ_0^* had returned a value. The advantage of this approach is clear: the function differs from the purely recursive summation function only to the extent to which the two specifications differ. The modification is simple and, we believe, easy to understand. The function definition is in no way obscured by auxiliary expressions. We classify this kind of continuation as an *escape* continuation.

When continuations can be passed around freely as first-class objects, like in Scheme and ISWIM, they may be used to implement more interesting control strategies. A classical example is the coroutine facility. There are numerous situations when a program is best expressed as two or more coroutines which simultaneously analyze input and synthesize output files [7]. If a programming language does not have a coroutine mecahnism but has continuation operators, this behavior can easily be achieved by passing around the current control state in the form of a continuation [18]. The advantage of continuations is even more apparent when bi-directional communication among the coroutines is required [5].

First-class continuations are also useful when multi-valued functions or relations are needed in a functional language. A (functional representation of a) relation generally produces more than one result for a given argument. To this end, it can return a result with a continuation. If more results are required, the calling function can reinvoke the continuation. With this technique it becomes possible to implement a Landin-style embedding of logic languages [2, 6]. Intelligent backtrack strategies are realized in a similar manner by retaining a store of appropriate continuations [4]. There are many other examples of advantageous uses of continuation operators, but an introduction to the techniques of programming with continuations is not our concern here.

Although this is well-recognized and control operators are heavily used in practical programming, non-functional operators are regarded with skepticism among language theoreticians. The imperative character of these operations is suspicious. Programs using them are difficult to prove correct. Whereas function application is modeled by the β-rule, which allows for algebraic manipulations of programs, there is nothing in the λ-calculus for reasoning about non-functional control.

This difficulty was overcome when we showed in a companion paper [3] that the λ_v-calculus [12] can incorporate axioms for general control operators. The development of this extension was rather *ad hoc*. The new axioms were based on our intuition about programming with continuations. The formal, operational semantics of the control operators had no direct influence on our work. Consequently, it was rather hard to connect the two systems. It finally turned out that the standard reduction semantics disagreed with the machine semantics. If the result contains continuations, the respective continuations can differ by huge pieces of dead code, *i.e.*, subterms which never play a role in an evaluation.

In this paper we show that there is a (more) systematic way to derive a calculus for a given machine. The approach generalizes Plotkin's way of deriving the λ_v-calculus from the SECD-machine. The main idea is to gradually eliminate components of the machine by merging them into the program component. The outcome is a program rewriting system. This in turn can be taken as the specification of a standard reduction function. The transition from it to a reduction system is easy.

In the next section we define our extended programming language and its meaning. For the latter part we revise Landin's SECD-machine. The third section contains a derivation of a program-oriented rewriting system from the original machine. The reduction system is presented in Section 4. For this we assume some general knowledge of the notation and terminology of the classical λ-calculus [1, Ch. 2, 3]. Our calculus is an extension of Plotkin's λ_v-calculus since our machine defines a call-by-value semantics for applications. It differs slightly from the one presented in our companion paper but we can still prove variations of the Church-Rosser Theorem and the Curry-Feys Standardization Theorem. In the fifth section we demonstrate in what sense the calculus corresponds to the machine. The standard reduction function does not simulate the machine evaluation function, but the difference is irrelevant. For the rest of the section we follow Plotkin's programme for the investigation of the λ_v-calculus. The last section is a brief discussion of our results.

2. The extended programming language

The traditional term set of the pure λ-calculus is the basis of our programming language. It includes two new types of applications: $\mathcal{C}$- and $\mathcal{A}$-applications. For the sake of simplicity we omit constant and function symbols, but they could easily be added. The formal syntax specification is shown in Definition 1. In what follows we liberally omit parentheses wherever it is unambiguous.

The notion of free and bound variables in a term M, $FV(M)$ and $BV(M)$, respectively, carries over directly from the *pure* λ-calculus under the provision that $\mathcal{C}$ and $\mathcal{A}$ are symbols which are neither free nor bound. 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 terms that are equal except for 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*. Furthermore, we extend Barendregt's definition of the substitution function, $M[x := N]$, to Λ_c in the natural way: $\mathcal{C}$- and $\mathcal{A}$-applications are treated like applications where the function part is simply ignored.

The intuitive meaning of the traditional constructs is approximately the same. A variable represents a value. An abstraction roughly corresponds to a function. An $\mathcal{A}$-application

Definition 1: The term sets Λ_c and Λ

The improper symbols are λ, (,), ., $\mathcal{C}$, and $\mathcal{A}$. *Var* is a countable set of variables. The symbols $x, \kappa, f, v, \ldots$, range over *Var* as meta-variables but are also used as if they were elements of *Var*. The *term set* Λ_c contains

- *variables:* x if $x \in Var$;
- *abstractions:* $(\lambda x.M)$ if $M \in \Lambda_c$ and $x \in Var$;
- *applications:* (MN) if $M, N \in \Lambda_c$; M is called the function, N is called the argument;
- $\mathcal{C}$*-applications:* $(\mathcal{C}M)$ if $M \in \Lambda_c$; M is called the $\mathcal{C}$-argument;
- $\mathcal{A}$*-applications:* $(\mathcal{A}M)$ if $M \in \Lambda_c$; M is called the $\mathcal{A}$-argument.

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_c restricted to variables, applications, and abstractions.

aborts the current computation and begins a new one with its argument as the starting point. A $\mathcal{C}$-application captures the current continuation of the program and passes it as a function-like abstraction to its argument. An application invokes a function or a continuation on an argument.

For historical reasons we use an abstract machine to formally define the semantics of Λ_c-programs. The machine is derived from Reynolds' extended interpreter IV [15]. It is similar to Landin's SECD-machine [8] but closer to denotational semantics and, hopefully, easier to understand. The machine works on states which are triples composed of a control string, an environment, and a continuation code; it is accordingly referred to as the CEK-machine.

A *control string* is either the symbol "$\natural$" or a Λ_c-expression. *Environments* are finite maps from the set of variables *Var* to the set of *semantic values*, that is, the union of closures and continuation points. If ρ is an environment, then $\rho[x := V]$ is the environment which is like ρ except for the point x where it is V. A *closure* is an ordered pair composed of an abstraction and an environment whose domain contains the free variables of the abstraction. A closure (M, ρ) is called *continuation-free* iff for all free x in M, $\rho(x)$ is a continuation-free closure. *Continuation points* are tagged structures of the form $(\mathbf{p}, \kappa)$ where κ is a continuation code.

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_c-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:

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

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.

Table 1: The CEK-transition function

$$
\begin{array}{ll}
(1) & \langle x, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \updownarrow, \mathbb{0}, (\kappa\ \mathbf{ret}\ \rho(x))\rangle \\
(2) & \langle \lambda x.M, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \updownarrow, \mathbb{0}, (\kappa\ \mathbf{ret}\ \langle \lambda x.M, \rho\rangle)\rangle \\
(3) & \langle MN, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle M, \rho, (\kappa\ \mathbf{arg}\ N\ \rho)\rangle \\
(4) & \langle \updownarrow, \mathbb{0}, ((\kappa\ \mathbf{arg}\ N\ \rho)\ \mathbf{ret}\ F)\rangle \overset{CEK}{\longmapsto} \langle N, \rho, (\kappa\ \mathbf{fun}\ F)\rangle \\
(5) & \langle \updownarrow, \mathbb{0}, ((\kappa\ \mathbf{fun}\ \langle \lambda x.M, \rho\rangle)\ \mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle M, \rho[x := V], \kappa\rangle \\
(6) & \langle \mathcal{C}M, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle M, \rho, (\kappa\ \mathbf{cont})\rangle \\
(7) & \langle \updownarrow, \mathbb{0}, ((\kappa\ \mathbf{cont})\ \mathbf{ret}\ \langle \lambda x.M, \rho\rangle)\rangle \overset{CEK}{\longmapsto} \langle M, \rho[x := \langle \mathrm{p}, \kappa\rangle], (\mathbf{stop})\rangle \\
(8) & \langle \updownarrow, \mathbb{0}, ((\kappa\ \mathbf{cont})\ \mathbf{ret}\ \langle \mathrm{p}, \kappa_0\rangle)\rangle \overset{CEK}{\longmapsto} \langle \updownarrow, \mathbb{0}, (\kappa_0\ \mathbf{ret}\ \langle \mathrm{p}, \kappa\rangle)\rangle \\
(9) & \langle \updownarrow, \mathbb{0}, ((\kappa\ \mathbf{fun}\ \langle \mathrm{p}, \kappa_0\rangle)\ \mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle \updownarrow, \mathbb{0}, (\kappa_0\ \mathbf{ret}\ V)\rangle \\
(10) & \langle \mathcal{A}M, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle M, \rho, (\mathbf{stop})\rangle
\end{array}
$$

A CEK-*machine state* is either a triple of the form $\langle \updownarrow, \mathbb{0}, \kappa\rangle$ where κ is a **ret**-continuation or a triple of the form $\langle M, \rho, \kappa\rangle$ where M is a Λ_c-term, ρ is an environment with $FV(M) \subseteq Dom(\rho)$, and κ is a p-continuation. Machine states of the form $\langle M, \mathbb{0}, (\mathbf{stop})\rangle$ are the *initial* states; for all semantic values V, $\langle \updownarrow, \mathbb{0}, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle$ is a *terminal* state.

The state transition function is displayed in Table 1. We use $\overset{CEK^+}{\longmapsto}$, $\overset{CEK^*}{\longmapsto}$, and $\overset{CEK}{\longmapsto}_=$ to denote the transitive, transitive-reflexive, and reflexive closure, respectively. The rules (CEK1) through (CEK5) correspond to an evaluator for the λ-value-calculus, *e.g.*, the classical SECD-machine. The rules (CEK6) to (CEK8) define the operations of a $\mathcal{C}$-application: the current continuation is marked, the $\mathcal{C}$-argument is evaluated, and, eventually, the continuation point is passed to the evaluated argument. The last step in this sequence also replaces the current continuation by the initial one. This is where *call/cc* and $\mathcal{C}$ differ: *call/cc* is equivalent to $\lambda f.\mathcal{C}\lambda\kappa.\kappa(f\kappa)$. Rule (CEK9) shows that the invocation of a continuation removes the current continuation, and moves the former one in its place. The invocation argument is placed on the stack. According to rule (CEK10), an $\mathcal{A}$-application ignores the continuation and starts the evaluation of the $\mathcal{A}$-argument.

In order to evaluate a program M, the machine is started in the initial state $\langle M, \mathbb{0}, (\mathbf{stop})\rangle$. Then the machine moves into the next legal state according to the transition function. This is repeated until a terminal state is reached. When the machine reaches a terminal state, it stops and returns the value on the stack as the answer. We summarize the evaluation process in the following:

$$eval_{CEK}(M) = V \text{ iff } \langle M, \mathbb{0}, (\mathbf{stop})\rangle \overset{CEK^+}{\longmapsto} \langle \updownarrow, \mathbb{0}, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle.$$

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

The evaluation function returns semantic values. A continuation-free closure can be mapped to a Λ_c-term by substituting all free variables by the terms which correspond to

their environment values. A continuation point does not have an obvious association. We accept this and do not define an unload function for the CEK-machine until later.

Convention. M, N, P, Q are variables ranging over terms in control string position. U, V, and F stand for values. The letters s, c, ρ, and κ denote machine states, control strings, environments, and continuation codes, respectively. We use similar conventions throughout the sequel (*proviso quod* the appropriate changes). **End of Convention**

Given the CEK-machine, one can theoretically reason about programs with non-functional control [18], but it is rather awkward. What a programmer really wants is a rewriting system that allows him to think about programs in terms of program code and related notions. In the next section, we show how to derive such a term rewriting system from the machine definition.

3. From the CEK-machine to a term rewriting system

The CEK-machine uses environments and continuations in addition to terms for the evaluation of programs. Hence, if we want to have a pure term rewriting system, we need to eliminate environments and continuations. We perform this transformation in three steps. The first step results in a machine with two state components: control strings and continuation codes. The second step is an intermediary which introduces a less machine-oriented encoding for continuations. The last one incorporates the two remaining components into a single one.

3.1 The CK-machine

Environments in the CEK-machine are merely a functional representation of substitutions. It is quite natural to replace environments by substitutions which are performed at the appropriate place. The resulting machine is called the CK-machine.

The CK-machine has control string-continuation pairs as states. The definition of control terms needs to be adjusted. They now include continuation points at the base case. Continuation codes contain control strings where they formerly had closures or term-environment pairs, *e.g.* $(\kappa\,\mathbf{arg}\ M\rho)$ becomes $(\kappa\,\mathbf{arg}\ N)$ for some control string N. We leave it at these informal revisions and define the CK-transition function in Table 2.

The substitution function is extended in the obvious way to work on control strings and semantic values. All other definitions of the CEK-machine are applied to the CK-machine *mutatis mutandis*.

For the transition from CEK-states to CK-states we adopt Plotkin's function *Real* to work on the entire control string domain:

$$\begin{aligned}
\mathcal{R}(\langle M, \rho\rangle) &\equiv M[x_1 := \mathcal{R}(\rho(x_1))]\ldots[x_n := \mathcal{R}(\rho(x_n))] \\
&\qquad \text{where } FV(M) = \{x_1, \ldots, x_n\} \\
\mathcal{R}(\langle \mathbf{p}, \kappa\rangle) &= \langle \mathbf{p}, \mathcal{K}(\kappa)\rangle \\
\mathcal{R}(\langle \lightning, \rho\rangle) &= \lightning.
\end{aligned}$$

Table 2: The CK-transition function

(1)	$\langle(\mathsf{p}, \kappa_0), \kappa\rangle \overset{CK}{\longmapsto} \langle \updownarrow, (\kappa\ \mathbf{ret}\ (\mathsf{p}, \kappa_0))\rangle$
(2)	$\langle \lambda x.M, \kappa\rangle \overset{CK}{\longmapsto} \langle \updownarrow, (\kappa\ \mathbf{ret}\ \lambda x.M)\rangle$
(3)	$\langle MN, \kappa\rangle \overset{CK}{\longmapsto} \langle M, (\kappa\ \mathbf{arg}\ N)\rangle$
(4)	$\langle \updownarrow, ((\kappa\ \mathbf{arg}\ N)\ \mathbf{ret}\ F)\rangle \overset{CK}{\longmapsto} \langle N, (\kappa\ \mathbf{fun}\ F)\rangle$
(5)	$\langle \updownarrow, ((\kappa\ \mathbf{fun}\ \lambda x.M)\ \mathbf{ret}\ V)\rangle \overset{CK}{\longmapsto} \langle M[x := V], \kappa\rangle$
(6)	$\langle \mathcal{C}M, \kappa\rangle \overset{CK}{\longmapsto} \langle M, (\kappa\ \mathbf{cont})\rangle$
(7)	$\langle \updownarrow, ((\kappa\ \mathbf{cont})\ \mathbf{ret}\ \lambda x.M)\rangle \overset{CK}{\longmapsto} \langle M[x := (\mathsf{p}, \kappa)], (\mathbf{stop})\rangle$
(8)	$\langle \updownarrow, ((\kappa\ \mathbf{cont})\ \mathbf{ret}\ (\mathsf{p}, \kappa_0))\rangle \overset{CK}{\longmapsto} \langle \updownarrow, (\kappa_0\ \mathbf{ret}\ (\mathsf{p}, \kappa))\rangle$
(9)	$\langle \updownarrow, ((\kappa\ \mathbf{fun}\ (\mathsf{p}, \kappa_0))\ \mathbf{ret}\ V)\rangle \overset{CK}{\longmapsto} \langle \updownarrow, (\kappa_0\ \mathbf{ret}\ V)\rangle$
(10)	$\langle \mathcal{A}M, \kappa\rangle \overset{CK}{\longmapsto} \langle M, (\mathbf{stop})\rangle.$

The auxiliary function $\mathcal{K}$ maps CEK-continuation codes to CK-continuation codes:

$$\begin{aligned}
\mathcal{K}((\mathbf{stop})) &= (\mathbf{stop}) \\
\mathcal{K}((\kappa\ \mathbf{cont})) &= (\mathcal{K}(\kappa)\ \mathbf{cont}) \\
\mathcal{K}((\kappa\ \mathbf{arg}\ N\rho)) &= (\mathcal{K}(\kappa)\ \mathbf{arg}\ \mathcal{R}(\langle N, \rho\rangle)) \\
\mathcal{K}((\kappa\ \mathbf{fun}\ F)) &= (\mathcal{K}(\kappa)\ \mathbf{fun}\ \mathcal{R}(F)) \\
\mathcal{K}((\kappa\ \mathbf{ret}\ V)) &= (\mathcal{K}(\kappa)\ \mathbf{ret}\ \mathcal{R}(V)).
\end{aligned}$$

With these functions we can now express in what sense the CEK-machine is equivalent to the CK-machine:

Theorem 3.1 (CK-simulation). *For any program M, $\mathcal{R}(eval_{CEK}(M)) = eval_{CK}(M)$.*

Proof. The clauses of the two transition functions obviously correspond to each other. The CK-machine has no variables in control string position, but will always contain a value in the respective place. Thus, rule (CK1) provides the means to return a continuation point as did (CEK1) in the CEK-machine. More formally, one can show that

$$\langle c_1, \rho_1, \kappa_1\rangle \overset{CEK}{\longmapsto} \langle c_2, \rho_2, \kappa_2\rangle$$

implies

$$\langle \mathcal{R}(\langle c_1, \rho_1\rangle), \mathcal{K}(\kappa_1)\rangle \overset{CK}{\longmapsto} \langle \mathcal{R}(\langle c_2, \rho_2\rangle), \mathcal{K}(\kappa_2)\rangle.$$

This in turn says that, if the CEK-machine halts in $\langle \updownarrow, \emptyset, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle$, the CK-machine reaches the state $\langle \updownarrow, ((\mathbf{stop})\ \mathbf{ret}\ \mathcal{R}(V))\rangle$. On the other hand, if the CEK-machine loops forever on a program M, then so does the CK-machine. There are no other cases and this concludes the proof. □

This first transition leaves us with a machine that works with control strings and continuation codes. The expected next step would be to incorporate continuation codes into

the term components. We have found, however, that the notion of continuation codes is too machine-oriented for a smooth transition. The next step is a replacement of continuation codes by a more familiar concept. The final step is then quite natural.

3.2 The CC-machine

An inspection of some sample evaluations on the CK-machine reveals that the machine repeats a two-phase procedure. In the first phase the control string is searched for either an application of the form (FV), a $\mathcal{C}$-application, or an $\mathcal{A}$-application. When one of these subterms is found, the machine performs a computation step proper. This may either be a substitution, the labeling of a continuation, or the throwing away of a continuation. Then the machine re-enters the search phase.

During the search phase the machine unravels the control string and shifts term components to the continuation part of the state. These program parts are saved for later use, *i.e.*, the continuation code memorizes the textual context of the next "good" application. The term **(stop)** in continuation position simply means that nothing has to be remembered; the machine is about to begin an evaluation and when this continuation is ever looked at again, the evaluation stops. A continuation like $(\kappa\,\mathbf{arg}\ N)$ recalls that the machine had found an application with argument part N, that the context of this application was encoded in κ, and that the machine is currently evaluating the function part. Conversely, $(\kappa\,\mathbf{fun}\ F)$ indicates that the function part is evaluated and that the argument part is directing the evaluation. A continuation of the type $(\kappa\,\mathbf{cont})$ originates from a $\mathcal{C}$-application. The continuation code was marked and the $\mathcal{C}$-argument determines the further course of the evaluation. The machine acts as if it had encountered an application at the root of a term. The argument is the current continuation; the $\mathcal{C}$-argument stands for the function part. **ret**-continuations finally express that a value has been found and that the machine has to inspect the continuation code—or, control memory—to find out what to do next.

With this description we are now in a position to design a term-like representation of continuation codes. The notion of a textual context is captured in the concept of a term context. For our special case we need contexts with one hole and, furthermore, the path from the root to the hole may only lead through applications. However, the machine not only needs to know the context, but it also needs to remember which part of an application it has already seen. To this end we introduce labeled applications and labeled sk-contexts. If M and N are Λ_c-terms, then $M \bullet N$ is a *labeled application* of M to N; MN is the corresponding unlabeled application. *Labeled sk-contexts* are defined inductively as follows:

(skC1) $[\ ]$ is a labeled sk-context,

(skC2) $C[\ ]P$ is a labeled sk-context if $C[\ ]$ is a labeled sk-context and P is a Λ_c-term,

(skC3) $P \bullet C[\ ]$ is a labeled sk-context if $C[\ ]$ is a labeled sk-context and P is a value.

Unlabeled sk-contexts are defined in the same way except that unlabeled applications are used in the third clause. If $C[\ ]$ is an sk-context, then $C[M]$ is the term where the hole is filled with the term M; $C[C'[\ ]]$ is the sk-context where the hole is filled with the sk-context $C'[\ ]$. The important connection between contexts and control strings $M \in \Lambda_c$ is captured in:

Lemma (Unique context). *For all control strings M there is a unique (labeled or unlabeled) sk-context $C[\ ]$ such that, if M is not a value, then $M \equiv C[FV]$ or $M \equiv C[\mathcal{C}N]$ or*

$M \equiv C[\mathcal{A}N]$ *(or* $M \equiv C[F \bullet V]$*, for labeled sk-contexts).*

Proof. A straightforward induction on the structure of M. □

The above description of the function of particular control codes leads to the following definition for a morphism $\mathcal{C}$ from continuation codes to sk-contexts:

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

$\mathcal{S}$ replaces codes in continuation points by contexts:

$$\mathcal{S}((\mathbf{p}, \kappa)) = (\mathbf{p}, \mathcal{C}(\kappa)),\ \mathcal{S}(\lightning) = \lightning,\ \mathcal{S}(x) = x,$$
$$\mathcal{S}(MN) = \mathcal{S}(M)\mathcal{S}(N),\ \mathcal{S}(\lambda x.M) = \lambda x.\mathcal{S}(M),$$
$$\mathcal{S}(\mathcal{C}M) = \mathcal{C}\mathcal{S}(M),\ \mathcal{S}(\mathcal{A}M) = \mathcal{A}\mathcal{S}(M).$$

The new CC-machine works on states which combine control strings and labeled sk-contexts. Control strings contain sk-contexts where they used to contain continuation codes. The initial state of the machine is $(M, [\ ])$; the machine stops when it reaches the state $(\lightning, V)$ for some value V. The CC-transition function is shown in Table 3. All other notions, in particular the one for the eval-function, are adapted in the appropriate way.

Table 3: The CC-transition function

(1)	$(\mathbf{p}, C_0[\ ]), C[\ ]) \xmapsto{CC} (\lightning, C[(\mathbf{p}, C_0[\ ])])$
(2)	$(\lambda x.M, C[\ ]) \xmapsto{CC} (\lightning, C[\lambda x.M])$
(3)	$(MN, C[\ ]) \xmapsto{CC} (M, C[[\ ]N])$
(4)	$(\lightning, C[V\,N]) \xmapsto{CC} (N, C[V \bullet [\ ]])$
(5)	$(\lightning, C[(\lambda x.M) \bullet V]) \xmapsto{CC} (M[x := V], C[\ ])$
(6)	$(\mathcal{C}M, C[\ ]) \xmapsto{CC} (M, [\ ] \bullet (\mathbf{p}, C[\ ]))$
(7)	$(\lightning, (\lambda x.M) \bullet (\mathbf{p}, C[\ ])) \xmapsto{CC} (M[x := (\mathbf{p}, C[\ ])], [\ ])$
(8)	$(\lightning, (\mathbf{p}, C_0[\ ]) \bullet (\mathbf{p}, C[\ ])) \xmapsto{CC} (\lightning, C_0[(\mathbf{p}, C[\ ])])$
(9)	$(\lightning, C[(\mathbf{p}, C_0[\ ]) \bullet V]) \xmapsto{CC} (\lightning, C_0[V])$
(10)	$(\mathcal{A}M, C[\ ]) \xmapsto{CC} (M, [\ ])$.

From the definition of the CC-transition function it follows that labeled applications are only needed to make the CC-transition relation into a proper function. Even though there is only one unique context surrounding the "good" subterm, the machine could still take

two different paths in case $M \equiv C[(\lambda x.P)V]$, *i.e.*, in the absence of labels it may either use (CC4) or (CC5). A labeled application means the machine has evaluated the function *and* the argument part. It is then safe to perform a computation step. Furthermore, this device makes the CC-machine reflect the CK-machine on a rule-by-rule basis:

Theorem 3.2 (CC-simulation). *For any program M,* $S(eval_{CK}(M)) = eval_{CC}(M)$.

Proof. Again, we can show that every CK-step is reflected by a CC-transition move, *i.e.*

$$\langle c_1, \kappa_1 \rangle \overset{CK}{\longmapsto} \langle c_2, \kappa_2 \rangle \text{ implies } \langle S(c_1), \mathcal{C}(\kappa_1) \rangle \overset{CC}{\longmapsto} \langle S(c_2), \mathcal{C}(\kappa_2) \rangle.$$

Hence, if the CK-machine returns a value V, then the CC-machine returns the value $S(V)$. Otherwise, both machines loop forever. □

Before we end this subsection we note that not all of the rules are necessary. The first two transitions may be merged into a single rule:

$$\langle V, C[\ ] \rangle \overset{CC}{\longmapsto} \langle \updownarrow, C[V] \rangle.$$

Rule (CC5) subsumes (CC7) and rule (CC9) subsumes (CC8). Furthermore, the first four rules are simply bookkeeping rules which allow the machine to remember which part of the term it has visited. For a pure rewriting system they may be eliminated.

3.3 The C-rewriting system

The key idea for the third and last transition step is simple. It eliminates the bookkeeping machinery of the CC-machine and directly relates control strings to each other. Every state in the CC-machine already corresponds to a control string. If all labels are removed from control strings and contexts, a state of the form $\langle \updownarrow, M \rangle$ stands for M and a state like $\langle M, C[\ ] \rangle$ represents $C[M]$. This translation does not map continuation points to terms; the result is like a Λ_c-term possibly containing contexts. The precise definition of the term set Λ_p is given in Definition 2. Given a function $\mathcal{J}$ which removes all the labels from applications including those that occur within continuation contexts, the morphism $|\cdot|$ from CC-states to Λ_p-terms is formalized by:

$$|\langle \updownarrow, M \rangle| = \mathcal{J}(M)$$
$$|\langle M, C[\ ] \rangle| = \mathcal{J}(C[M]).$$

As this correspondence is not one-to-one, the CC-transition function only induces a relation. This is easily remedied by throwing out all the rules that just keep track of what the machine has already seen. Together with the simplifications at the end of the previous subsection we get the control string rewriting function as shown in Table 4. The "Unique context"-Lemma assures that this is indeed a well-defined function.

The transition function induces the usual evaluation function:

$$eval_C(M) = N \text{ iff } M \overset{C}{\longmapsto}{}^{*} N \text{ such that } N \text{ is a value.}$$

With this eval-function we can state our final simulation theorem.

Definition 2: The term set Λ_p

The *term set* Λ_p and the set of sk-contexts are defined by mutual induction. Given $x \in Var$ and a Λ_p-sk-context $C[\ ]$,

$$x \text{ and } (\mathbf{p}, C[\ ]) \text{ are in } \Lambda_p.$$

If M and N are in Λ_p, then, for any x,

$$\lambda x.M, MN, \mathcal{C}M, \text{ and } \mathcal{A}M \text{ are in } \Lambda_p.$$

Variables, continuation points, and abstractions are referred to as values. The set of Λ_p-*sk-contexts* contains

$$[\ ], \text{ and,}$$

if $C[\ ]$ is a Λ_p sk-context, P is a Λ_p-term, and Q is a Λ_p-value, then

$$C[\ ]P \text{ and } QC[\ ] \text{ are } \Lambda_p\text{-sk-contexts.}$$

Table 4: The C-transition function

$$\begin{array}{ll}
(1) & C[(\lambda x.M)V] \overset{C}{\longmapsto} C[M[x := V]] \\
(2) & C[\mathcal{C}M] \overset{C}{\longmapsto} M(\mathbf{p}, C[\ ]) \\
(3) & C[(\mathbf{p}, C_0[\ ])V] \overset{C}{\longmapsto} C_0[V] \\
(4) & C[\mathcal{A}M] \overset{C}{\longmapsto} M.
\end{array}$$

Theorem 3.3 (C-simulation). *For any program M, $\mathcal{J}(eval_{CC}(M)) = eval_C(M)$.*

Proof. The proof is similar to the previous ones. The major difference is that the C-rewriting system does not mirror all CC-moves directly. One can only prove that if $s_1 \overset{CC}{\longmapsto} s_2$ then $|s_1| \overset{C}{\longmapsto}_= |s_2|$. This is not surprising since the C-transition function was obtained from the CC-machine by *dropping* all the bookkeeping rules. □

The three simulation theorems immediately imply:

Theorem 3.4. *For any program M, $\mathcal{J}(\mathcal{S}(\mathcal{R}(eval_{CEK}(M)))) = eval_C(M)$.*

From a different point of view this theorem stipulates that $eval_{CEK}$ should be redefined. Instead of returning a semantic value, the function should unload the machine with the function $\mathcal{J} \circ \mathcal{S} \circ \mathcal{R}$. Since $\mathcal{J}$ and $\mathcal{S}$ behave like the identity function on Λ_c, this would certainly make sense for continuation-free results. The results are Λ_c-terms; the unload function is

equal to $\mathcal{R}$ which, when restricted to Λ, is the unload function for the SECD-machine [12].

The case when the result of a program contains a continuation needs some further consideration. Continuations represent machine behavior. It is therefore not clear what it means when a *batch* computation returns a continuation as a (part of the) result. One naturally wants to interpret results as numbers, truth values, *etc*. On the other hand, if a machine is used for *interactive* computations where intermediate results can be saved, the user or programmer can only be interested in getting a continuation back for potential future use. He is then quite satisfied just to see the word "CONTINUATION." If he wants to know more, contexts tell him more than continuation codes. We can thus define:

$$eval_{CEK}(M) = \mathcal{I}(S(\mathcal{R}(V))) \text{ iff } (M, \bullet, (\mathbf{stop})) \overset{CEK}{\longmapsto} (\updownarrow, \bullet, ((\mathbf{stop})\,\mathbf{ret}\ V))$$

and, hence, we can consider $eval_{CEK}$ and $eval_C$ to be the same function.

The C-rewriting system is certainly easier to understand than the CEK-machine. But we have not yet achieved our goal of expressing all the rewriting rules within the programming language Λ_c. We still need the notion of a context in order to capture the concept of a continuation. Although we feel that contexts are naturally related to terms and that it is rather intuitive to reason with them, the rewriting system is not a calculus. It neither explicitly defines program equivalences nor justifies local transformations. What we really aim for is an equational system which gives a programmer the same power over programs as the pure λ-calculus. This is the topic of the next section.

4. The λ_c-calculus

The traditional λ-calculus can be perceived as an axiomatic theory as well as a reduction system. The two views are equivalent. The theory can only prove terms equal that are equal under the congruence relation generated from the β-reduction. From an operational or computational viewpoint the reduction system is more attractive since it exposes the rule character of the λ-calculus. Reductions also lead in a straightforward way to the standard reduction function. Thus, it is quite natural when we go the inverse direction in this section, taking the specification of the C-transition function as the point of departure and deriving the reduction system.

We clearly need the β-value reduction:

$$(\lambda x.M)N \xrightarrow{\beta_v} M[x := N] \text{ provided that } N \text{ is a value.} \qquad (\beta_v)$$

It completely captures (C1) and the underlying λ_v-calculus.

Next we turn our attention to $\mathcal{A}$-applications. According to (C4), an $\mathcal{A}$-application removes its *sk*-context. A case analysis of sk-contexts leads to appropriate *notions* of reduction. If an $\mathcal{A}$-application $\mathcal{A}M$ is within an sk-context $C[\ ]$ and to the left of some arbitrary term N, then first the N must be thrown away and, second, the rest of the context must be removed. This is a recursive problem: $C[\ ]$ can be eliminated in favor of M by simply placing $\mathcal{A}M$ in the hole. Thus, the relation should state that $C[(\mathcal{A}M)N]$ goes to $C[\mathcal{A}M]$. Since this is independent of the sk-context, we can formulate our first notion of reduction for $\mathcal{A}$-applications:

$$(\mathcal{A}M)N \xrightarrow{\mathcal{A}_L} \mathcal{A}M. \qquad (\mathcal{A}_L)$$

The second possible case, where $\mathcal{A}N$ is to the right of a value M, is treated symmetrically:

$$M(\mathcal{A}N) \xrightarrow{\mathcal{A}_R} \mathcal{A}N \text{ provided that } M \text{ is a value.} \qquad (\mathcal{A}_R)$$

This covers all but the *base* case of sk-contexts.

The case of the empty context requires special treatment. An occurrence of $\mathcal{A}M$ at the root of a term must evaluate to M, but this cannot be a proper reduction. One can only apply this rule when the $\mathcal{A}$-application is not embedded in a term. Otherwise the reduction system becomes inconsistent. Consider, for example, the expression $(\mathcal{A}\mathbf{I})\mathbf{K}$. Applying the $\mathcal{A}_L$-step results in $\mathcal{A}\mathbf{I}$; the top-level rule then leads to $\mathbf{I}$. When the top-level relation is first applied to the embedded $\mathcal{A}$-application, we get: $(\mathcal{A}\mathbf{I})\mathbf{K}$ goes to $\mathbf{IK}$ which, in turn, results in $\mathbf{K}$. $\mathbf{I}$ would equal $\mathbf{K}$ and this is inconsistent with the λ-calculus. We therefore introduce this top-level relation as a *computation rule* and use a $\triangleright$ instead of $\rightarrow$:

$$\mathcal{A}M \triangleright_{\mathcal{A}} M. \qquad (\mathcal{A}_T)$$

When we build the calculus later, care must be taken to add this computation rule at the right place.

The considerations for $\mathcal{C}$-applications move along the same line. We need to satisfy equations (C2) and (C3). Again, (C2) specifies that the context of a $\mathcal{C}$-application must be removed. So we expect that the $\mathcal{C}$-reduction rules must be designed according to the position of $\mathcal{C}M$ in an sk-context and that they must be similar to $\mathcal{A}$-reductions. For example, the expression $(\mathcal{C}M)N$ must relate to a term $\mathcal{C}X$ for some term X.

For the correct design of X we appeal to the intended semantics of the $\mathcal{C}$-application. The $\mathcal{C}$-application must capture the current continuation and supply it to its argument. Hence, if X is the next $\mathcal{C}$-argument, it will be applied to the continuation which stands for the rest of the context. This continuation must be passed on to the original $\mathcal{C}$-argument M. Furthermore, M's context also includes an application with N as the argument. In other words, if we let f be the function which must be applied to N, then the continuation of $\mathcal{C}M$ could be characterized by $\kappa(fN)$ where κ stands for the continuation of $\mathcal{C}X$. Since the continuation gets the function when it is invoked, it must be an abstraction whose parameter is f: $\lambda f.\kappa(fN)$. The term X, on the other hand, must be a function which accepts the continuation κ and passes it to M via $\lambda f.\kappa(fN)$. A first approximation of X is hence $\lambda\kappa.M(\lambda f.\kappa(fN))$. This satisfies (C2) since it removes the context of a $\mathcal{C}$-application and applies its argument to some encoding of the context. But continuations also need to respect (C3).

The rewriting rule (C3) demands that, when a continuation is invoked, the current context is removed. This reflects the fact that upon a continuation invocation, the CEK-machine ignores the current continuation. It means for our Λ_c-continuations that the first action must be an *abort* action to remove the curent context. Hence, $\lambda f.\mathcal{A}(\kappa(fN))$ is the correct continuation for M. The symmetric case where $\mathcal{C}N$ is to the right of a value M is treated in a similar way and so we define the two notions of reduction for the $\mathcal{C}$-application by:

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

$$M(\mathcal{C}N) \xrightarrow{\mathcal{C}_R} \mathcal{C}\lambda\kappa.N(\lambda v.\mathcal{A}(\kappa(Mv))) \text{ provided that } M \text{ is a value.} \qquad (\mathcal{C}_R)$$

We still need to investigate the case of the empty context, *i.e.*, the occurrence of a $\mathcal{C}$-application at the root of a term. The $\mathcal{C}$-argument M must now be applied to a function which simulates the continuation-point $(\mathbf{p}, [\ ])$. The natural choice is $\lambda x.\mathcal{A}x$. Again, this relation is not a proper notion of reduction but a computation rule:

$$\mathcal{C}M \rhd_{\mathcal{C}} M(\lambda x.\mathcal{A}x). \qquad (\mathcal{C}_T)$$

With this last rule we have derived all the reduction and computation rules that are intuitively needed for a standard reduction function equivalent to $\overset{C}{\longmapsto}$.

Defining notions of reduction is only the first step on the way to a reduction system. The next one is to build a one-step reduction relation. A *one-step reduction relation* is the extension of a notion of reduction to a relation which is compatible with the syntactic constructors. In other words, the extended relation connects terms which are the same except for two subterms related by a reduction rule. In our case four syntactic constructions must be considered: abstraction, application, $\mathcal{C}$-application, and $\mathcal{A}$-application. The two computation rules cannot be included in this step since they are not applicable to nested subterms. Definition 3 contains a formal description of the one-step reduction relation $\rightarrow_c$.

The last step in the development of a calculus is to make a congruence relation out of the reduction relation, *i.e.*, an equivalence relation which respects the syntactic constructors. Conforming to tradition, we do this in two stages: $\twoheadrightarrow_c$ is the transitive-reflexive closure of $\rightarrow_c$; its respective equivalence relation is $=_c$. This, however, is not yet the final goal. We still need to build in the computation rules. Without computation rules it is impossible to find a standard reduction function which simulates the machine evaluation: occurrences of $\mathcal{C}$- and $\mathcal{A}$-applications at the root of a term cannot be removed. We extend the reduction relation $\twoheadrightarrow_c$ to a computation relation $\rhd_k$ by adding the top-level relations. Forming the symmetric, reflexive, and transitive closure of $\rhd_k$ results in an equivalence relation $=_k$ which establishes equality among terms according to reductions *and* computations. All these concepts are summarized in Definition 3.

The relation $=_k$ determines the λ_c-calculus and we write $\lambda_c \vdash M =_k N$ if the terms M and N are equal under $=_k$. This calculus is not traditional in the sense that it uses incompatible relations. The congruence relation $=_c$ is somewhat weaker but more traditional and we consider it as a subcalculus. We also write $\lambda_c \vdash M =_c N$ when we refer to proofs within the subcalculus.

The preceding derivation of the λ_c-calculus has produced a system which is similar to the one described in our earlier paper [3]. The C-rewriting system and the previous experience directed our search this time. The important difference is that the invocation of a continuation immediately removes the current context. The correspondence of the λ_c-calculus to the real machine is almost built-in. Before we can discuss this issue further, we need to recall some earlier results on the logical properties of the calculus.

For the next section two questions are of importance:

- Are the relations $\overset{c}{\rightarrow}$ and $\rhd_k$ Church-Rosser?, and
- Is there a standard reduction function?

Since the relation $\rhd_k$ is not a compatible relation, it is clear that it cannot be Church-Rosser in the classical sense, but that we have to check whether it satisfies the diamond property. The reduction relation $\twoheadrightarrow_c$ can be treated in a more conventional manner. The following theorem states our version of the CR-theorem for the λ_c-calculus:

Definition 3: The λ_c-calculus

Let $\xrightarrow{c} = \xrightarrow{C_I} \cup \xrightarrow{C_S} \cup \xrightarrow{A_I} \cup \xrightarrow{A_S} \cup \xrightarrow{\beta_S}$. Then define the *one-step C-reduction* $\rightarrow_c$ as the compatible closure of $\xrightarrow{c}$:

$$M \xrightarrow{c} N \Rightarrow M \rightarrow_c N;$$
$$M \rightarrow_c N \Rightarrow \lambda x.M \rightarrow_c \lambda x.N;$$
$$M \rightarrow_c N \Rightarrow ZM \rightarrow_c ZN, MZ \rightarrow_c NZ \quad \text{for } Z \in \Lambda_c;$$
$$M \rightarrow_c N \Rightarrow \mathbf{C}M \rightarrow_c \mathbf{C}N;$$
$$M \rightarrow_c N \Rightarrow \mathbf{A}M \rightarrow_c \mathbf{A}N.$$

The *C-reduction* is denoted by $\twoheadrightarrow_c$ and is the transitive-reflexive closure of $\rightarrow_c$. We denote the smallest congruence relation generated by $\twoheadrightarrow_c$ with $=_c$ and call it *C-equality*.
The *computation* $\triangleright_k$ is defined by: $\triangleright_k = \triangleright_C \cup \triangleright_A \cup \twoheadrightarrow_c$. The relation $=_k$ is the smallest equivalence relation generated by $\triangleright_k$. We refer to it as *computational equality* or just *K-equality*.
The left-hand side of the reduction and computation rules are called *C-redexes*. A *C-normal form* M is a term that does not contain a C-redex. A term M *has a C-normal form* N if $M =_k N$ and N is in C-normal form.

Theorem 4.1 (Church-Rosser).

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

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

(iii) If $M =_c N$ then there exists an L such that $M \twoheadrightarrow_c L$ and $N \twoheadrightarrow_c L$.

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

The proof of this theorem is a modification of the one for the traditional λ-calculus [3].

We also need to show the existence of a standard reduction function. A standard reduction function is *the* function which reduces a term to a value by performing outermost-leftmost reductions or computation steps. It is defined in two stages. First, the relation $\xrightarrow{c}$ is extended to a function which works on all sk-contexts. Second, the computation steps are added. Definition 4 contains the formal specification. The respective theorem is:

Theorem 4.2. *$M \triangleright_k^* N$ for some value N iff $M \mapsto_{ck}^* N'$ for some value N'.*

In other words, if a program can be interpreted as a value, then the standard reduction function will produce a value. The theorem is a direct consequence of the Curry-Feys Standardization Theorem for the λ_c-calculus [3]. The next question is whether the value of a program produced by the machine is equivalent to the value produced by the standard reduction function. This is a part of the correspondence problem discussed in the next section.

Definition 4: Standard reduction sequences and functions

Given an sk-context $C[\ ]$ and $M \xrightarrow{c} N$, then the *standard reduction function*, $\mapsto_{sc}$, *for* $\xrightarrow{c}$ maps $C[M]$ to $C[N]$:

$$C[M] \mapsto_{sc} C[N].$$

The *standard reduction function for* $\triangleright_k$ extends $\mapsto_{sc}$ to computations:

$$\mapsto_{sk} = \triangleright_{\mathcal{C}} \cup \triangleright_{\mathcal{A}} \cup \mapsto_{sc} .$$

$\mapsto_{sk}^+$ and $\mapsto_{sk}^*$ stand for the transitive and transitive-reflexive closure of $\mapsto_{sk}$, respectively; $\mapsto_{sk}^i$ indicates i applications of $\mapsto_{sk}$.

5. The machine-calculus correspondence

In order to prove the equivalence of the machine semantics with the operational rewriting semantics of λ_c we need to show that the standard reduction function simulates the rules (C1) through (C4). As in the previous transition steps, we must construct a morphism from Λ_p to Λ_c since the two functions work on different term sets.

The only real task of the morphism between Λ_p and Λ_c is to encode continuation points—or sk-contexts—as terms. We had the same goal when we designed the notions of reduction for $\mathcal{C}$-applications, so we can use these relations as an orientation.

The empty context in a continuation point means that the continuation was captured with a $\mathcal{C}$-application at the root of the term. Hence, $[\ ]$ maps to $\lambda x.\mathcal{A}x$. If the hole is to the left of some arbitrary term P in some context $C[\ ]$, then a $\mathcal{C}$-application would use $\mathcal{C}_L$ to construct the next piece of the continuation. This new piece would look like $\lambda f.\mathcal{A}(\kappa(fP))$ where κ stands for the encoding of $C[\ ]$ and so we are led to the following inductive definition of the map $[\![\cdot]\!]_c$ from contexts to terms:

$$\begin{aligned}
[\![[\ ]]\!]_c &\equiv \lambda x.\mathcal{A}x \\
[\![C[[\ ]P]]\!]_c &\equiv \lambda f.\mathcal{A}([\![C[\ ]]\!]_c(f\overline{P})) \\
[\![C[V[\ ]]]\!]_c &\equiv \lambda v.\mathcal{A}([\![C[\ ]]\!]_c(\overline{V}v)).
\end{aligned}$$

The map from Q to $\overline{Q}$ replaces continuation points in Q by terms:

$$\overline{\langle \mathbf{p}, C[\ ]\rangle} \equiv [\![C[\ ]]\!]_c,\ \overline{x} \equiv x,\ \overline{\lambda x.M} \equiv \lambda x.\overline{M},\ \overline{MN} \equiv \overline{M}\,\overline{N},\ \overline{\mathcal{C}M} \equiv \mathcal{C}\overline{M},\ \overline{\mathcal{A}M} \equiv \mathcal{A}\overline{M}.$$

Given this morphism, we could now attempt to prove a simulation theorem similar to the ones in Section 3. The β_v-step, *i.e.* (C1), is clearly reflected in the definition of $\mapsto_{sk}$. It is also easy to see that the two C-transition rules (C2) and (C4) are simulated by the standard reduction function. Both rules were a major guide in the derivation of the reduction system and the map $[\![\cdot]\!]_c$ was designed according to the resulting notions of reduction:

Lemma 5.1. *For any sk-context* $C[\ ]$,

(i) $C[\mathcal{C}M] \mapsto_{sk}^+ M[\![C[\ ]]\!]_c$, *and*

(ii) $C[\mathcal{A}M] \mapsto_{sk}^{+} M$.

Proof. Both statements are proved by an induction on the depth of the redex in the context [3]. □

We are, however, unable to show that the standard reduction function satisfies rule (C3). This transition rule requires that a continuation invocation removes the current context and that it continues as if the old context—filled with the argument—were the new term. The first condition is clearly implemented since continuations immediately perform an $\mathcal{A}$-application. The second one causes problems. In the λ_c-calculus continuations are constructed to *simulate* the behavior of contexts, but in the machine continuations *are* contexts. Thus, when a continuation is to be captured after another one was invoked, the transition in the machine and the one via the sk-function diverge. The machine simply labels the current context which contains the old continuation context; the sk-reduction sequence encodes for a second time the term which simulates the former continuation.

The nature of the problem is best illustrated with an example. Suppose $\langle \mathbf{p}, C[[\ ]V] \rangle$ is invoked on the value F: $\langle \mathbf{p}, C[[\ ]V] \rangle F \xrightarrow{C} C[FV]$. Furthermore, assume that the application FV evaluates to $D[\mathcal{C}P]$ after some β_v-steps. Then the C-reduction sequence reaches the term $P\langle \mathbf{p}, C[D[\ ]] \rangle$. According to Lemma 5.1, if $K_c \equiv [\![C[\ ]]\!]_c$, the corresponding reduction sequence in the λ_c-calculus begins with:

$$[\![C[[\ ]V]]\!]_c \overline{F} \mapsto_{sk}^{+} K_c(\overline{F}\,\overline{V}).$$

The next few β_v-steps for $\overline{F}\,\overline{V}$ are correctly performed by the sk-function:

$$K_c(\overline{F}\,\overline{V}) \mapsto_{sk}^{+} K_c\overline{D}[\mathcal{C}\overline{P}].$$

This last term also constructs a continuation—just like its correspondent $C[D[\mathcal{C}P]]$—, but the continuation encodes the term K_c instead of the context $C[\ ]$:

$$K_c\overline{D}[\mathcal{C}\overline{P}] \mapsto_{sk}^{+} \overline{P}[\![K_c\overline{D}[\ ]]\!]_c.$$

One readily sees that $[\![C[D[\ ]]]\!]_c$ is not equal to $[\![K_c\overline{D}[\ ]]\!]_c$. This means that a naïve version of the simulation theorem fails. The best we can hope for is that the sk-simulation of the C-transition function preserves a relation between continuation points and terms.

From the above lemma and the example one could suspect that a continuation point like $\langle \mathbf{p}, C[D[\ ]] \rangle$ is related to the terms $[\![C[D[\ ]]]\!]_c$ and $[\![[\![C[\ ]]\!]_c D[\ ]]\!]_c$. However, the situation in our example could recur many times. Instead of having two contexts composing a new one, we would then have several of them. In fact, we have to take into account all possible finite decompositions of a given context into smaller contexts, including the empty one. Each sub-context can be encoded as a term by itself; each of these encoded contexts can be a part of a bigger context which is being encoded. We have formalized this relation in Definition 5.

The relation $\approx_p$ in Definition 5 is implicit. It is well-suited to capture the different continuation representations from the example, but it does not expose the structure of the terms which stand for continuation points. A brief investigation reveals that these terms are rather similar and that they share another important attribute: they are behaviorally indistinguishable with respect to β_v-steps in standard reduction sequences. Empty contexts

Definition 5: The continuation point correspondence

The relation $\approx_p$ relates terms of Λ_p and Λ_p-sk-contexts to terms in Λ_c and sk-contexts. It is defined inductively by:

$\langle \mathbf{p}, C[\ ]\rangle \approx_p K_c$ iff

for some finite number of sk-contexts $C_1[\ ], \ldots, C_n[\ ]$ such that $C[\ ] \equiv C_1[C_2[\ldots C_n[\ ]\ldots]]$ the term K_c is determined by:

$$K_c \equiv [\ldots[[\overline{C}_1[\ ]]_c \overline{C}_2[\ ]]_c \ldots \overline{C}_n[\ ]]_c$$

where $C_i[\ ] \approx_p \overline{C}_i[\ ]$ for all i.

If $P \approx_p \overline{P}$ and $Q \approx_p \overline{Q}$, then, for all x

$$x \approx_p x,\ \lambda x.P \approx_p \lambda x.\overline{P},\ PQ \approx_p \overline{P}\,\overline{Q},\ \mathcal{C}P \approx_p \mathcal{C}\overline{P},\ \mathcal{A}P \approx_p \mathcal{A}\overline{P}.$$

For sk-contexts we have to add

$$[\ ] \approx_p [\ ].$$

Note, we use the notation $\overline{P}$ ambiguously for both the result of mapping P to $\overline{P}$ and a term in Λ_c that is related to a term P in Λ_p via $\approx_p$.

in the partitioning of a continuation-point context add an extra $\lambda x.\mathcal{A}x$ to its representation. On the other hand, if there is a proper term contained in the context, exactly one of the sub-contexts will cover it. Therefore, each subterm appears exactly once in the representation. Putting this together, we see that the terms that are related to a continuation point are the same modulo some occurrences of $\lambda x.\mathcal{A}x$:

Lemma 5.2. *Define $K_{i+1} \equiv \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)(K_i x))$ to be a term sequence which is parameterized with respect to its first element K_1. Then the relation of a continuation point to a term is characterized by exactly one of the following three statements:*

(i) $\langle \mathbf{p}, [\ ]\rangle \approx_p K_i$ *where* $K_1 \equiv \lambda x.\mathcal{A}x$, *or*

(ii) $\langle \mathbf{p}, C[[\ ]P]\rangle \approx_p K_i$ *where* $K_1 \equiv \lambda f.\mathcal{A}(K_c(f\overline{P}))$, $\langle \mathbf{p}, C[\ ]\rangle \approx_p K_c$, *and* $P \approx_p \overline{P}$, *or*

(iii) $\langle \mathbf{p}, C[U[\ ]]\rangle \approx_p K_i$ *where* $K_1 \equiv \lambda v.\mathcal{A}(K_c(\overline{U}v))$, $\langle \mathbf{p}, C[\ ]\rangle \approx_p K_c$, *and* $U \approx_p \overline{U}$.

Furthermore, we can generalize this to

$$\langle \mathbf{p}, C[D[\ ]]\rangle \approx_p [K_c D[\ ]]_c \text{ iff } \langle \mathbf{p}, C[\ ]\rangle \approx_p K_c.$$

Proof. First note that (i), (ii), and (iii) cover all possible cases of sk-contexts. One of them must match a particular sk-context. Furthermore, the proof of all three statements is naturally divided into two parts: one for $i = 1$ and one for $i > 1$. The latter is the same in all cases. For the former we demonstrate how to prove case (ii) as a typical example.

From the definition of $\approx_p$ we know that for any context $C[[\ ]P]$ and finite number of contexts $C_i[\ ]$ which compose $C[[\ ]P]$, we have

$$\langle \mathbf{p}, C[[\ ]P]\rangle \approx_p [\ldots[[\overline{C}_1[\ ]]_c \overline{C}_2[\ ]]_c \ldots \overline{C}_n[\ ]]_c.$$

For the base case we assume that $C_n[\ ] \neq [\ ]$. Then, in (ii), $C_n[\ ] \equiv D[[\ ]P]$ for some context $D[\ ]$ since P is the term next to the hole in the continuation-point context. This implies that

$$[\ldots[[\overline{C}_1[\ ]]_c\overline{C}_2[\ ]]_c\ldots\overline{D}[[\ ]\overline{P}]]_c \equiv \lambda f.\mathcal{A}([\ldots[[\overline{C}_1[\ ]]_c\overline{C}_2[\ ]]_c\ldots\overline{D}[\ ]]_c(f\overline{P})).$$

On the other hand, $C[\ ] \equiv C_1[\ldots D[\ ]\ldots]$ and thus

$$\langle \mathbf{p}, C[\ ]\rangle \approx_p [\ldots[[\overline{C}_1[\ ]]_c\overline{C}_2[\ ]]_c\ldots\overline{D}[\ ]]_c.$$

This proves the case for $i = 1$.

For the case where $i > 1$ assume that the last few, say $j \geq 1$, contexts in this sequence are empty, *i.e.* equal to $[\ ]$. By factoring out the first one, we get

$$\begin{aligned}[\ldots[[\overline{C}_1[\ ]]_c\overline{C}_2[\ ]]_c\ldots[\ ]]_c &\equiv \lambda x.\mathcal{A}([[\ ]]_c([\ldots[[\overline{C}_1[\ ]]_c\overline{C}_2[\ ]]_c\ldots]_c x))\\ &\equiv \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)([\ldots[[\overline{C}_1[\ ]]_c\overline{C}_2[\ ]]_c\ldots]_c x)).\end{aligned}$$

Thus we see that, as mentioned above, every empty context adds one term $\lambda x.\mathcal{A}x$. Hence, $\langle \mathbf{p}, C[[\ ]P]\rangle \approx_p K_{j+1}$ and this concludes the proof.

The generalization follows immediately. □

Lemma 5.2 verifies our claim about the behavior of the terms in the representation set of $\langle \mathbf{p}, C[\ ]\rangle$. They invoke a continuation and, since continuations always remove the current context, none of the $\lambda x.\mathcal{A}x$ will ever play a role in an evaluation.

Proposition 5.3. *Define three series K_i as in Lemma 5.2. with the initial terms $\lambda x.\mathcal{A}x$, $\lambda f.\mathcal{A}(K_c(f\overline{P}))$, and $\lambda v.\mathcal{A}(K_c(\overline{U}v))$. Then we can show:*

$$\begin{aligned}K_i &\twoheadrightarrow_c \lambda x.\mathcal{A}\ldots(i\text{-times})\ldots x,\\ K_i &\twoheadrightarrow_c \lambda f.\mathcal{A}\ldots(i\text{-times})\ldots(K_c(f\overline{P})),\\ K_i &\twoheadrightarrow_c \lambda v.\mathcal{A}\ldots(i\text{-times})\ldots(K_c(\overline{U}v)),\end{aligned}$$

respectively.

Remark. If we had formalized standard reduction *sequences*, we could replace $\twoheadrightarrow_c$ by standard reduction steps. **End of Remark**

Proof. Clearly, $K_i \equiv \lambda x.\mathcal{A}M_i^x$ for some (open) term M_i^x. Hence,

$$\begin{aligned}K_{i+1} &\twoheadrightarrow_c \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)((\lambda x.\mathcal{A}M_i^x)x))\\ &\twoheadrightarrow_c \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)(\mathcal{A}M_i^x[x := x]))\\ &\twoheadrightarrow_c \lambda x.\mathcal{A}(\mathcal{A}M_i^x).\end{aligned}$$

But, the three M_i^x's for the base cases are x, $(K_c(x\overline{P}))$, and $(K_c(\overline{U}x))$, respectively. □

All continuations that are related to a continuation point behave similarly when invoked; the difference is the number of abort operations. Thus, we can show that evaluations via $\mapsto_{ek}$ and $\xrightarrow{C}$ only differ in their outcome. First, we prove that $\mapsto_{ek}$ mirrors C-transition steps as long as no continuation is invoked:

Lemma 5.4. *Assume $C[\ ] \approx_p \overline{C}[\ ]$, $P \approx_p \overline{P}$, and $U \approx_p \overline{U}$. The simulation of the rules (C1), (C2), and (C4) via $\mapsto_{ek}$ respects $\approx_p$:*

(i) *if* $C[(\lambda x.P)U] \xmapsto{C} C[P[x := U]]$ *then* $\overline{D}[(\lambda x.\overline{P})\overline{U}] \mapsto_{sk} \overline{D}[\overline{P}[x := \overline{U}]]$ *for any sk-context* $\overline{D}[\ ]$;

(ii) *if* $C[\mathcal{A}P] \xmapsto{C} P$ *then* $\overline{C}[\mathcal{A}\overline{P}] \mapsto_{sk}^{+} \overline{P}$;

(iii) *if* $C[\mathcal{C}P] \xmapsto{C} P(\mathbf{p}, C[\ ])$ *then* $\overline{C}[\mathcal{C}\overline{P}] \mapsto_{sk}^{+} \overline{P}[\![\overline{C}[\ ]]\!]_c$.

Proof. The first statement reiterates that β-steps are simulated independently of the context. Points (ii) and (iii) are consequences of Lemma 5.1. □

Things get more complicated when a continuation is invoked. The sk-reduction sequence contains a series of auxiliary moves in order to simulate the jump to a different context in the C-reduction sequence. Since proper simulation steps are interspersed in this detour, it is impossible to prove a corresponding lemma for (C3). However, a direct proof that continuation invocations are correctly implemented by $\mapsto_{sk}$ is possible:

Lemma 5.5. *Suppose* $(\mathbf{p}, C_0[\ ]) \approx_p K_0$, $V \approx_p \overline{V}$, *and* $U \approx_p \overline{U}$. *Then,*

$$C[(\mathbf{p}, C_0[\ ])V] \xmapsto{C}{}^{+} U \text{ iff } \overline{C}[K_0\overline{V}] \mapsto_{sk}^{+} \overline{U}.$$

Proof. The condition $C[\ ] \approx_p \overline{C}[\ ]$ is unnecessary for the antecedent since a continuation immediately performs some $\mathcal{A}$-applications.

The equivalence is proved by an induction on the unique number of steps, n, in the $\xmapsto{C}$-reduction sequence from $C[(\mathbf{p}, C_0[\ ])]$ to U. We proceed by a case analysis on the structure of $C_0[\ ]$:

(skC1) $C_0[\ ] \equiv [\ ]$: This case is trivial. It implies that

$$K_0 \equiv K_1 \equiv \lambda x.\mathcal{A}x \text{ or}$$
$$K_0 \equiv K_2 \equiv \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)((\lambda x.\mathcal{A}x)x)), \textit{ etc.}$$

In any case, we have

$$(\mathbf{p}, C_0[\ ])V \xmapsto{C} V \text{ and } K_0\overline{V} \mapsto_{sk}^{+} \overline{V}.$$

(skC2) $C_0[\ ] \equiv D[[\ ]P]$ for some Λ_p-sk-context and term P. Now we know from Lemma 5.2 that

$$K_0 \equiv K_1 \equiv \lambda f.\mathcal{A}(K_D(f\overline{P})), \text{ or}$$
$$K_0 \equiv K_2 \equiv \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)(K_1 x)), \textit{ etc.}$$

where $(\mathbf{p}, D[\ ])V \approx_p K_D$ and $P \approx_p \overline{P}$. The two reduction sequences start out with

$$C[(\mathbf{p}, C_0[\ ])V] \xmapsto{C} D[VP]$$

and

$$\overline{C}[K_0\overline{V}] \mapsto_{sk}^{+} K_D(\overline{V}\,\overline{P}).$$

Next, we consider the possible evaluations of VP and $\overline{V}\,\overline{P}$. The previous lemma reassures us that as long as the rule (C1) is used the context plays no role and, more importantly, the relation $\approx_p$ is preserved. The first transition step which does not conform to (C1) is the distinguishing criteria for the rest of the reduction sequence. Since this sequence is finite, four cases must be analyzed:

a) $VP \overset{C}{\longmapsto}^{+}_{(C1)} W$ where W is a value. This means that $\overline{VP} \mapsto_{sk}{}^{+} \overline{W}$ and we have the following development for the C-transition:

$$D[VP] \overset{C}{\longmapsto}^{+} D[W].$$

For the one according to $\mapsto_{sk}$ we get

$$K_D(\overline{V}\,\overline{P}) \mapsto_{sk}{}^{+} K_D\overline{W}.$$

By assumption we know that

$$D[W] \overset{C}{\longmapsto}^{m} U \text{ with } m \leq n-2.$$

From the definition of $\overset{C}{\longmapsto}$ we see that, $D[W] \overset{C}{\longmapsto}^{m} U$ iff $(\mathbf{p}, D[\ ])W \overset{C}{\longmapsto}^{m+1} U$. Thus, we can safely replace $D[W]$ by $(\mathbf{p}, D[\ ])W$ since $m+1 \leq n-1$. But note, $(\mathbf{p}, D[\ ]) \approx_p K_D$ and so, by inductive hypothesis, we get the desired conclusion.

b) $VP \overset{C}{\longmapsto}^{\bullet}_{(C1)} E[\mathcal{A}Q]$ and $\overline{VP} \mapsto_{sk}{}^{\bullet} \overline{E}[\mathcal{A}\overline{Q}]$ for some term Q and sk-context $E[\ ]$. Comparing the two reduction sequences

$$D[VP] \overset{C}{\longmapsto}^{\bullet} D[E[\mathcal{A}Q]] \overset{C}{\longmapsto} Q$$

and

$$K_D(\overline{V}\,\overline{P}) \mapsto_{sk}{}^{\bullet} K_D\overline{E}[\mathcal{A}\overline{Q}] \mapsto_{sk}{}^{\bullet} \overline{Q},$$

we see that both continue with related terms. From this point on, two developments are possible: the rest of the sequence either uses the (C3) rule or it doesn't:

b1) If $Q \overset{C}{\longmapsto}^{\bullet} U$ does not use (C3), then according to Lemma 5.4 $\overline{Q} \mapsto_{sk}{}^{\bullet} \overline{U}$ is immediate.

b2) Suppose (C3) is used a first time. That means, that $Q \overset{C}{\longmapsto}^{\bullet} F[(\mathbf{p}, F_0[\ ])W]$ and also by Lemma 5.4 that $\overline{Q} \mapsto_{sk}{}^{\bullet} \overline{F}[K_F\overline{W}]$ such that $(\mathbf{p}, F_0[\ ]) \approx_p K_F$. Since the reduction sequence is at least one step shorter, we can now apply our inductive hypothesis and this finishes case b).

c) $VP \overset{C}{\longmapsto}^{\bullet}_{(C1)} E[\mathcal{C}Q]$ and $\overline{VP} \mapsto_{sk}{}^{\bullet} \overline{E}[\mathcal{C}\overline{Q}]$. The reduction sequence according to $\mapsto_{sk}$ continues as:

$$K_D(\overline{V}\,\overline{P}) \mapsto_{sk}{}^{\bullet} K_D\overline{E}[\mathcal{C}\overline{Q}] \mapsto_{sk}{}^{+} \overline{Q}[\![K_D\overline{E}[\ ]]\!]_c.$$

The transition rule (C2) accomplishes the capturing of this continuation in one step:

$$D[VP] \overset{C}{\longmapsto}^{\bullet} D[E[\mathcal{C}Q]] \overset{C}{\longmapsto} Q(\mathbf{p}, D[E[\ ]]).$$

By assumption $(\mathbf{p}, D[\ ]) \approx_p K_D$ and, hence, $(\mathbf{p}, D[E[\ ]]) \approx_p [\![K_D\overline{E}[\ ]]\!]$ by Lemma 5.2. The rest of this subcase is as in b).

d) $VP \overset{C}{\longmapsto}^{\bullet}_{(C1)} E[(\mathbf{p}, E_0[\ ])W]$ and $\overline{VP} \mapsto_{sk}{}^{\bullet} \overline{E}[K_E\overline{W}]$ such that $(\mathbf{p}, E_0[\ ]) \approx_p K_E$. This is an instance of the inductive hypothesis and the case (skC2) is finished.

(skC3) $C_0[\ ] \equiv D[P[\ ]]$ for some Λ_p-sk-context and value P. Again, the respective continuations are characterized by $K_1 \equiv \lambda v.\mathcal{A}(K_D(\overline{P}v))$, *etc.* The two reduction sequences immediately arrive at the same constellation as in (skC2):

$$C[(\mathbf{p}, C_0[\ ])V] \xmapsto{C} D[PV]$$

and

$$\overline{C}[K_0\overline{V}] \mapsto_{sk}^{+} K_D(\overline{P}\,\overline{V}).$$

The rest is analogous to the previous case. □

Putting the previous two lemmas together, the following theorem is obvious:

Theorem 5.6 (Sk-simulation). *For any program $M \in \Lambda_c$, values V, $\overline{V}$ such that $V \approx_p \overline{V}$*

$$M \xmapsto{C}^{+} V \text{ iff } M \mapsto_{sk}^{+} \overline{V}.$$

Since $V \approx_p \overline{V}$ implies $V \equiv \overline{V}$ for $V \in \Lambda_c$, the theorem can be specialized:

Corollary 5.7. *For any program $M \in \Lambda_c$ whose result V is continuation-free,*

$$M \xmapsto{C}^{+} V \text{ iff } M \mapsto_{sk}^{+} V.$$

Finally, we can note that $eval_{CEK}$ is only defined if the program is equivalent to a value:

Corollary 5.8. *There exists a value N such that $\lambda_c \vdash M =_k N$ iff $eval_{CEK}(M)$ is defined.*

Informally, these results mean that the CEK-machine is characterized by a standard reduction function (and sequence) of a calculus modulo some syntactic difference. In order to eliminate this difference, we would have to change the standard reduction function in such a way that a term $K(\mathcal{C}M)$ evaluates to MK for a continuation K. From the above definition of $[\![\cdot]\!]_c$ one can see that recognizing terms as continuations is possible. But, a user could easily construct such a term K and then the normal evaluation sequence would be preferable: without knowing the history of a term, it is impossible to know when to apply the new rule.

Although the difference cannot be eliminated, it is not stringent. Since—as dicussed after Theorem 3.4—the result of a computation is generally considered to be some kind of ground value, *e.g.* number, boolean value, tree, list, *etc.*, and not a continuation, we are safe. Corollary 5.8 assures us that we get the correct result back if we encode ground values in Λ_c or even better in Λ. If continuations are a legitimate part of the result, then it is their potential behavior that is interesting. In this case we are safe because of Proposition 5.3. All terms that are related to a continuation point are behaviorally equivalent. Thus, we can indeed assume that $eval_{CEK}$ and the operational semantics of λ_c are equivalent.

A disadvantage of the above theorem and corollaries is their dependence on the standard reduction function of the calculus. One would prefer to interpret terms in a less operational way using $=_k$ instead. Traditionally, one thinks of terms as functions from some set of basic constants to basic constants. Since we have neither a model for our calculus nor constant names in our language, we follow Morris's example [11] and define the set of *basic values* to be the set of closed normal forms in Λ. This definition is not restrictive because ground values can all be encoded in a normal-form representation, and on the other hand, it allows us to compare values syntactically.

Next we define two interpretations of terms [11, 12]. For all $n \geq 0$, the *calculus interpretation* of a term M is the function $\mathcal{I}^n_M = \{(N_1, \ldots, N_n, U) | \lambda_c \vdash MN_1 \ldots N_n =_k U\}$ where the N_i and U are basic values. The *machine interpretation* of a term M is the function $\mathcal{M}^n_M = \{(N_1, \ldots, N_n, U) | eval_{CEK}(MN_1 \ldots N_n) \equiv U\}$.

With these interpretations we can show that the correspondence of the CEK-machine to the λ_c-calculus is independent of a standard reduction function:

Theorem 5.9. *For any program M in Λ_c, its calculus and machine interpretation are the same for all $n \geq 0$:*

$$\mathcal{I}^n_M = \mathcal{M}^n_M.$$

Proof. The proof is a straightforward consequence of the Church-Rosser Theorem, Corollary 5.7, and Corollary 5.8. □

Theorem 5.9 essentially says that the machine and the calculus interpret a program as the same function. Given that the classical λ-calculus is for reasoning about the equivalence of these functions, the question naturally arises what proofs in λ_c *mean*.

Since the relation $=_k$ is not a congruence relation, it is clear that $M =_k N$ does *not* mean that $\mathcal{M}^n_M = \mathcal{M}^n_N$ for any $n > 0$. The relation $=_k$ only compares programs that are already supplied with all their input arguments. Intuitively, the equivalence relation $=_k$ is equating the global control intentions of programs. The subrelation $=_c$ is more like $=_{\beta_v}$: it compares the functionality and *local* control structure of terms.

The question can also be generalized to what equality in λ_c means for open expressions. In Morris's words, we ask whether equality is preserved under all possible interpretations; for Plotkin it is the question if equations in λ_v are true with respect to the machine. For this last step in our investigation we adapt Morris's $\simeq$ and Plotkin's $\approx_V$ relation. M is *operationally equivalent* to N, $M \simeq_{CEK} N$, if for any program context $C[\ ]$—a term with one hole at an arbitrary position—, $C[M]$ and $C[N]$ are programs, $eval_{CEK}$ is undefined for both, or, if it is defined for both programs, it produces the same basic value. From the above discussion about $=_k$ and $=_c$, we know that only $=_c$ implies operational equivalence. For $=_k$ we need to make sure that the terms behave equivalently in all cases, then it also implies operational equivalence:

Theorem 5.10. *For M, N in Λ_c,*

(i) if $\lambda_c \vdash M =_c N$, then $M \simeq_{CEK} N$, and

(ii) if $\lambda_c \vdash C[M] =_k C[N]$ for all sk-contexts $C[\ ]$, then $M \simeq_{CEK} N$.

Proof. The proof of (i) is easy. It is essentially a transcription of Plotkin's corresponding proof for the λ_v-calculus.

Part (ii) deserves some elaboration. Assume the hypothesis and w.l.o.g. assume that M and N are in Λ_c proper. Let $D[\ ]$ be a context such that $D[MN]$ is closed. Now, suppose that $eval_{CEK}(D[M])$ is defined and, furthermore, that it is a basic constant. By Theorem 4.2 and Corollary 5.8

$$\lambda_c \vdash D[M] =_k eval_{CEK}(D[M]).$$

Depending on the role of the fill-in term during the evaluation, we have to distinguish two cases. It is possible that the term in the hole is never a direct component of an sk-redex. Then it gets thrown away since the result is a basic constant. The conclusion is immediate.

Otherwise, at some point a closed form of M or N is an immediate component of some sk-redex in some sk-context. But note, $\lambda_c \vdash C[M] =_k C[N]$ clearly implies $\lambda_c \vdash C[M[x := L]] =_k C[N[x := L]]$ for all values L. Therefore, with the necessary generalization to multiple substitutions, we have

$$\lambda_c \vdash eval_{CEK}(D[M]) =_k eval_{CEK}(C[M[\vec{x} := \vec{L}]]) =_k eval_{CEK}(C[N[\vec{x} := \vec{L}]]).$$

Hence, by the Church-Rosser theorem, $D[M]$ and $D[N]$ produce the same basic value. □

The inverse of both statements is false. This is inherited from the λ_v-calculus for which Plotkin has already shown that it is consistent but not complete with respect to $\approx_V$.

The second point of Theorem 5.10 is important. Although $=_k$ is only an equivalence relation it induces a natural, consistent extension of $=_c$. Instead of requiring $C[M] =_k C[N]$ for *any* context in the antecedent, sk-contexts are sufficient. Since sk-contexts represent control contexts and nothing else, this statement reaffirms that $=_k$ compares control intentions in programs.

6. Discussion

In the preceding sections we have demonstrated how a calculus can be derived from an operational semantics of a programming language. Our particular example involved the derivation of a calculus for a language with non-functional control operators.

Two points deserve mentioning. First, the derivation produced a symbolic evaluation function which works on the level of programs and related concepts. This is an important tool when tracing of programs is required. Until now programs with control operators could only be understood in terms of machine implementations. Second, the existence of a calculus which corresponds to the machine stipulates that many aspects of control are independent of a particular evaluation order. The two-part definition of the calculus expresses the fact that the imperative nature of our operators only shows up at isolated points. These results also encourage us to continue our research on other, seemingly imperative programming constructs.

Acknowledgement. We thank Carolyn Talcott for her careful reading of an earlier draft. Her comments led to a simplification in the presentation of the proofs of Theorem 3.1 and Theorem 3.2.

References

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

2. CARLSSON, M. On implementing Prolog in functional programming, *New Generation Computing* **2**, 1984, 347–359.

3. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, B. DUBA. Reasoning with Continuations, *Proc. Symp. Logic in Computer Science*, 1986, 131–141; also available in extended form as Technical Report No. 191, Indiana University Computer Science Department, 1986.

4. 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.

5. HAYNES, C.T., D.P. FRIEDMAN, M. WAND. Obtaining coroutines from continuations, *Computer Languages*, to appear.

6. HAYNES, C. T. Logic continuations, Technical Report No. 183, Indiana University Computer Science Department, to appear in the proceedings of the *Third International Conference on Logic Programming*, London, Springer-Verlag, 1986.

7. JACKSON, M. A. *Principles of Program Design*, Academic Press, New York, 1975.

8. LANDIN, P.J. The mechanical evaluation of expressions, *Computer Journal* **6**(4), 1964, 308–320.

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

10. LANDIN, P.J. A formal description of ALGOL 60, in *Formal Language Description Languages for Computer Programming*, T.B. Steel (Ed.), 1966, 266–294.

11. MORRIS, J.H. *Lambda-Calculus Models of Programming Languages*, Ph.D. Thesis, Project MAC, MAC-TR-57, MIT, 1968.

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

13. REES, J. (Ed.) The revised^3 report on Scheme, Joint Technical Report Indiana University and MIT Laboratory for Computer Science, 1986, to appear in *SIGPLAN Notices*.

14. 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.

15. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages, *Proc. ACM Annual Conference*, 1972, 717–740.

16. STEELE, G. *COMMON LISP—The Language*, Digital Press, 1984.

17. SUSSMAN G.J., G. STEELE. Scheme: An interpreter for extended lambda calculus, Memo 349, MIT AI-Lab, 1975.

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