Theoretical Computer Science 52 (1987) 205-237
North-Holland

A SYNTACTIC THEORY OF SEQUENTIAL CONTROL *

Matthias FELLEISEN, Daniel P. FRIEDMAN, Eugene KOHLBECKER and Bruce DUBA

Computer Science Department, Lindley Hall 101, Indiana University, Bloomington, IN 47405, U.S.A.

Communicated by R. Milner
Received April 1986
Revised February 1987

Abstract. Sequential 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 evaluation strategy. We also prove a standardization theorem and use it to investigate the correspondence between the machine and the calculus. Together, the calculus and the rewriting machine form a syntactic theory of control, which provides a natural basis for reasoning about programs with nonfunctional control operators.

1. Deficiencies of the λ-calculus as a programming language

"The lambda calculus is a type-free theory about functions as *rules*, rather than graphs. 'Functions as rules'...refers to the process of going from argument to value,...." [1, p. 3] No other words can better express why computer scientists have been intrigued with the λ-calculus. The rule character of function evaluation comes close to a programmer's operational understanding of computer programs and, at the same time, the calculus provides an algebraic framework for reasoning about functions. Yet, this concurrence is also a major obstacle in the further development of the calculus as a programming language since it is based on simplicity rather than convenience.

The one and only means of computation in the calculus is the β-reduction rule which directly models function application. Although this suffices from a purist's point of view, it is in many cases insufficient with respect to expressiveness and inefficient with respect to the evaluation process. For example, when a recursive program discovers the final result in the middle of the computation process, it should be allowed to immediately *escape* and report its value. Similarly, in an erroneus situation a program must be able to terminate or to call an exception handler without delay. We could easily lengthen this list of examples, but the thrust is clear: functions-as-programs need more control over their evaluation.

* This paper is partly based on work supported by NSF Grants DCR 85-01277 and DCR 85-03279.

The most general solution to the control problem within the functional realm originated in denotational semantics[1] [4, 11, 19]. A program is evaluated by computing the value of its pieces and combining the results. When a particular component is being evaluated, one can think of the remaining subevaluations and the combination step as the rest of the computation or as the *continuation* of the current subevaluation. The crucial idea is to write programs in a style where functions can be used to *simulate* continuations. The programs always pass around and explicitly invoke (a function representation of) the current continuation. They are thus able to direct the evaluation process: they may decide *not* to use the current continuation, to save it for later use, or to resume a continuation from some other point in time. However, such programs look clumsy and are hard to design. It is better to introduce linguistic facilities which give programs access to the current continuation when needed. Programs using these facilities are "much simpler, easier to understand (given a little practice) and easier to write. They are also more reliable since the machine carrying out the computations constructs the continuations mechanically...".[2] Typical examples of such facilities in λ-calculus based languages are the J-operator [8, 9], label values [17], escape functions [16], call-with-current-continuation (abbreviated as call/cc) [15], and catch and throw [20].

Nonfunctional control operators "provide a way of pruning unnecessary computation and allow certain computations to be expressed by more compact and conceptually manageable programs."[3] If these operations make continuations available as first-class objects, as in Scheme or ISWIM, it is easy to imitate any desired sequential control construct, e.g., escapes, error stops, search strategies as applied in logic programming [7, 12], intelligent backtracking [5], and coroutining [21]. Even though this is widely recognized, control operators are still regarded with skepticism. Their addition seems rather ad hoc, because it only advances the calculus as a programming language. On the algebraic side there are no rules reflecting the new operations. Proofs of program properties can no longer be carried out in the syntactic domain; they must be based upon a semantic interpretation in terms of abstract machines or denotational definitions [11, 21].

In this paper, an expanded revision of two preliminary reports [2, 3], we show that the λ-calculus as an equational system can incorporate control operators and

[1] Indeed, the origin of the concept of a continuation can be traced back to Van Wijngaarden who explained in a discussion at the IFIP Working Conference on Formal Language Description Languages, 1964 [18, p. 24] that "this implementation [of procedures] is only so difficult because you have to take care of the **goto** statement. However, if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure return. That is the whole difficulty with procedure implementation. That's why this is so simple; it's exactly the same as a **goto**, only called in other words."

[2] C. Talcott about the introduction of **note** into Rum, a lexically-scoped dialect of Lisp [21, p. 68].

[3] C. Talcott wrote this remark in the context of escape mechanisms, but the spirit of her dissertation makes clear that it is also applicable to jump operations in general [21, p. 16].

that nonfunctional control may be characterized in a purely syntactic manner. Starting from an idealized Scheme with an operational semantics, we design an extended calculus-like system that captures the behavior of the additional control operators. The theory is consistent in the sense that two different derivations starting with the same term are confluent. Hence, it permits algebraic calculations in the familiar style. A standardization theorem shows that the calculus defines a programming language in its own right. Comparing the calculus and the programming language semantics, we find that the two truly correspond to each other. The standard computation function agrees with the evaluation function except for some negligible differences, and equality in the calculus implies indistinguishability on the machine. Together with the machine semantics, the calculus provides a powerful framework for reasoning about programs with nonfunctional control operators. We exemplify and discuss this in more depth in the last section.

Since our base language has an applicative-order semantics, the resulting calculus is an extension of Plotkin's λ-value calculus. Although assuming familiarity with the conventional λ-calculus [1], we do not require understanding of Plotkin's variant [14]. Furthermore, we follow Plotkin's plan [14] for the comparison of calculi and programming languages, but, to keep the paper self-contained, we include some appropriate explanations at key places.

2. Λ_c

The programming language Scheme [15] is the starting point of our language design. Of all the λ-calculus-based languages, its continuation-accessing operation, call/cc, is independent of the rest of the primitive language concepts. For example, the J-operator [8] interferes with functional abstraction; the **escape**-construct [16] introduces a second binding facility. Thus, call/cc, which simply takes an argument and applies it to the current continuation, appears to be the ideal operator. However, it also complicates the development of a calculus. When call/cc is used, it not only transfers control over the current continuation to the program, it also installs this continuation. This implies that a program has only partial control over the continuation.

To avoid these problems, we introduce a variant of call/cc into the λ-calculus term language. The next two subsections contain the extended syntax and its semantics. In the third one, we illustrate how to program with this new facility. We compare our example to a functional solution and briefly discuss the continuation-passing programming style as a generalization of the latter.

2.1. Syntax

The core of our programming language is the λ-calculus term set Λ. For the sake of simplicity, we concentrate on constant-free expressions. The extended language Λ_c includes two new types of applications: $\mathscr{C}$- and $\mathscr{A}$-applications. The formal definition of Λ_c is displayed in Definition 2.1. We adopt the notational conventions of the classical λ-calculus and write $\lambda xy.M$ for $\lambda x.\lambda y.M$, LMN for $((LM)N)$, and also $\mathcal{C}M$ for $(\mathcal{C}M)$, etc. where this is unambiguous.

Definition 2.1 (*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, k, f, v, etc. range over Var as metavariables but are also used as if they were elements of Var. $L, M, N, \ldots$ are metavariables for Λ_c. The *term set* Λ_c contains
-*variables*: x if $x \in \mathrm{Var}$;
-*abstractions*: $(\lambda x.M)$ if $M \in \Lambda_c$ and $x \in \mathrm{Var}$;
-*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 $M \in \Lambda_c$, and M is called the $\mathcal{C}$-argument;
-*$\mathcal{A}$-applications*: $(\mathcal{A}M)$ if $M \in \Lambda_c$, and M is called the $\mathcal{A}$-argument.
The union of variables and abstractions is referred to as the set of *values*; $U, V, \ldots$ are metavariables for values. Λ, the term set of the traditional λ-calculus, stands for Λ_c restricted to variables, applications, and abstractions.

The notion of free and bound variables in a term M carries over directly from the λ-calculus. Terms with no free variables are called *closed terms* or *programs*. We adopt Barendregt's convention of identifying terms ($\equiv$) that are equal except for some renaming of bound variables and his hygiene condition which says that *in a discussion, free and bound variables are assumed to be distinct.* Furthermore, we extend Barendregt's definition of the substitution function $M[x := N]$ to Λ_c in the natural way:

$$x[x := N] \equiv N, \quad y[x := N] \equiv y \qquad (x \neq y),$$
$$(\lambda y.M)[x := N] \equiv (\lambda y.M[x := N]),$$
$$(LM)[x := N] \equiv (L[x := N]M[x := N])$$

and

$$(\mathcal{C}M)[x := N] \equiv (\mathcal{C}M[x := N]), \qquad (\mathcal{A}M)[x := N] \equiv (\mathcal{A}M[x := N]).$$

2.2. Operational semantics

The intention behind the two operations $\mathcal{C}$ and $\mathcal{A}$ can be explained informally. An $\mathcal{A}$-application represents an *abort* or stop operation, which terminates the program and returns the value of its argument. Whereas such an operation is commonly found in traditional languages, $\mathcal{C}$ and its relatives are only available in λ-calculus based languages. The operation gives its argument complete control over the current continuation, that is, $\mathcal{C}$ applies its argument to an abstraction of what must be done in order to complete the program after evaluating the $\mathcal{C}$-application. This step is also called *labeling*—or capturing—of continuations with reference to label values in more traditional languages. A continuation is *invoked*—or thrown to—by applying it to a value, just like a function. The $\mathcal{C}$-operation and call/cc only differ in one point: call/cc implicitly invokes the current continuation on the value of its argument; $\mathcal{C}$ leaves this to its argument. $\mathcal{C}$ is equivalent to call/cc and $\mathcal{A}$ and vice versa:

$$(\mathcal{C}M) \equiv (\text{call/cc}(\lambda k.\,\mathcal{A}(Mk))) \quad \text{and} \quad (\text{call/cc}M) \equiv (\mathcal{C}\lambda k.\,k(Mk)).$$

The formal semantics of Λ_c is defined via an abstract machine. We started from an operational interpretation of a denotational semantics in the style of Reynolds's continuations-as-data structures interpreter [16]. Such a machine has three-state components: terms as control strings, environments for the evaluation of free variables, and continuation structures to remember the rest of a computation. The elimination of environments by quasi-substitution and the replacement of continuation codes by a special kind of context leads to an equivalent quasi term-rewriting system [2]. Since this rewriting system is more appropriate for the derivation of a calculus, we take it as the basis of our development.

Definition 2.2 (*the C-rewriting system*). The term language Λ_p:

$$M ::= x \mid \langle p, C[\;]\rangle \mid \lambda x.\,M \mid MN \mid \mathcal{C}M \mid \mathcal{A}M,$$

$$C[\;] ::= [\;] \mid VC[\;] \mid C[\;]M.$$

The C-transition function:

$$C[(\lambda x.\,M)V] \mapsto^C C[M[x := V]], \tag{C1}$$

$$C[\mathcal{C}M] \mapsto^C M\langle p, C[\;]\rangle, \tag{C2}$$

$$C[\langle p, C_0[\;]\rangle V] \mapsto^C C_0[V], \tag{C3}$$

$$C[\mathcal{A}M] \mapsto^C M. \tag{C4}$$

The underlying term language Λ_p—see Definition 2.2—of the C-rewriting machine is an extension of Λ_c with a new set of values: continuation points. A *continuation point* is a p-tagged applicative context. An applicative context is a term with a hole in it such that the path from the root of the term to the hole leads through applications only, and every subterm to the immediate left of the path is a value. We use $C[\;], \ldots$ to range over applicative contexts, $C[M]$ to denote a term which is like the context $C[\;]$ but with M put into the hole. Because the hole of an applicative context is never within an abstraction, the filling-in of a hole cannot capture free variables.

The transition function of the C-writing system performs a single evaluation step on an entire program. The unique partitioning of a program into an applicative context and a C-redex determines the next rewriting step. This completely orders evaluations on the C-rewriting machine; it implies that the evaluation of an application proceeds from left to right. The applicative context of a C-rewriting step also represents the current continuation. For the evaluation of a $\mathcal{C}$-application the context is packaged into a continuation point, for an $\mathcal{A}$-application it is thrown away. The invocation of a continuation point installs the associated control context, forgetting the current one. The evaluation function for programs is the transitive closure of the transition function:

$$\mathrm{eval}_C(M) = N \text{ iff } M \mapsto^{C*} N \text{ such that } N \text{ is a value.}$$

From the transition rules we can deduce that eval_C is a partial function on programs: it either yields a value or it diverges. If eval_C yields a value for a program, we say that M has a value; we also generalize this to open expressions. The result is not necessarily in Λ_c since continuation points are a new kind of value; if it is in Λ_c, we call it continuation-free.

The C-rewriting system defines an equivalence relation over Λ_c. Intuitively two terms are operationally equivalent if one can replace the other in any program and the two resulting programs are indistinguishable with respect to eval_C. This notion is due to Morris [13]; Plotkin [14] introduces the name "operational equivalence." Its formalization depends on two concepts: program contexts and basic constants.

A *program context* is an arbitrary term with one hole. Unlike applicative contexts, it may capture free variables when filled with an open term. A set of values is referred to as a set of *basic constants* if it has a decidable equality predicate. Since Λ_c does not contain constant names, we arbitrarily pick the set of normal-form number representations in Λ, e.g., the Church- or Turing-numerals [1]. With this in place, we can formulate the following definition.

Definition 2.3. *M, $N \in \Lambda_c$ are operationally equivalent, $M \simeq_C N$, iff, for any program context $C[\ ]$ such that $C[MN]$ is closed, eval_C is undefined for both $C[M]$ and $C[N]$, or it is defined for both and if one of the programs yields a basic constant, then the value of the other is the same constant.*

With an operational semantics à la SECD-machine [9], operational equivalence is rather opaque. Verifying the equivalence of two useful program pieces is almost impossible. With the C-rewriting system this becomes simpler. The transition function only uses one extraneous concept, namely applicative contexts, and this is naturally related to terms. However, the system only accounts for entire program rewriting steps and these steps are completely ordered. There is no provision for local transformations, independent of a context, nor for a simultaneous reduction of redexes. In other words, the C-rewriting system is not quite the calculus-extension one would like for Λ_c. Before we consider this problem, we briefly illustrate programming in Λ_c and clarify its advantages.

2.3. *Programming in Λ_c*

Many traditional programming languages contain linguistic devices that can be programmed with continuation accessing operations like $\mathscr{C}$. A typical example is the function-exit facility. It permits a procedure to return a result immediately to its caller, avoiding recursive nestings. We demonstrate with a sample program how this is achieved with $\mathscr{C}$. Yet, this is only a trivial use of continuations; for more interesting programs we refer the reader to the literature [6].

First we recall some common combinators and syntactic forms [1, 9]. The function $\mathbf{I} \equiv \lambda x.\, x$ is the identity function; $\mathbf{T} \equiv \lambda xy.\, x$ and $\mathbf{F} \equiv \lambda xy.\, y$ stand for the truth values *true* and *false* respectively. Given truth values, the implementation of the call-by-value version of the branching-form (**if** BMN) is $B(\lambda d.\, M)(\lambda d.\, N)\mathbf{I}$ where d is a dummy variable. We also adopt Barendregt's numeral system for the λ_c-calculus where $\lceil n \rceil$ represents the number n, **zero?** is the 0-test predicate, and + denotes the addition function for numerals. For the example we assume that a tree is either empty or that it consists of a *left-son* tree, a number, and a *right-son* tree. The functions **mt?**, **lson**, **rson**, and **num** are the respective predicate and selector functions.

Given these definitions, consider the function Σ^* which takes a tree of numbers and returns their sum:

$$\begin{aligned}\Sigma^* \equiv \mathbf{Y}_v(\lambda s.\, \lambda t.&\\ &(\mathbf{if}(\mathbf{mt?}\ t)\lceil 0 \rceil\\ &\quad (+(\mathbf{num}\ t)(+(s(\mathbf{lson}\ t))(s(\mathbf{rson}\ t)))))),\end{aligned}$$

where $\mathbf{Y}_v \equiv \lambda f.\, (\lambda x.\, f(\lambda z.\, xxz))(\lambda x.\, f(\lambda z. xxz))$ represents the call-by-value recursion combinator [17]. Designing a purely functional program that immediately returns $\lceil 0 \rceil$ upon encountering a tree element $\lceil 0 \rceil$ is less trivial although this is only a minor modification to the original specification. With $\mathscr{C}$, this new function Σ_0^* is a straightforward extension of the Σ^*-function:

$$\begin{aligned}\Sigma_0^* \equiv \lambda t.\, \mathscr{C}\lambda k.\, k(\mathbf{Y}_v(\lambda s.\, \lambda t.&\\ &(\mathbf{if}(\mathbf{mt?}\ t)\lceil 0 \rceil\\ &\quad (\mathbf{if}(\mathbf{zero?}(\mathbf{num}\ t))(k\lceil 0 \rceil)\\ &\qquad (+(\mathbf{num}\ t)(+(s(\mathbf{lson}\ t))(s(\mathbf{rson}\ t)))))))\\ &t).\end{aligned}$$

For a comparison we give an *intensionally equivalent* definition of this function in the λ-calculus. By this we mean that the following function is like Σ_0^* in that it performs a single-pass over the tree, adds numbers only if necessary and escapes as soon as a $\lceil 0 \rceil$ is discovered[4]:

$$\begin{aligned}\lambda - \Sigma_0^* \equiv \lambda t.\, (\mathbf{Y}_v(\lambda s.\, \lambda tk.\, &(\mathbf{if}(\mathbf{mt?}\ t)(k\lceil 0 \rceil)\\ &\quad(\mathbf{if}(\mathbf{zero?}(\mathbf{num}\ t))\lceil 0 \rceil\\ &\qquad(s(\mathbf{lson}\ t)\\ &\qquad\quad(\lambda l.\, (s(\mathbf{rson}\ t)(\lambda r.\, (k(+(\mathbf{num}\ t)(+lr))))))))))\\ t\mathbf{I}).&\end{aligned}$$

The internal loop of this program simultaneously passes around the current tree and a function that can perform the rest of the $\lambda - \Sigma_0^*$-computation. Initially, this simulated continuation is the identity function, indicating that $\lambda - \Sigma_0^*$ need only return the result. At every junction of two subtrees, $\lambda - \Sigma_0^*$ builds two continuations: $(\lambda l.\, (s(\mathbf{rson}\ t)\ldots))$, which represents the application of the function to the right subtree, and $\lambda r.\, k\ldots$, which combines the two partial results, passing them on to

[4] There are alternatives to this version but they are all derivable from this function [22].

another continuation. When the function encounters an empty tree, it applies the simulated continuation to the intermediate result 0; when a 0 is discovered, the continuation function is thrown away so that no more tree nodes are visited.

The function $\lambda-\Sigma_0^*$ is an optimized instance of a more general programming pattern: the continuation-passing style alluded to in the introduction. All Λ-programs can be restructured into this style via the following well-known transformation (cps) [4, 14]:

$$[\![x]\!] = \lambda\kappa\,.\,\kappa x, \tag{cps1}$$

$$[\![(\lambda x\,.\,M)]\!] = \lambda\kappa\,.\,\kappa(\lambda x\,.\,[\![M]\!]), \tag{cps2}$$

$$[\![(MN)]\!] = \lambda\kappa\,.\,[\![M]\!](\lambda m\,.\,[\![N]\!](\lambda n\,.\,mn\kappa)). \tag{cps3}$$

To evaluate a cps'ed program, it must be applied to the initial continuation **I**. If the result of a program $M \in \Lambda$ is $\lambda x\,.\,N$, then $[\![M]\!]\mathbf{I}$ yields $\lambda x\,.\,[\![N]\!]$ [14]. For Λ_c we must add two clauses:

$$[\![(\mathscr{C}M)]\!] = \lambda\kappa\,.\,[\![M]\!](\lambda m\,.\,m(\lambda v\kappa'\,.\,\kappa v)\mathbf{I}), \tag{cps4}$$

$$[\![(\mathscr{A}M)]\!] = \lambda\kappa\,.\,[\![M]\!]\mathbf{I}. \tag{cps5}$$

These equations reflect the formal definition of $\mathscr{C}$ and $\mathscr{A}$: $\mathscr{C}$ transfers control over the current continuation to the program, continuing the evaluation as if it had just begun; $\mathscr{A}$ throws away the current continuation.

The existence of the $[\![\cdot]\!]$-morphism shows that $\mathscr{C}$- and $\mathscr{A}$-applications abstract from recurring programming patterns in a functional language. Theoretically, one can write programs in Λ_c and translate them into Λ for further manipulations [11], but this would defeat the purpose of abstraction. The correct solution is to extend the λ_v-calculus with axioms for these semantic abstractions. The need for these additional axioms is documented by the following two reformulations of (cps4) and (cps5):

$$[\![\mathscr{C}M]\!] = \lambda\kappa\,.\,(\lambda\kappa'\,.\,\kappa'\underline{(\lambda f\kappa\,.\,(f(\lambda v\kappa'\,.\,(\kappa v))\mathbf{I}))})(\lambda f\,.\,[\![M]\!](\lambda m\,.\,fm\kappa)), \tag{cps4'}$$

$$[\![\mathscr{A}M]\!] = \lambda\kappa\,.\,(\lambda\kappa'\,.\,\kappa'\underline{(\lambda v\kappa\,.\,v)})(\lambda f\,.\,[\![M]\!](\lambda m\,.\,fm\kappa)). \tag{cps5'}$$

The two equations treat $\mathscr{C}$- and $\mathscr{A}$-applications as if they were applications and as if $\mathscr{C}$ and $\mathscr{A}$ were ordinary abstractions in Λ_c that map to the underlined parts respectively. On the other hand, neither $\mathscr{C}$ nor $\mathscr{A}$ are the images of values in Λ_c and therefore, ordinary β-reduction cannot support reasoning with $\mathscr{C}$- or $\mathscr{A}$-applications. New axioms must be developed.

3. The λ_c-calculus

The traditional λ-calculus may 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 a computational viewpoint the reduction system is more attractive since it exposes the rule character of the calculus and its operational nature. Thus, it is quite natural when we go the inverse direction in this section, taking the specification of the C-rewriting rules as the point of departure and deriving the reduction system.

Once the calculus is derived, the next step is to investigate its fundamental properties. Among these, consistency and standardization are the most important. The former means that equations in the calculus make sense, the latter implies that the calculus defines a programming language. Together, the two theorems provide a basis to tackle a correctness proof for the calculus.

3.1. Reductions and computations

A transition step in the C-rewriting system depends on partitioning the program into a C-redex and an applicative context. For the (C1)-step this dependency is superficial, for the others it is inherent because of the sequential nature of continuations. To establish a calculus, we must try to eliminate the context dependency as far as possible. In the case of (C1), this is trivial and yields the β_v-relation:

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

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

Next we consider $\mathscr{A}$-applications. According to (C4), an $\mathscr{A}$-application removes its applicative context. Case analysis of applicative contexts leads to appropriate *notions of reduction.* If an $\mathscr{A}$-application $\mathscr{A}M$ is within an applicative 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 $\mathscr{A}M$ in the hole. Thus, $C[(\mathscr{A}M)N]$ should be related to $C[\mathscr{A}M]$. Since this is independent of the applicative context, we can formulate our first notion of reduction for $\mathscr{A}$-applications:

$$(\mathscr{A}M)N \to^{\mathscr{A}_L} \mathscr{A}M. \qquad (\mathscr{A}_L)$$

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

$$M(\mathscr{A}N) \to^{\mathscr{A}_R} \mathscr{A}N \quad \text{provided that } M \text{ is a value.} \qquad (\mathscr{A}_R)$$

This covers all but the *base* case of applicative contexts.

The empty context requires special treatment. An occurrence of $\mathscr{A}M$ 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 $\mathscr{A}$-application is not embedded in a term. Otherwise the reduction system becomes inconsistent. Consider the expression $(\mathscr{A}\mathbf{F})\mathbf{T}\mathbf{T}$. Applying the $\mathscr{A}_L$-step twice results in $\mathscr{A}\mathbf{F}$; the top-level rule then leads to $\mathbf{F}$. When the top-level relation is first applied to the embedded $\mathscr{A}$-application, $(\mathscr{A}\mathbf{F})\mathbf{T}\mathbf{T}$ goes to $\mathbf{F}\mathbf{T}\mathbf{T}$ which in turn results in $\mathbf{T}$. $\mathbf{T}$ would thus equal $\mathbf{F}$ and this is inconsistent. We therefore introduce this top-level relation as a *computation rule* and denote it with a $\triangleright$ instead of the customary $\to$:

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

When we complete the calculus later, we must add this computation rule at the right place.

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

For the correct design of X we appeal to the intended semantics of the $\mathscr{C}$-application. The $\mathscr{C}$-application must capture the current continuation and supply it to its argument. Hence, if X is the next $\mathscr{C}$-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 $\mathscr{C}$-argument M. Furthermore, M's context 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 $\mathscr{C}M$ could be characterized by $\kappa(fN)$ where κ stands for the continuation of $\mathscr{C}X$. 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 on to the term $\lambda f.\,\kappa(fN)$. A first approximation of X is $\lambda\kappa.\,M(\lambda f.\,\kappa(fN))$. This satisfies (C2) since it removes the context of a $\mathscr{C}$-application and applies its argument to some encoding of the context, but continuation points also need to respect (C3).

The rewriting rule (C3) demands that when a continuation is invoked, the current context is removed. This means for Λ_c-continuations that the first action must be an *abort* action to remove the current context. Hence, $\lambda f.\,\mathscr{A}(\kappa(fN))$ is the correct continuation for M. The symmetric case where $\mathscr{C}N$ 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 $\mathscr{C}$-application:

$$(\mathscr{C}M)N \rightarrow^{\mathscr{C}_L} \mathscr{C}\lambda\kappa.\,M(\lambda f.\,\mathscr{A}(\kappa(fN))), \qquad (\mathscr{C}_L)$$

$$M(\mathscr{C}N) \rightarrow^{\mathscr{C}_R} \mathscr{C}\lambda\kappa.\,N(\lambda v.\,\mathscr{A}(\kappa(Mv))) \quad \text{provided that } M \text{ is a value.} \qquad (\mathscr{C}_R)$$

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

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

With this last rule we have derived the reduction and computation rules that are intuitively needed to simulate the C-transition function, but defining notions of reduction is only the first step. 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, $\mathscr{C}$-application, and $\mathscr{A}$-application. The two computation rules cannot be included in this relation since they are not applicable to nested subterms. Definition 3.1 contains a formal description of the one-step reduction relation $\to_c$.

Definition 3.1 (*the λ_c-calculus*). Let $\to^c = \to^{\mathscr{C}_L} \cup \to^{\mathscr{C}_R} \cup \to^{\mathscr{A}_L} \cup \to^{\mathscr{A}_R} \cup \to^{\beta_v}$. The *one-step C-reduction* $\to_c$ is the compatible closure of $\to^c$:

$$\begin{aligned}
M \to^c N &\Rightarrow M \to_c N;\\
M \to_c N &\Rightarrow \lambda x.M \to_c \lambda x.N;\\
M \to_c N &\Rightarrow ZM \to_c ZN, MZ \to_c NZ \quad \text{for } Z \in A_c;\\
M \to_c N &\Rightarrow \mathscr{C}M \to_c \mathscr{C}N;\\
M \to_c N &\Rightarrow \mathscr{A}M \to_c \mathscr{A}N.
\end{aligned}$$

The *C-reduction* is denoted by $\twoheadrightarrow_c$ and is the transitive-reflexive closure of $\to_c$. We denote the smallest congruence relation generated by $\twoheadrightarrow_c$ with $=_c$ and call it *C-equality.*

The *C-computation* $\rhd_c$ is defined by: $\rhd_c = \rhd_{\mathscr{C}} \cup \rhd_{\mathscr{A}} \cup \twoheadrightarrow_c$. The relation $=_c^{\rhd}$ is the smallest equivalence relation generated by $\rhd_c$. We refer to it as *computational C-equality.*

The final step in the development of a calculus is the construction of a congruence relation from the reduction relation, i.e., an equivalence relation which respects the syntactic constructors. Conforming to tradition, we do this in two states: $\twoheadrightarrow_c$ is the transitive-reflexive closure of $\to_c$; 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 computation function which simulates the machine evaluation: occurrences of $\mathscr{C}$- and $\mathscr{A}$-applications at the root of a term cannot be removed. We extend the reduction relation $\twoheadrightarrow_c$ to a computation relation $\rhd_c$ by adding the top-level relations. Forming the symmetric, reflexive, and transitive closure of $\rhd_c$ results in an equivalence relation $=_c^{\rhd}$ which establishes equality among terms according to reductions *and* computations. All these concepts are summarized in Definition 3.1.

The relation $=_c^{\rhd}$ determines the λ_c-calculus and we write $\lambda_c \vdash M =_c^{\rhd} N$ if the terms M and N are equal under $=_c^{\rhd}$. 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.

3.2. Fundamental properties of the λ_c-calculus

The development of the basic notions of the λ_c-calculus raises a number of interesting questions. In principle, it makes sense to inspect every λ-calculus-theorem on its validity for the extended calculus. As mentioned at the outset of this section, we concentrate on the question of consistency and the existence of standard reduction and standard computation sequences. The central results of this subsection are captured in the Consistency Theorems 3.7 and 3.8, the definition of a standard computation function (Definition 3.9), and the Standardization Theorem 3.10; the reader may wish to concentrate on these points and to skip over the proof details for the first reading.

The consistency problem depends on proving the confluence of reduction and computation paths that start from the same place. In other words, we must prove a classical Church-Rosser theorem for reductions and a diamond theorem for computation. The proof of the Church-Rosser property for $\rightarrow^c$ is an application of Martin-Löf's method for showing the corresponding result for $\rightarrow^\beta$. Since our presentation follows Barendregt's rather closely, we only state the necessary lemmas and demonstrate some of the major modifications to the proofs.

First, we define a version of the parallel reduction relation $\twoheadrightarrow_l$ for $\rightarrow^c$. For the proof of the standardization theorem we also define a notion of the length or size of the parallel reduction: see Definition 3.2. Note that if M is a value and $M \twoheadrightarrow_l N$, then N is a value.

Definition 3.2 (*the parallel reduction* $\twoheadrightarrow_l$). The *parallel reduction* over Λ_c is denoted by $\twoheadrightarrow_l$. $s_{M \twoheadrightarrow_l N}$ or just s is the function which measures the *size of the derivation* $M \twoheadrightarrow_l N$. $n(x, M)$ is the number of free occurrences of x in M.

(P1) $M \twoheadrightarrow_l M, \quad s = 0;$

(P2) $M \twoheadrightarrow_l M', \quad N \twoheadrightarrow_l N',$
N is a value $\Rightarrow (\lambda x.\, M)N \twoheadrightarrow_l M'[x := N'],$
$s = s_{M \twoheadrightarrow_l M'} + n(x, M')s_{N \twoheadrightarrow_l N'} + 1;$

(P3) $M \twoheadrightarrow_l M' \Rightarrow (\mathscr{A}M)N \twoheadrightarrow_l (\mathscr{A}M'),$
$s = s_{M \twoheadrightarrow_l M'} + 1;$

(P4) $N \twoheadrightarrow_l N', M$ is a value $\Rightarrow M(\mathscr{A}N) \twoheadrightarrow_l \mathscr{A}N',$
$s = s_{N \twoheadrightarrow_l N'} + 1;$

(P5) $M \twoheadrightarrow_l M', N \twoheadrightarrow_l N' \Rightarrow (\mathscr{C}M)N \twoheadrightarrow_l \mathscr{C}\lambda\kappa.\, M'(\lambda f.\, \mathscr{A}(\kappa(fN'))),$
$s = s_{M \twoheadrightarrow_l M'} + s_{N \twoheadrightarrow_l N'} + 1;$

(P6) $M \twoheadrightarrow_l M', N \twoheadrightarrow_l N',$
M is a value $\Rightarrow M(\mathscr{C}N) \twoheadrightarrow_l \mathscr{C}\lambda\kappa.\, N'(\lambda v.\, \mathscr{A}(\kappa(M'v))),$
$s = s_{M \twoheadrightarrow_l M'} + s_{N \twoheadrightarrow_l N'} + 1;$

(P7) $M \twoheadrightarrow_l N \Rightarrow \lambda x.\, M \twoheadrightarrow_l \lambda x.\, N,$
$s = s_{M \twoheadrightarrow_l N};$

(P8) $M \twoheadrightarrow_l N \Rightarrow \mathscr{C}M \twoheadrightarrow_l \mathscr{C}N,$
$s = s_{M \twoheadrightarrow_l N};$

(P9) $M \twoheadrightarrow_l N \Rightarrow \mathscr{A}M \twoheadrightarrow_l \mathscr{A}N,$
$s = s_{M \twoheadrightarrow_l N};$

(P10) $M \twoheadrightarrow_l M', N \twoheadrightarrow_l N' \Rightarrow MN \twoheadrightarrow_l M'N',$
$s = s_{M \twoheadrightarrow_l M'} + s_{N \twoheadrightarrow_l N'}.$

The following lemma shows the relationship between $\twoheadrightarrow_c$ and $\twoheadrightarrow_l$. Its proof is obvious and omitted.

Lemma 3.3. $\rightarrow^c \subset \rightarrow_c \subset \twoheadrightarrow_l \subset \twoheadrightarrow_c$.

Next we prove that in $\twoheadrightarrow_l$, unlike in $\rightarrow_c$, the expression $M[x := N]$ reduces to $M'[x := N']$ in one step if M and N reduce to M' and N' in one step respectively. However, for the proof of the standardization theorem we also need to know that this reduction is shorter than the one from $(\lambda x. M)N$ to $M'[x := N']$. The two proofs have the same structure and therefore—following Plotkin—we merge them.

Lemma 3.4. *Suppose $M \twoheadrightarrow_l M'$, $N \twoheadrightarrow_l N'$, and N is a value. Then the following statements hold:*

(i) $M[x := N] \twoheadrightarrow_l M'[x := N']$;

(ii) $s_{\mathrm{R}} = s_{M[x := N] \twoheadrightarrow_l M'[x := N']} < s_{\mathrm{L}} = s_{(\lambda x. M)N \twoheadrightarrow_l M'[x := N']}$.

Proof. The proof is a structural induction on the reduction $M \twoheadrightarrow_l M'$. We omit the cases which are similar to the given ones.

(P1): $M \twoheadrightarrow_l M' \equiv M$. The result follows by induction on the structure of M. We demonstrate it for the subcase $M \equiv \mathscr{C}P \equiv \mathscr{C}P' \equiv M'$.

(i) $(\mathscr{C}P)[x := N] \equiv (\mathscr{C}P[x := N]) \twoheadrightarrow_l (\mathscr{C}P'[x := N'])$ since P is smaller than M.

(ii)
$$\begin{aligned} s_R &= s_{P[x := N] \twoheadrightarrow_l P'[x := N']} \\ &\leq s_{P \twoheadrightarrow_l P'} + n(x, P') s_{N \twoheadrightarrow_l N'} && \text{by inductive hypothesis} \\ &< n(x, P) s_{N \twoheadrightarrow_l N'} + 1 = s_{\mathrm{L}} && \text{since } P' \equiv P. \end{aligned}$$

(P3): $M \equiv (\mathscr{A}P)Q \twoheadrightarrow_l M' \equiv \mathscr{A}P'$ and $P \twoheadrightarrow_l P'$.

(i) $((\mathscr{A}P)Q)[x := N] \equiv (\mathscr{A}P[x := N])Q[x := N] \twoheadrightarrow_l \mathscr{A}P'[x := N']$ by inductive hypothesis for $P[x := N] \twoheadrightarrow_l P'[x := N']$.

(ii)
$$\begin{aligned} s_{\mathrm{R}} &= s_{P[x := N] \twoheadrightarrow_l P'[x := N']} + 1 \\ &\leq s_{P \twoheadrightarrow_l P'} + n(x, P') s_{N \twoheadrightarrow_l N'} + 1 && \text{by inductive hypothesis} \\ &< (s_{P \twoheadrightarrow_l P'} + 1) + n(x, P') s_{N \twoheadrightarrow_l N'} + 1 = s_{\mathrm{L}}. \end{aligned}$$

(P6): $M \equiv P(\mathscr{C}Q) \twoheadrightarrow_l M' \equiv \mathscr{C}\lambda\kappa. Q'(\lambda v. \mathscr{A}(\kappa(P'v)))$ and $P \twoheadrightarrow_l P'$, $Q \twoheadrightarrow_l Q'$, and P is a value.

(i)
$$\begin{aligned} (P(\mathscr{C}Q))[x := N] &\equiv P[x := N](\mathscr{C}Q[x := N]) \\ &\twoheadrightarrow_l \mathscr{C}\lambda\kappa. Q'[x := N'](\lambda v. \mathscr{A}(\kappa(P'[x := N']v))) \end{aligned}$$

by inductive hypotheses for P and Q and the fact that $P[x := N]$ is a value.

(ii) $s_{\mathrm{R}} = s_{P[x:=N](\mathscr{C}Q[x:=N]) \twoheadrightarrow_l \mathscr{C}\lambda\kappa.Q'[x:=N'](\lambda v.\mathscr{A}(\kappa(P'[x:=N']v)))}$

$$= s_{P[x:=n] \twoheadrightarrow_l P'[x:=N']} + s_{Q[x:=N] \twoheadrightarrow_l Q'[x:=N']} + 1$$

$$\leqslant s_{P \twoheadrightarrow_l P'} + n(x, P')s_{N \twoheadrightarrow_l N'} + s_{Q \twoheadrightarrow_l Q'} + n(x, Q')s_{N \twoheadrightarrow_l N'} + 1$$

by inductive hypothesis for $s_{P[x:=N] \twoheadrightarrow_l P'[x:=N']}$ and $s_{Q[x:=N] \twoheadrightarrow_l Q'[x:=N']}$

$$< (s_{P \twoheadrightarrow_l P'} + s_{Q \twoheadrightarrow_1 Q'} + 1) + n(x, M')s_{N \twoheadrightarrow_l N'} + 1 = s_{\mathrm{L}}.$$

(P9): $M \equiv \mathscr{C}P \twoheadrightarrow_l M' \equiv \mathscr{C}P'$ and $P \twoheadrightarrow_l P'$.

(i) $(\mathscr{C}P)[x := N] \equiv \mathscr{C}P[x := N] \twoheadrightarrow_l \mathscr{C}P'[x := N']$ by inductive hypothesis.

(ii) $s_{\mathrm{R}} = s_{P[x:=N] \twoheadrightarrow_l P'[x:=N']}$

$$\leqslant s_{P \twoheadrightarrow_l P'} + n(x, P')s_{N \twoheadrightarrow_l N'} \quad \text{by inductive hypothesis}$$

$$< s_{\mathrm{L}}. \quad \square$$

In addition to Lemma 3.4 we must show that two contractums of $\mathscr{C}_{\mathrm{L}}$- or $\mathscr{C}_{\mathrm{R}}$-redexes reduce to each other in one $\twoheadrightarrow_l$-step if the respective subterms do. Again, the second and fourth claim of the following lemma are actually needed for the standardization theorem.

Lemma 3.5. *Suppose $M \twoheadrightarrow_l M'$ and $N \twoheadrightarrow_l N'$. Then the following statements hold:*

(i) $\mathscr{C}\lambda\kappa\,.\,M(\lambda f.\,\mathscr{A}(\kappa(fN))) \twoheadrightarrow_l \mathscr{C}\lambda\kappa\,.\,M'(\lambda f.\,\mathscr{A}(\kappa(fN')))$

(ii) $s_{\mathrm{R}} < s_{\mathrm{L}}$ *where*

$$s_{\mathrm{R}} = s_{\mathscr{C}\lambda\kappa.M(\lambda f.\mathscr{A}(\kappa(fN))) \twoheadrightarrow_l \mathscr{C}\lambda\kappa.M'(\lambda f.\mathscr{A}(\kappa(fN')))},$$

$$s_{\mathrm{L}} = s_{(\mathscr{C}M)N \twoheadrightarrow_l \mathscr{C}\lambda\kappa.M'(\lambda f.\mathscr{A}(\kappa(fN')))}.$$

And if M is a value, then

(iii) $\mathscr{C}\lambda\kappa\,.\,N(\lambda v\,.\,\mathscr{A}(\kappa(Mv))) \twoheadrightarrow_l \mathscr{C}\lambda\kappa\,.\,N'(\lambda v\,.\,\mathscr{A}(\kappa(M'v)))$

(iv) $s_{\mathrm{R}} < s_{\mathrm{L}}$ *where*

$$s_{\mathrm{R}} = s_{\mathscr{C}\lambda\kappa.N(\lambda v.\mathscr{A}(\kappa(Mv))) \twoheadrightarrow_l \mathscr{C}\lambda\kappa.N'(\lambda v.\mathscr{A}(\kappa(M'v)))},$$

$$s_{\mathrm{L}} = s_{M(\mathscr{C}N) \twoheadrightarrow_l \mathscr{C}\lambda\kappa.N'(\lambda v.\mathscr{A}(\kappa(M'v)))}.$$

Proof. We show (i) and (ii) by straightforward calculations:

(i) $N \twoheadrightarrow_l N'$, hence $\lambda f.\,\mathscr{A}(\kappa(fN)) \twoheadrightarrow_l \lambda f.\,\mathscr{A}(\kappa(fN'))$; further $M \twoheadrightarrow_l M'$, hence $M(\lambda f.\,\mathscr{A}(\kappa(fN))) \twoheadrightarrow_l M'(\lambda f.\,\mathscr{A}(\kappa(fN')))$, and

$$\lambda\kappa\,.\,(M(\lambda f.\,\mathscr{A}(\kappa(fN)))) \twoheadrightarrow_l \lambda\kappa\,.\,(M'(\lambda f.\,\mathscr{A}(\kappa(fN')))),$$

and therefore, $\mathscr{C}\lambda\kappa\,.\,M(\lambda f.\,\mathscr{A}(\kappa(fN))) \twoheadrightarrow_l \mathscr{C}\lambda\kappa\,.\,M'(\lambda f.\,\mathscr{A}(\kappa(fN')))$.

(ii) $s_{\mathrm{R}} = s_{M \twoheadrightarrow_l M'} + s_{N \twoheadrightarrow_l N'} < s_{M \twoheadrightarrow_l M'} + s_{N \twoheadrightarrow_l N'} + 1 = s_{\mathrm{L}}$.

The proofs of propositions (iii) and (iv) follow the same pattern. $\square$

Everything is in place to state and prove the diamond lemma for the parallel reduction:

Lemma 3.6. *The relation $\twoheadrightarrow_I$ satisfies the diamond property, i.e., if $M \twoheadrightarrow_I L_i$, then there exists an N such that $L_i \twoheadrightarrow_I N$ for $i = 1, 2$.*

Proof. Again, the proof is an induction on the structure of the reduction $M \twoheadrightarrow_I L_1$. We only discuss two cases. The rest of the possible cases are either similar to some of the presented ones or can be found in Barendregt's corresponding proof.

(P6): $M \equiv P(\mathscr{C}Q) \twoheadrightarrow_I L_1 \equiv \mathscr{C}\lambda\kappa . Q_1(\lambda v . \mathscr{A}(\kappa(P_1 v)))$ and $P \twoheadrightarrow_I P_1$, $Q \twoheadrightarrow_I Q_1$, and P is a value. There are two possible cases for the reduction from M to L_2 since P is a value and $\mathscr{C}Q$ is not:

(a) $L_2 \equiv P_2(\mathscr{C}Q_2)$ and $P \twoheadrightarrow_I P_2$, $Q \twoheadrightarrow_I Q_2$. Note that P_2 is a value. An application of Lemma 3.5 yields $N \equiv \mathscr{C}\lambda\kappa . Q_3(\lambda v . \mathscr{A}(\kappa(P_3 v)))$ where P_3 and Q_3 are the terms which must exist for $P \twoheadrightarrow_I P_i$ and $Q \twoheadrightarrow_I Q_i$ for $i = 1, 2$ according to the inductive hypothesis.

(b) $L_2 \equiv \mathscr{C}\lambda\kappa . Q_2(\lambda v . \mathscr{A}(\kappa(P_2 v)))$ and $P \twoheadrightarrow_I P_2$, $Q \twoheadrightarrow_I Q_2$. Again, an application of Lemma 3.5 and of the inductive hypothesis for $P \twoheadrightarrow_I P_i$, $Q \twoheadrightarrow_I Q_i$, $i = 1, 2$ produces terms P_3, Q_3 such that $N \equiv \mathscr{C}\lambda\kappa . Q_3(\lambda v . \mathscr{A}(\kappa(P_3 v)))$.

(P10): $M \equiv PQ \twoheadrightarrow_I L_1 \equiv P_1 Q_1$ and $P \twoheadrightarrow_I P_1$, $Q \twoheadrightarrow_I Q_1$. This time we have to distinguish six possible subcases:

(a) $L_2 \equiv R_2[x := Q_2]$ and $P \equiv \lambda x . R$, $R \twoheadrightarrow_I R_2$, $Q \twoheadrightarrow_I Q_2$, and Q is a value. But then $P_1 \equiv \lambda x . R_1$, $R \twoheadrightarrow_I R_1$, and Q_1 is a value. By inductive hypothesis we must be able to find R_3 and Q_3 such that, with Lemma 3.4, $N \equiv R_3[x := Q_3]$.

(b) $L_2 \equiv \mathscr{A}R_2$ and $P \equiv \mathscr{A}R$, $R \twoheadrightarrow_I R_2$. Again, $P_1 \equiv \mathscr{A}R_1$ and $R \twoheadrightarrow_I R_1$. By inductive hypothesis we can find an R_3 such that $N \equiv \mathscr{A}R_3$.

(c) $L_2 \equiv \mathscr{A}R_2$ and $Q \equiv \mathscr{A}R$, $R \twoheadrightarrow_I R_2$, and P is a value. This case is like (b).

(d) $L_2 \equiv \mathscr{C}\lambda\kappa . R_2(\lambda f . \mathscr{A}(\kappa(fQ_2)))$ and $P \equiv \mathscr{C}R$, $Q \twoheadrightarrow_I Q_2$, $R \twoheadrightarrow_I R_2$. This implies that $P_1 \equiv \mathscr{C}R_1$, $R \twoheadrightarrow_I R_1$. An application of the inductive hypothesis and Lemma 3.5 shows that $N \equiv \mathscr{C}\lambda\kappa . R_3(\lambda f . \mathscr{A}(\kappa(fQ_3)))$.

(e) $L_2 \equiv \mathscr{C}\lambda\kappa . R_2(\lambda v . \mathscr{A}(\kappa(P_2 v)))$ and $Q \equiv \mathscr{C}R$, $P \twoheadrightarrow_I P_2$, $R \twoheadrightarrow_I R_2$, and P is a value. This case is like (d).

(f) $L_2 \equiv P_2 Q_2$. Trivial. □

Putting things together we get the Church-Rosser property for $\rightarrow^c$ as follows.

Theorem 3.7. *The relation $\rightarrow^c$ is Church-Rosser.*

Proof. $\twoheadrightarrow_c$ is the transitive closure of $\twoheadrightarrow_I$. Since $\twoheadrightarrow_I$ satisfies the diamond property so does $\twoheadrightarrow_c$. □

An alternative proof of the above theorem is based on the Hindley-Rosen method for showing that the $\beta\eta$-reduction is CR. This requires checking that each reduction is CR and that they commute with each other. In our case the second part would be laborious since five (!) different rules are involved. The above proof also has the advantage that it neatly ties in with the proof of the standardization theorem in the second half of this section.

Based on Theorem 3.7 we can show that $\triangleright_c$ satisfies the diamond property which is sufficient to establish consistency.

Theorem 3.8 (Consistency). *The relation $\triangleright_c$ satisfies the diamond property.*

Proof. Assuming that $M \triangleright_c L_i$ for $i = 1, 2$, we need to show that there exists a term N such that $L_i \triangleright_c N$. We proceed by a case analysis on $M \triangleright_c L_1$.

($\mathcal{A}_T$): $M \triangleright_{\mathcal{A}} L_1$ and $M \equiv (\mathcal{A}L_1)$. Then there are only two possible cases for the step from M to L_2:

(a) $M \twoheadrightarrow_c (\mathcal{A}K_2)$. But then $L_1 \twoheadrightarrow_c K_2$ and we can take $N \equiv K_2$.

(b) $M \triangleright_{\mathcal{A}} L_2$. Trivially, $M \equiv (\mathcal{A}L_1) \equiv (\mathcal{A}L_2)$ and $N \equiv L_1 \equiv L_2$.

($\mathcal{C}_T$): $M \triangleright_{\mathcal{C}} L_1$ and $M \equiv (\mathcal{C}L_1)$. This case is just like ($\mathcal{A}_T$).

($\twoheadrightarrow_c$): $M \twoheadrightarrow_c L_1$. Three cases are possible. Two of them are symmetric to the previous ones. The third one is $M \twoheadrightarrow_c L_2$, but then we just apply the Church-Rosser Theorem for $\rightarrow^c$. □

The theorem establishes the following traditional corollary.

Corollary. *If $M \overset{\triangleright}{=}_c N$, then there exists an L such that $M \triangleright_c^* L$* and *$N \triangleright_c^* L$.*

With the Church-Rosser theorem in place, we can tackle the standardization theorem. A standard reduction sequence for the λ-calculus is usually defined with respect to the position of redexes within a term and their residuals. Plotkin gives an equivalent, but more elegant and intuitive definition. It requires the notion of a standard reduction function which reduces the first—top-down and left-to-right—redex in a Λ-term not inside an abstraction. This function is then extended to standard reduction sequences by forming something like a compatible closure. The extended syntax and the computation rules in the λ_c-calculus require a slightly more complex construction. In particular, the constructions of a standard reduction function and a standard reduction sequence must proceed in two stages such that computation rules do not interfere with reductions. Nevertheless, the definitions remain intuitive and are formalized in Definition 3.9.

Definition 3.9 (*standard reduction sequences and functions*). The *standard reduction function*, denoted by $\mapsto_{sc}$, is defined by

$$M \rightarrow^c N \Rightarrow M \mapsto_{sc} N;$$

$$M \mapsto_{sc} M' \Rightarrow MN \mapsto_{sc} M'N;$$

$$M \text{ is a value, } N \mapsto_{sc} N' \Rightarrow MN \mapsto_{sc} MN'.$$

Standard reduction sequences, abbreviated C-SRS-s, are defined by

$x \in V \Rightarrow x$ is a C-SRS;

$N_1, \ldots, N_k$ is a C-SRS

$\Rightarrow \lambda x . N_1, \ldots, \lambda x . N_k, \mathscr{C}N_1, \ldots, \mathscr{C}N_k$, and

$\mathscr{A}N_1, \ldots, \mathscr{A}N_k$ are C-SRS's;

$M \mapsto_{\text{sc}} N_1$, and $N_1, \ldots, N_k$ is a C-SRS $\Rightarrow M, N_1, \ldots, N_k$ is a C-SRS

$M_1, \ldots, M_j$ and $N_1, \ldots, N_k$ are C-SRS's

$\Rightarrow M_1N_1, \ldots, M_jN_1, \ldots, M_jN_k$ is a C-SRS.

The standard computation function for λ_c extends $\mapsto_{\text{sc}}$ to computations:

$$\overset{\rhd}{\mapsto}_{\text{sc}} = \rhd_{\mathscr{C}} \cup \rhd_{\mathscr{A}} \cup \mapsto_{\text{sc}}.$$

Standard computation sequences, $C^{\rhd}$-SCS-s, are defined by

$N_1, \ldots, N_k$ is a C-SRS $\Rightarrow N_1, \ldots, N_k$ is a $C^{\rhd}$-SCS;

$M \overset{\rhd}{\mapsto}_{\text{sc}} N_1$ and $N_1, \ldots, N_k$ is a $C^{\rhd}$-SCS $\Rightarrow M, N_1, \ldots, N_k$ is a $C^{\rhd}$-SCS.

The notation $\overset{\rhd +}{\mapsto}_{\text{sc}}$ and $\overset{\rhd *}{\mapsto}_{\text{sc}}$ stand for the transitive and transitive-reflexive closure of $\overset{\rhd}{\mapsto}_{\text{sc}}$ respectively; $\overset{\rhd i}{\mapsto}_{\text{sc}}$ indicates i applications of $\overset{\rhd}{\mapsto}_{\text{sc}}$.

The theorem which we want to prove can now be stated as follows.

Theorem 3.10 (Standardization). *$M \rhd_c^* N$ if and only if there exists a $C^{\rhd}$-SCS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N$.*

The proof is divided into two parts. First, we show that there is a standardization theorem for reductions. Second, we give a method for reshuffling $\rhd_c$-computation sequences into $C^{\rhd}$-SCS's. The method is based on the first standardization theorem and it utilizes the consequence of Theorem 3.8 that at the root of the term computations and reductions are interchangeable.

The standardization theorem for the reductions is as follows.

Theorem 3.11. *$M \twoheadrightarrow_c N$ if and only if there is a C-SRS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N$.*

Proof. The direction from right to left is trivial. For the opposite we follow Plotkin's plan for the corresponding theorem about the λ_v-calculus. First, the sequence of $\rightarrow_c$-steps is replaced by a sequence of steps using the parallel reduction $\twoheadrightarrow_l$. This follows from Lemma 3.3. Then we show with the following lemma that one can recursively transform the resulting sequence of $\twoheadrightarrow_l$ reductions into a C-SRS. □

Lemma 3.12. *If $M \twoheadrightarrow_l N_1$ and $N_1, \ldots, N_j$ is a C-SRS, then there exists a C-SRS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N_j$.*

Proof. The proof is a lexicographic induction on j, on the size of the proof $M \twoheadrightarrow_l N_1$, and on the structure of M. We proceed by case analysis on the last step in $M \twoheadrightarrow_l N_1$ and omit all the cases which are similar to the presented ones or which are treated by Plotkin:

(P3): $M \equiv (\mathcal{A}P)Q \twoheadrightarrow_l N_1 \equiv \mathcal{A}P_1$ and $P \twoheadrightarrow_l P_1$. But then we also have $M \mapsto_{sc} \mathcal{A}P$ and $\mathcal{A}P \twoheadrightarrow_l \mathcal{A}P_1$ by a proof which is shorter than the proof $M \twoheadrightarrow_l N_1$. Hence, by inductive hypothesis we find a C-SRS from $\mathcal{A}P$ to N_j and can then build the required C-SRS from M to N_j.

(P6): $M \equiv P(\mathcal{C}Q) \twoheadrightarrow_l N_1 \equiv \mathcal{C}\lambda\kappa\,.\,Q_1(\lambda v\,.\,\mathcal{A}(\kappa(P_1 v)))$ and $P \twoheadrightarrow_l P_1$, $Q \twoheadrightarrow_l Q_1$, and P is a value. Again, M can immediately be reduced to $\mathcal{C}\lambda\kappa\,.\,Q(\lambda v\,.\,\mathcal{A}(\kappa(Pv)))$ by $\mapsto_{sc}$. By Lemma 3.4 we know that

$$\mathcal{C}(\lambda\kappa\,.\,(Q(\lambda v\,.\,(\kappa(Pv))))) \twoheadrightarrow_l \mathcal{C}(\lambda\kappa\,.\,(Q_1(\lambda v\,.\,\mathcal{A}(\kappa(P_1 v)))))$$

by a proof that is shorter than the one for $M \twoheadrightarrow_l N_1$. Therefore, by inductive hypothesis, we can find a C-SRS from $\mathcal{C}\lambda\kappa\,.\,Q_1(\lambda v\,.\,\mathcal{A}(\kappa(P_1 v)))$ to N_j from which we build the required C-SRS from M to N_j.

(P9): $M \equiv \mathcal{C}P \twoheadrightarrow_l M' \equiv \mathcal{C}P'$ and $P \twoheadrightarrow_l P'$. Here $N_i \equiv \mathcal{C}N'_i$ for all i, $1 \leq i \leq j$. Now consider $P \twoheadrightarrow_l N'_1, \ldots, N'_j$. P is obviously smaller than M and we can apply the inductive hypothesis to find a C-SRS from P to N'_j. Wrapping every element of this sequence in a $\mathcal{C}$-application yields the required reduction sequence.

(P10): This case does not differ from Plotkin's corresponding case but it requires that $M \twoheadrightarrow_l N \mapsto_{sc} L$ can be transformed into $M \mapsto_{sc} K \twoheadrightarrow_l L$ for some appropriate term K. This is proven in a separate lemma. □

The next lemma shows that $\twoheadrightarrow_l$ and $\mapsto_{sc}$ commute as required by case (P10) of the preceding lemma:

Lemma 3.13. *If $M \twoheadrightarrow_l M' \mapsto_{sc} M''$, then there exists an L such that $M \mapsto_{sc}^{+} L \twoheadrightarrow_l M''$.*

Proof. Plotkin's proof of his Lemma 8, Section IV, goes through with almost no change. It is a lexicographic induction on the size of the reduction $M \twoheadrightarrow_l M'$ and of M. It is divided according to the last parallel reduction step. Cases (P1) through (P9) are routine with (P5) and (P6) relying on Lemma 3.5. The last case again induces the need for another lemma and deserves some explanation.

For case (P10) we assume that $M \equiv PQ \twoheadrightarrow_l M' \equiv P'Q'$ because $P \twoheadrightarrow_l P'$, $Q \twoheadrightarrow_l Q'$, and $M' \mapsto_{sc} M''$. We now proceed by an analysis on this standard reduction step and consider two typical subcases:

(a) $M' \equiv (\mathcal{A}P'_1)Q' \mapsto_{sc} M'' \equiv \mathcal{A}P'_1$. So we have $P \twoheadrightarrow_l \mathcal{A}P'_1$. But then we claim that there exists an L such that $P \mapsto_{sc}^{*} L \twoheadrightarrow_l \mathcal{A}P'_1$ and L is an $\mathcal{A}$-application. With the rules for $\mapsto_{sc}$ we get that $(PQ) \mapsto_{sc}^{*} (LQ) \mapsto_{sc} L \twoheadrightarrow_l \mathcal{A}P'_1 \equiv M''$.

(b) $M' \equiv (\mathscr{C}P'_1)Q' \mapsto_{\mathrm{sc}} M'' \equiv \mathscr{C}\lambda\kappa\,.\,P'_1(\lambda f\,.\,\mathscr{A}(\kappa(fQ')))$. We proceed just like in (a). Given that $P \twoheadrightarrow_l \mathscr{C}P'_1$ we again claim that there is a K such that $P \mapsto^*_{\mathrm{sc}} K \twoheadrightarrow_l \mathscr{C}P'_1$ and K is a $\mathscr{C}$-application. K and Q' form the required L in the obvious way and the rest is similar to (a). □

There are four propositions left that we have claimed or that we need through our adoption of Plotkin's proofs. All the necessary proofs are quite straightforward, but for the sake of completeness we state the lemmas.

Lemma. *If $M \twoheadrightarrow_l (\mathscr{A}N)$ where M is an application, then there exists an L which is an $\mathscr{A}$-application and $M \mapsto^+_{\mathrm{sc}} L \twoheadrightarrow_l (\mathscr{A}N)$.*

Lemma. *If $M \twoheadrightarrow_l (\mathscr{C}N)$ where M is an application, then there exists an L which is a $\mathscr{C}$-application and $M \mapsto^+_{\mathrm{sc}} L \twoheadrightarrow_l (\mathscr{C}N)$.*

Lemma. *If $M \twoheadrightarrow_l (\lambda x\,.\,N)$ where M is an application, then there exists an L which is an abstraction and $M \mapsto^+_{\mathrm{sc}} L \twoheadrightarrow_l (\lambda x\,.\,N)$.*

Lemma. *If $M \twoheadrightarrow_l x$ where x is a variable, then $M \mapsto^+_{\mathrm{sc}} x$.*

Equipped with this first standardization theorem for reductions, it is easy to finish the proof of Theorem 3.10, but first, we need to clarify one more fact.

Lemma 3.14. *If $N_1, \ldots, N_k$ is a C-SRS where $N_k \equiv \mathscr{C}L$ or $N_k \equiv \mathscr{A}L$, then there exists a j, $1 \leq j < k$, such that for all i, $1 \leq i < j$, $N_i \mapsto_{\mathrm{sc}} N_{i+1}$, and, for all i, $j \leq i < k$, $N_i \rightarrow_c N_{i+1}$ and $N_i \not\mapsto_{\mathrm{sc}} N_{i+1}$.*

Proof. A straightforward induction on k. □

And finally, here is the proof of the main result of this section.

Proof of Theorem 3.10. The proof is an induction on the number of computations, $\rhd_{\mathscr{C}}$ and $\rhd_{\mathscr{A}}$, used in the evaluation of M to N. If there are no computations involved, we can form the C-SRS for reducing M to N and we have the desired result. Now suppose there is at least one reduction of type $\rhd_{\mathscr{A}}$. Then we have the following situation:

$$M \equiv M_1 \rightarrow_c \cdots \rightarrow_c M_k \equiv \mathscr{A}M'_k \rhd_{\mathscr{A}} M_{k+1} \equiv M'_k \rhd_c \cdots \rhd_c M_n \equiv N.$$

By forming the C-SRS for the reduction from M_1 to M_k we get, by Theorem 3.11 and Lemma 3.14,

$$M \equiv M_1 \mapsto_{\mathrm{sc}} \cdots \mapsto_{\mathrm{sc}} M_l \equiv \mathscr{A}M'_l \rightarrow_c \cdots \rightarrow_c M_k \equiv \mathscr{A}M'_k$$

$$\rhd_{\mathscr{A}} M_{k+1} \equiv M'_k \rhd_c \cdots \rhd_c M_n \equiv N.$$

Since $\rightarrow_c$ and $\triangleright_{\mathscr{A}}$ are interchangeable at the root of a term—see Theorem 3.8—we can move the computation forward:

$$M \equiv M_1 \mapsto_{sc} \cdots \mapsto_{sc} M_l \equiv \mathscr{A}M_l' \triangleright_{\mathscr{A}} M_l' \rightarrow_c \cdots \rightarrow_c M_k' \triangleright_c \cdots \triangleright_c M_n \equiv N.$$

By inductive hypothesis we get a similar reduction sequence for the reduction from M_l' to N. Since $M \mapsto_{sc}^{\triangleright +} M_l'$ we can form the desired $C^{\triangleright}$-SCS from M to N. The case of $\triangleright_{\mathscr{C}}$-computations is treated similarly. □

This ends the investigation of the logical properties of the λ_c-calculus. The two results enable us to show that the λ_c-calculus corresponds to the C-rewriting system.

4. The machine-calculus correspondence

Following Plotkin [14] a calculus is correct with respect to a programming language if the operational semantics of the calculus agrees with the original machine semantics and if equality in the calculus implies operational equality. If a calculus is correct, we say that it corresponds to the machine, thus underlining that calculi and programming languages are pairs which determine each other.

To prove the equivalence of the machine semantics with the operational rewriting semantics of λ_c, we show that the standard computation function simulates the rules (C1) through (C4) of Definition 2.2. Since an evaluation may return a continuation point as (part of) the result, we first construct a morphism from Λ_p-terms to Λ_c-terms so that the C-rewriting machine can be properly unloaded. The major task of this morphism is to encode contexts as terms in the same way as the $\mathscr{C}$-reductions do.

For the construction of the morphism it is advantageous to look at contexts from the inside out, i.e.,

$$C[\,] ::= [\,] \mid C[[\,]M] \mid C[V[\,]].$$

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

$$[\![[\,]]\!]_c \equiv \lambda x.\,\mathscr{A}x,$$

$$[\![C[[\,]M]]\!]_c \equiv \lambda f.\,\mathscr{A}([\![C[\,]]\!]_c(f\bar{M})),$$

$$[\![C[V[\,]]]\!]_c \equiv \lambda v.\,\mathscr{A}([\![C[\,]]\!]_c(\bar{V}v)).$$

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

$$\overline{\langle p, C[\,]\rangle} \equiv [\![C[\,]]\!]_c, \qquad \bar{x} \equiv x, \qquad \overline{\lambda x.\,M} \equiv \lambda x.\,\bar{M},$$

$$\overline{MN} \equiv \bar{M}\bar{N}, \qquad \overline{\mathscr{C}M} \equiv \mathscr{C}\bar{M}, \qquad \overline{\mathscr{A}M} \equiv \mathscr{A}\bar{M}.$$

Given the morphisms, we can attempt to prove a simulation theorem for the four C-rewriting clauses. The β_v-step, i.e., (C1), and steps (C2) and (C4) are clearly reflected in the definition of the standard computation function. In particular, the latter two 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 4.1. *For any applicative context* $C[\ ]$,
(i) $C[\mathscr{C}M] \mapsto_{\mathrm{sc}}^{\rhd +} M[\![C[\ ]]\!]_c$, *and*
(ii) $C[\mathscr{A}M] \mapsto_{\mathrm{sc}}^{+} \mathscr{A}M \mapsto_{\mathrm{sc}}^{\rhd} M$.

Proof. The proof is a straightforward induction on the structure of applicative contexts. □

We are, however, unable to show that the standard reduction function satisfies rule (C3). This transition rule requires that a continuation invocation remove the current context and that it continue as if the old context—filled with the argument—were the new term. The first condition is clearly implemented since continuations immediately perform an $\mathscr{A}$-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 is invoked, the transition in the machine and the one via the standard computation function diverge. The machine simply labels the current context which contains the old continuation context; the standard computation 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 the continuation point $\langle p, C[[\ ]V]\rangle$ is invoked on the value F:

$$\langle p, C[[\ ]V]\rangle F \mapsto^{C} C[FV].$$

Furthermore, assume that the application FV evaluates to $D[\mathscr{C}P]$ after some β_v-steps. Then the C-transition reaches the term $P\langle p, C[D[\ ]]\rangle$. According to Lemma 4.1, if $K_c \equiv [\![C[\ ]]\!]_c$, the corresponding reduction sequence in the λ_c-calculus begins with

$$[\![C[[\ ]V]]\!]_c F \mapsto_{\mathrm{sc}}^{\rhd +} K_c(FV).$$

The next few β_v-steps for FV are correctly simulated by the standard computation function:

$$K_c(FV) \mapsto_{\mathrm{sc}}^{\rhd +} K_c D[\mathscr{C}P].$$

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

$$K_c D[\mathscr{C}P] \mapsto_{\mathrm{sc}}^{\rhd +} P[\![K_c D[\ ]]\!]_c.$$

Clearly, $[\![C[D[\ ]]]\!]_c$ is not equal to $[\![K_c D[\ ]]\!]_c$ and thus, a naive version of the simulation theorem fails. The best we can hope for is that the standard computation simulation of the C-transition preserves a relation between continuation points and terms.

From the above lemma and the example one may suspect that a continuation point like $\langle p, C[D[\;]]\rangle$ is related to the terms $[\![C[D[\;]]]\!]_c$ and $[\![[\![C[\;]]\!]_c D[\;]]\!]_c$. However, the situation in our example could recur may times. Instead of having two contexts composing a new one, we would then have several of them. In fact, we must account for all possible finite decompositions of a given context into smaller contexts, including the empty one. Each 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 4.2.

Definition 4.2 (*the continuation point-term correspondence*). The relation $\approx_p$ basically compares continuation points in Λ_p to terms in Λ_c. It is defined inductively over Λ_p and applicative contexts (over Λ_p):

$$\langle p, C[\;]\rangle \approx_p [\![\ldots[\![[\![\bar{C}_1[\;]]\!]_c \bar{C}_2[\;]]\!]_c \ldots \bar{C}_n[\;]]\!]_c$$
for all n such that $C[\;] \equiv C_1[C_2[\ldots C_n[\;]\ldots]]$
and $C_i[\;] \approx_p \bar{C}_i[\;]$ for all $i \leq n$;

$$x \approx_p x,\; \lambda x.P \approx_p \lambda x.\bar{P},\; PQ \approx_p \bar{P}\bar{Q},\; \mathscr{C}P \approx_p \mathscr{C}\bar{P},\; \mathscr{A}P \approx_p \mathscr{A}\bar{P}$$
iff $P \approx_p \bar{P}$ and $Q \approx_p \bar{Q}$.

For applicative contexts we add $[\;] \approx_p [\;]$.

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

The relation $\approx_p$ in Definition 4.2 is implicit. It is well-suited to capture the different continuation representations from the above 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. If there is a proper term contained in the continuation point-context, exactly one of the partitioning contexts covers it, and therefore, each subterm appears exactly once in the representation. Furthermore, empty contexts correspond to $\lambda x.\mathscr{A}x$ and, putting the two observations together, we see that the terms that are related to a continuation point are the same modulo some occurrences of $\lambda x.\mathscr{A}x$:

Lemma 4.3. *Let $\langle p, C[\;]\rangle$ be a continuation point. Furthermore, let $P, V \in \Lambda_p$ and $\bar{P}, \bar{V} \in \Lambda_c$ such that $P \approx_p \bar{P}$ and $V \approx_p \bar{V}$. Define three term-sequence schemas K_i^1, K_i^2, and K_i^3 for all representations of K_c, i.e., $\langle p, C[\;]\rangle \approx_p K_c$, such that $K_1^1 \equiv \lambda x.\mathscr{A}x$, $K_1^2 \equiv \lambda f.\mathscr{A}(K_c(f\bar{P}))$, $K_1^3 \equiv \lambda v.\mathscr{A}(K_c(\bar{V}v))$, and $K_{i+1} \equiv \lambda x.\mathscr{A}((\lambda x.\mathscr{A}x)(K_i x))$. Then, a continuation point is related to exactly one of the three schemas, i.e., for all i and K_c,*

(i) $\langle p, [\;]\rangle \approx_p K_i^1$,
(ii) $\langle p, C[[\;]P]\rangle \approx_p K_i^2$, *or*
(iii) $\langle p, C[V[\;]]\rangle \approx_p K_i^3$.

Furthermore, we can generalize this to

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

Proof. First note that (i), (ii), and (iii) cover all possible cases of applicative contexts. One of them must match a particular applicative 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

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

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

$$\begin{aligned}&[\![\ldots [\![[\![\bar{C}_1[\ ]]\!]_c \bar{C}_2[\ ]]\!]_c \ldots \bar{D}[[\ ]\bar{P}]]\!]_c \\ &\quad \equiv \lambda f . \mathscr{A}([\![\ldots [\![[\![\bar{C}_1[\ ]]\!]_c \bar{C}_2[\ ]]\!]_c \ldots \bar{D}[\ ]]\!]_c (f\bar{P})).\end{aligned}$$

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

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

This proves the case for $i = 1$.

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

$$\begin{aligned}[\![\ldots [\![[\![\bar{C}_1[\ ]]\!]_c \bar{C}_2[\ ]]\!]_c \ldots [\ ]]\!]_c &\equiv \lambda x . \mathscr{A}([\![[\ ]]\!]_c([\![\ldots [\![[\![\bar{C}_1[\ ]]\!]_c \bar{C}_2[\ ]]\!]_c \ldots]\!]_c x)) \\ &\equiv \lambda x . \mathscr{A}((\lambda x . \mathscr{A}x) \\ &\qquad ([\![\ldots [\![[\![\bar{C}_1[\ ]]\!]_c \bar{C}_2[\ ]]\!]_c \ldots]\!]_c x)).\end{aligned}$$

Thus we see that, as mentioned above, every empty context adds one term $\lambda_x . \mathscr{A}x$. Hence, $\langle p, C[[\ \ ]P]\rangle \approx_p K_{i+1}$ and this concludes the induction step.

The generalization follows immediately. □

Lemma 4.3 indicates an important fact about the terms in the representation set of $\langle p, C[\ ]\rangle$: they are behaviorally indistinguishable in standard computation sequences with respect to β_v-steps. They invoke a continuation and, since continuations always remove the current context, none of the $\lambda x . \mathscr{A}x$ ever plays a role in an evaluation.

Proposition 4.4. *Define three term sets as in Lemma* 4.3 *with the initial terms* $\lambda x . \mathscr{A}x$, $\lambda f . \mathscr{A}(K_c(f\bar{P}))$, *and* $\lambda v . \mathscr{A}(K_c(\bar{V}v))$. *Then we can show that*

$$K_i^1, \ldots, \lambda x . \mathscr{A} \ldots (i \text{ times}) \ldots x,$$

$$K_i^2, \ldots, \lambda f . \mathscr{A} \ldots (i \text{ times}) \ldots (K_c(f\bar{P})),$$

$$K_i^3, \ldots, \lambda v . \mathscr{A} \ldots (i \text{ times}) \ldots (K_c(\bar{V}v)),$$

are standard reduction sequences.

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

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

and all are standard steps. But, the three M_i^x's for the base cases are x, $(K_c(x\bar{P}))$, and $(K_c(\bar{V}x))$ respectively. □

Proposition 4.4 says that all continuations related to a continuation point behave similarly when invoked; the difference is the number of abort operations. Thus, we can show that evaluations via the standard computation function and the C-rewriting system only differ in their outcome. First, we prove that the standard computation function mirrors C-transition steps as long as no continuation is invoked:

Lemma 4.5. *Assume $C[\ ] \approx_p \bar{C}[\ ]$, $P \approx_p \bar{P}$, and $U \approx_p \bar{U}$. The simulation of the rules* (C1), (C2), *and* (C4) *via* $\mapsto_{sc}^{\triangleright}$ *respects* $\approx_p$:

(i) *if* $C[(\lambda x.\,P)U] \mapsto^C C[P[x:=U]]$, *then* $\bar{D}[(\lambda x.\,\bar{P})\bar{U}] \mapsto_{sc}^{\triangleright} \bar{D}[\bar{P}[x:=\bar{U}]]$ *for any applicative context* $\bar{D}[\ ]$;

(ii) *if* $C[\mathcal{A}P] \mapsto^C P$, *then* $\bar{C}[\mathcal{A}\bar{P}] \mapsto_{sc}^{\triangleright +} \bar{P}$;

(iii) *if* $C[\mathcal{C}P] \mapsto^C P\langle p, C[\ ]\rangle$, *then* $\bar{C}[\mathcal{C}\bar{P}] \mapsto_{sc}^{\triangleright +} \bar{P}[\![\bar{C}[\ ]]\!]_c$.

Proof. The first statement reiterates that β-steps are simulated independently of the context. Points (ii) and (iii) are consequences of Lemma 4.1. □

Things get more complicated when a continuation is invoked. The standard computation 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 the standard computation function is possible.

Lemma 4.6. *Suppose* $\langle p, C_0[\ ]\rangle \approx_p K_0$, $V \approx_p \bar{V}$, *and* $U \approx_p \bar{U}$. *Then,*

$$C[\langle p, C_0[\ ]\rangle V] \mapsto^{C+} U \text{ iff } \bar{C}[K_0\bar{V}] \mapsto_{sc}^{\triangleright +} \bar{U}.$$

Proof. The condition $C[\ ] \approx_p \bar{C}[\ ]$ 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 $\mapsto^{C}$-reduction sequence from $C[\langle p, C_0[\ ]\rangle]$ to U. We proceed by case analysis on the structure of $C_0[\ ]$:

(apC1): $C_0[\ ] \equiv [\ ]$. This case is trivial. It implies that

$$K_0 \equiv K_1 \equiv \lambda x.\mathcal{A}x, \quad \text{or}$$

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

In any case, we have

$$\langle p, C_0[\ ]\rangle V \mapsto^{C} V \quad \text{and} \quad K_0\bar{V} \mapsto_{\text{sc}}^{\rhd+} \bar{V}.$$

(apC2): $C_0[\ ] \equiv D[[\ ]P]$ for some Λ_p-applicative context and term P. Now we know from Lemma 4.3 that

$$K_0 \equiv K_1 \equiv \lambda f.\mathcal{A}(K_D(f\bar{P})), \quad \text{or}$$

$$K_0 \equiv K_2 \equiv \lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)(K_1x)), \quad \text{etc.}$$

where $\langle p, D[\ ]\rangle V \approx_p K_D$ and $P \approx_p \bar{P}$. The two reduction sequences start out with

$$C[\langle p, C_0[\ ]\rangle V] \mapsto^{C} D[VP]$$

and

$$\bar{C}[K_0\bar{V}] \mapsto_{\text{sc}}^{\rhd+} K_D(\bar{V}\bar{P}).$$

Next, we consider the possible evaluations of VP and $\bar{V}\bar{P}$. 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 criterium for the rest of the reduction sequence. Since this sequence is finite, four cases must be analysed:

(a) $VP \mapsto_{(\text{C1})}^{C+} W$ where W is a value. This means that $\bar{V}\bar{P} \mapsto_{\text{sc}}^{\rhd+} \bar{W}$ and we have the following development for the C-transition:

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

For the one according to $\mapsto_{\text{sc}}^{\rhd}$ we get

$$K_D(\bar{V}\bar{P}) \mapsto_{\text{sc}}^{\rhd+} K_D\bar{W}.$$

By assumption we know that

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

From the definition of $\mapsto^{C}$ we see that

$$D[W] \mapsto^{C\,m} U \ \text{iff}\ \langle p, D[\ ]\rangle W \mapsto^{C\,m+1} U.$$

Thus, we can safely replace $D[W]$ by $\langle p, D[\ ]\rangle W$ since $m+1 \leq n-1$. But note, $\langle p, D[\ ]\rangle \approx_p K_D$ and so, by inductive hypothesis, we get the desired conclusion.

(b) $VP \mapsto^{C*}_{(C1)} E[\mathscr{A}Q]$ and $\bar{VP} \mapsto^{\rhd *}_{sc} \bar{E}[\mathscr{A}\bar{Q}]$ for some term Q and applicative context $E[\ ]$. Comparing the two reduction sequences

$$D[VP] \mapsto^{C*} D[E[\mathscr{A}Q]] \mapsto^{C} Q$$

and

$$K_D(\bar{VP}) \mapsto^{\rhd *}_{sc} K_D\bar{E}[\mathscr{A}\bar{Q}] \mapsto^{\rhd *}_{sc} \bar{Q},$$

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 does not:

(b1) If $Q \mapsto^{C*} U$ does not use (C3), then, according to Lemma 4.5, $\bar{Q} \mapsto^{\rhd *}_{sc} \bar{U}$ is immediate.

(b2) Suppose (C3) is used a first time. That means that

$$Q \mapsto^{C*} F[\langle p, F_0[\ ]\rangle W]$$

and also, by Lemma 4.5, that

$$\bar{Q} \mapsto^{\rhd *}_{sc} \bar{F}[K_F \bar{W}]$$

such that $\langle p, F_0[\ ]\rangle \approx_p K_F$. Since the reduction sequence is at least one step shorter, we can now apply our inductive hypothesis, and this finishes case (b).

(c) $VP \mapsto^{C*}_{(C1)} E[\mathscr{C}Q]$ and $\bar{VP} \mapsto^{\rhd *}_{sc} \bar{E}[\mathscr{C}\bar{Q}]$. The reduction sequence according to $\mapsto^{\rhd}_{sc}$ continues as

$$K_D(\bar{VP}) \mapsto^{\rhd *}_{sc} K_D\bar{E}[\mathscr{C}\bar{Q}] \mapsto^{\rhd +}_{sc} \bar{Q}[\![K_D\bar{E}[\ ]]\!]_c.$$

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

$$D[VP] \mapsto^{C*} D[E[\mathscr{C}Q]] \mapsto^{C} Q\langle p, D[E[\ ]]\rangle.$$

By assumption, $\langle p, D[\ ]\rangle \approx_p K_D$ and hence, $\langle p, D[E[\ ]]\rangle \approx_p [\![K_D\bar{E}[\ ]]\!]$ by Lemma 4.3. The rest of this subcase is as in (b).

(d) $VP \mapsto^{C*}_{(C1)} E[\langle p, E_0[\ ]\rangle W]$ and $\bar{VP} \mapsto^{\rhd *}_{sc} \bar{E}[K_E \bar{W}]$ with $\langle p, E_0[\ ]\rangle \approx_p K_E$. This is an instance of the inductive hypothesis and the case (apC2) is finished.

(apC3): $C_0[\ ] \equiv D[P[\ ]]$ for some Λ_p-applicative context and value P. Again, the respective continuations are characterized by $K_1 \equiv \lambda v\,.\,\mathscr{A}(K_D(\bar{P}v))$, etc. The two reduction sequences immediately arrive at the same constellation as in (apC2):

$$C[\langle p, C_0[\ ]\rangle V] \mapsto^{C} D[PV]$$

and

$$\bar{C}[K_0 \bar{V}] \mapsto^{\rhd +}_{sc} K_D(\bar{P}\bar{V}).$$

The rest is analogous to the previous case. □

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

Theorem 4.7 (Simulation). *For any program $M \in \Lambda_c$, and value V, $\bar{V}$ exist such that $V \approx_p \bar{V}$*

$$M \mapsto^{C+} V \textit{ iff } M \mapsto^{\rhd +}_{sc} \bar{V}.$$

Since $V \approx_p \bar{V}$ implies $V \equiv \bar{V}$ for $V \in \Lambda_c$, the theorem can be specialized to the following corollary.

Corollary 4.8. *For any program $M \in \Lambda_c$ whose result V is continuation-free,*

$$M \mapsto^{C+} V \text{ iff } M \mapsto_{sc}^{\triangleright+} V.$$

A more general consequence is that eval_C is only defined if the program is equivalent to a value:

Corollary 4.9. *For any program $M \in \Lambda_c$, there exists a value V such that $\lambda_c \vdash M =_c^{\triangleright} V$ iff* $\text{eval}_C(M)$ *is defined.*

Informally, these results mean that the C-machine is characterized by a standard computation 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 $K(\mathscr{C}M)$ 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 one could easily construct such a term K by hand, 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. The result of a *batch* computation is generally expected to be a basic constant, and Corollary 4.9 assures us that we get the correct result back since we encode these values in Λ. Otherwise, if a nonbasic value is the result, a sensible interpretation is impossible because these values represent machine behavior. On the other hand, if a machine is used interactively where intermediate results are saved, the user can only be interested in getting such values back for potential future use. In this case we are safe because of Proposition 4.4. All terms that are related to a continuation point are behaviorally equivalent. Thus, we can assume that eval_C and the operational semantics of λ_c are equivalent.

A disadvantage of the above theorem and corollaries is their dependence on the standard compution function of the calculus. One would prefer to interpret terms in a less operational way using the equivalence relation instead. Traditionally, one thinks of terms as functions from some set of basic constants[5] to basic constants. A program is equivalent to a basic constant and hence, it is a null-ary function. Following Morris [13] and Plotkin [14] we define two interpretations of terms. For all $n \geq 0$, the *calculus interpretation* of a term M is the function

$$\mathscr{I}_M^n = \{\langle N_1, \ldots, N_n, V\rangle \mid \lambda_c \vdash MN_1 \ldots N_n =_c^{\triangleright} V\}$$

[5] For the following two theorems we must assume that constants are represented by an equivalence class of terms, the normal form being the typical representative. Alternatively, we could have introduced constants into Λ_c, but that would have complicated the treatment of fundamental properties.

where the N_i and V are basic values. The *machine interpretation* of a term M is the function

$$\mathcal{M}^n_M = \{\langle N_1, \ldots, N_n, V\rangle \mid \text{eval}_C(MN_1 \ldots N_n) \equiv V\}.$$

Given these interpretations, the correspondence of the C-machine to the λ_c-calculus is independent of a standard computation function.

Theorem 4.10. *For any program M in Λ_c, its calculus and machine interpretation are the same for all $n \geqslant 0$:*

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

Proof. The theorem is a consequence of the Church–Rosser Theorem, Corollary 4.8, and Corollary 4.9. □

Theorem 4.10 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 $=^{\triangleright}_c$ is not a congruence relation, it is clear that $M =^{\triangleright}_c N$ does *not* mean that $\mathcal{M}^n_M = \mathcal{M}^n_N$ for any $n > 0$. The relation $=^{\triangleright}_c$ only compares programs that are already supplied with all their input arguments. Intuitively, the equivalence relation $=^{\triangleright}_c$ equates 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 generalizes to what equality in λ_c means for open expressions, i.e., whether equality is preserved under all possible interpretations [13]. Put differently, we are asking whether equality in the calculus implies operational equality. From the above discussion about $=^{\triangleright}_c$ and $=_c$, we know that only $=_c$ implies operational equivalence. For $=^{\triangleright}_c$ we need to make sure that the terms behave equivalently in all applicative contexts; then it also implies operational equivalence.

Theorem 4.11. *For M, N in Λ_c,*

(i) *if $\lambda_c \vdash M =_c N$, then $M \simeq_C N$, and*

(ii) *if $\lambda_c \vdash C[M] =^{\triangleright}_c C[N]$ for all applicative contexts $C[\ ]$, then $M \simeq_C N$.*

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, without loss of generality, assume that M and N are in Λ_c proper. Let $D[\ ]$ be a context such that $D[MN]$ is closed. Now, suppose that $\text{eval}_C(D[M])$ is defined and, furthermore, that it is a basic constant. By Theorem 3.10 and Corollary 4.9,

$$\lambda_c \vdash D[M] =^{\triangleright}_c \text{eval}_C(D[M]).$$

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 a standard 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 redex in some applicative context. But note, $\lambda_c \vdash C[M] =_c^{\triangleright} C[N]$ implies $\lambda_c \vdash C[M[x := L]] =_c^{\triangleright} C[N[x := L]]$ for all values L. Therefore, with the necessary generalization to multiple substitutions, we have

$$\lambda_c \vdash \text{eval}_C(D[M]) =_c^{\triangleright} \text{eval}_C(C[M[\vec{x} := \vec{L}]]) =_c^{\triangleright} \text{eval}_C(C[N[\vec{x} := \vec{L}]]).$$

Hence, by the Church–Rosser Theorem, $D[M]$ and $D[N]$ produce the same result. □

The inverses of both statements are 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 $\simeq_C$.

The second point of Theorem 4.11 is important. Together with the above theorems and corollaries it yields an interesting system for dealing with continuations. The theorem implies that it is sufficient to consider all applicative contexts instead of program contexts for behavioral considerations. Theorem 4.7 and Corollary 4.8 provide the basis for using the context-rewriting rules in conjunction with the calculus reductions; this is helpful when reasoning about redexes in arbitrary applicative contexts. In the last section we briefly discuss some possible applications of this theory.

5. Reasoning with the λ_c-calculus

In the preceding sections we have shown how the λ-calculus can be extended to a control calculus. The resulting system is correct with respect to the C-rewriting semantics which in turn is equivalent to a classical operational interpretation of a denotational semantics. Together, the two characterizations form a syntactic theory of control. The predominant use that we perceive is for symbolic manipulations of programs, e.g., verification and evaluation.

With our first application, we return to the Σ_0^*-function from Subsection 2.3. Recall that Σ_0^* sums the numbers in a tree unless a 0 occurs in the tree, in which case it returns a 0. Given a predicate **has-zero?** that tests whether a tree contains a 0 or not, we can formulate and prove a correctness claim.

Proposition 5.1. *For all number-trees t, Σ_0^* operationally satisfies its specification:*

$$(\Sigma_0^* t) \simeq_C (\textbf{if } (\textbf{has-zero?}\ t) \ulcorner 0 \urcorner (\Sigma^* t)).$$

Proof. Let $C[\ ]$ be an arbitrary applicative context and consider an application of Σ_0^* in this context:

$$C[\Sigma_0^* t] =_c^{\triangleright} C[\mathscr{C}\lambda k.\, k(\mathrm{Y}_v E_k t)] =_c^{\triangleright} C[\mathrm{Y}_v F t],$$

where

$$E_k \equiv \lambda s.\lambda t.(\mathbf{if}(\mathbf{mt?}\ t)\lceil 0\rceil \\ (\mathbf{if}(\mathbf{zero?}(\mathbf{num}\ t))(k\lceil 0\rceil) \\ (+(\mathbf{num}\ t)(+(s(\mathbf{lson}\ t))(s(\mathbf{rson}\ t))))))$$

and

$$F \equiv E_k[k := \langle p, C[\ ]\rangle].$$

At this point, we must show that the recursive function $\mathbf{Y}_v F$ behaves correctly. To this end, we split the claim into two subcases:

(i) if $(\mathbf{has\text{-}zero?}\ t) \equiv \mathbf{F}$, then $D[\mathbf{Y}_v Ft] =_c^{\triangleright} D[\Sigma^* t]$;

(ii) if $(\mathbf{has\text{-}zero?}\ t) \equiv \mathbf{T}$, then $D[\mathbf{Y}_v Ft] =_c^{\triangleright} D[\langle p, C[\ ]\rangle\lceil 0\rceil] =_c^{\triangleright} C[\lceil 0\rceil]$.

For (i), observe that $(\mathbf{zero?}(\mathbf{num}\ t))$ can never be true and hence,

$$(\mathbf{if}(\mathbf{zero?}(\mathbf{num}\ t))(k\lceil 0\rceil)(+(\mathbf{num}\ t)(+(s(\mathbf{lson}\ t))(s(\mathbf{rson}\ t)))))$$
$$=_c (+(\mathbf{num}\ t)(s(\mathbf{lson}\ t))(s(\mathbf{rson}\ t))).$$

From this, (i) follows immediately.

The second claim we prove by an induction on the structure of a tree. Suppose that there is a 0 in the tree and that it is at the root. Then we have the following evaluation in any applicative context $D[\ ]$:

$$D[\mathbf{Y}_v Ft] =_c^{\triangleright} D[\langle p, C[\ ]\rangle\lceil 0\rceil].$$

Otherwise, the valuation takes the second **if**-branch and we have

$$D[\mathbf{Y}_v Ft] =_c^{\triangleright} D[(+(\mathrm{num}\ t)(+(\mathbf{Y}_v F(\mathbf{lson}\ t))(\mathbf{Y}_v F(\mathbf{rson}\ t))))].$$

Assuming that a 0 is in the left subtree, the inductive hypothesis yields

$$\ldots =_c^{\triangleright} D[(+(\mathbf{num}\ t)(+(\langle p, C[\ ]\rangle\lceil 0\rceil)(\mathbf{Y}_v F(\mathbf{rson}\ t))))]$$

for the applicative context $D[(+(\mathbf{num}\ t)(+[\ ](\mathbf{Y}_v F(\mathbf{rson}\ t))))]$. But note, because of the abortive character of continuation representations in applicative contexts (see Proposition 4.4), this results in

$$\ldots =_c^{\triangleright} D[\langle p, C[\ ]\rangle\lceil 0\rceil].$$

For the final case, where the 0 is in the right subtree, the proof relies on part (i), but is otherwise similar to the above. □

This first application of the λ_c-calculus demonstrates how to reason about a specific use of $\mathscr{C}$-applications. Since $\mathscr{C}$- and $\mathscr{A}$-applications abstract from an extension of continuation-passing style programming, a more general question is whether the λ_c-calculus can prove as many results as the λ-calculus about the respective cps'ed programs. The question naturally divides into two subproblems, namely whether the λ_c-calculus-relations preserve equality in the λ-calculus. and vice versa. We refer to the first as the soundness and the second as the completeness question.

As for soundness, the proof is a tedious but straightforward calculation. The soundness of the β_v-reduction is known from Plotkin's investigation of the λ_v-calculus [14][6].

Proposition 5.2 (Soundness). *For $M \in \Lambda_c$ and for $L \in \Lambda$ an abstraction, if $M =_c N$, then $[\![M]\!]L =_\beta [\![N]\!]L$, and if $M =_c^{\triangleright} N$, then $[\![M]\!]\mathbf{I} =_\beta [\![N]\!]\mathbf{I}$.*

Proof. The proof is a two-step procedure which for the most part can be carried out by a program. First, the left-hand side and the right-hand side of each rule is translated into the λ-calculus via cps. Then the resulting expressions are reduced until no β-redexes are left. For the $\mathscr{C}_R$ and $\mathscr{A}_R$ cases one needs the assumption that L is an abstraction; in all other cases, the respective left-hand and right-hand terms are already equivalent. □

Unfortunately, the calculus is not complete as expressed in the following proposition.

Proposition 5.2′ (Incompleteness). *There are M and N such that, for some abstraction L, $[\![M]\!]L =_\beta [\![N]\!]L$ but $M \neq_c N$.*

Proof. The proposition is a consequence of Theorem 3.8 and of Plotkin's value-calculus being a subcalculus. An example is given by $M \equiv (\omega\omega)y$ and $N \equiv (\lambda x . xy)(\omega\omega)$ where $\omega \equiv (\lambda x . xx)$. □

Thus far, we have used the λ_c-calculus as an equational extension of the C-rewriting system. With the exception of the induction step in Proposition 5.1, the C-reductions have played no role. The following proposition shows that they are indeed useful for local program transformations.

Proposition 5.3. *Define $[\![\cdot]\!]_c^k$ as a variant of $[\![\cdot]\!]_c$:*

$$[\![[\ ]]\!]_c^k \equiv k,$$

$$[\![C[[\ ]M]]\!]_c^k \equiv \lambda f . \mathscr{A}([\![C[\ ]]\!]_c^k(f\bar{M})),$$

$$[\![C[V[\ ]]]\!]_c^k \equiv \lambda v . \mathscr{A}([\![C[\ ]]\!]_c^k(\bar{V}v)).$$

Then the following two statements hold for any applicative context $C[\ ]$:

(i) $\lambda x . C[\mathscr{A}M] =_C \lambda x . \mathscr{A}M$, *and*

(ii) $\lambda x . C[\mathscr{C}M] =_C \lambda x . \mathscr{C}\lambda k . M[\![C[\ ]]\!]_c^k$.

[6] See Plotkin's Translation Theorem [14, p. 148] which discusses the use of cps'ed programs for the simulation of a call-by-value abstraction in the traditional λ-calculus.

Proof. (i): By Lemma 4.1, $\lambda_c \vdash \lambda x.\, C[\mathscr{A}M] =_c \lambda x.\, \mathscr{A}M$.

(ii): We prove the statement by an induction on the structure of the context $C[\;]$. If it is empty, we must show that $\lambda x.\, \mathscr{C}M \simeq_C \lambda x.\, \mathscr{C}\lambda k.\, Mk$. But this follows from $\mathscr{C}M \simeq_C \mathscr{C}\lambda k.\, Mk$, which obviously holds. Otherwise, assume, without loss of generality, that we have a context of the form $C[[\;]N]$. Then we get the conclusion by a simple calculation:

$$\begin{aligned}\lambda x.\, C[(\mathscr{C}M)N] &=_c \lambda x.\, C[\mathscr{C}\lambda\kappa.\, M(\lambda f.\, \mathscr{A}(\kappa(fN)))]\\ &\simeq_C \lambda x.\, \mathscr{C}\lambda k.\,(\mathscr{C}\lambda\kappa.\, M(\lambda f.\, \mathscr{A}(\kappa(fN))))[\![C[\;]]\!]^k_c\\ &\qquad\qquad\text{by inductive hypothesis}\\ &=_c \lambda x.\, \mathscr{C}\lambda k.\, M(\lambda f.\, \mathscr{A}([\![C[\;]]\!]^k_c(fN)))\\ &=_c \lambda x.\, \mathscr{C}\lambda k.\, M[\![C[[\;]N]]\!]^k_c. \qquad \square\end{aligned}$$

The proposition is useful in two different areas: source-to-source transformations and compiler optimizations. Moving the $\mathscr{C}$-application to a procedure entrance can improve a programmer's understanding about the intension of a procedure body. At the same time it may save some cost since on various machine architectures it may be cheaper to label continuations at the invocation of a function.

For non-Von-Neumann machine architectures the reduction characterization of control operations is even more important. For example, a term-rewriting machine [10] can reduce all (reduction) redexes in parallel. Only computation steps must be ordered, yet this happens naturally. At first glance, it may appear that the reduction system exchanges short C-rewriting steps for long chains of reductions, but these appearances are deceiving. Newly built β_v-redexes within right-hand sides of $\mathscr{C}_L$- and $\mathscr{C}_R$-steps can immediately be reduced. By the time a $\mathscr{C}$-application has reached the top of the term, it may already have finished the construction of the continuation. The trade-offs between these two computational models clearly deserve more investigation.

In essence the development of a syntactic theory of control has demonstrated that our understanding of programming with control operations is far from complete. It raises the question of what continuations really are. In denotational semantics they are represented by functions. But this only works because the definitions are expressed in a particular style, namely continuation-passing style. Their nature as programming objects remains concealed. We expect that a further investigation of the λ_c-calculus will deepen our understanding of progamming with continuations and the nature of control operations in programming languages.

Acknowledgment

We wish to thank Carolyn Talcott and Mitchell Wand for their helpful discussions and comments on early drafts of this paper. Michael Dunn helped to clarify the

Soundness and Incompleteness Propositions. We also thank John Gately and Chris Haynes for their comments. The anonymous referees suggested several improvements to the organization and presentation of the material.

Matthias Felleisen and Eugene Kohlbecker are IBM Research Graduate Fellows.

References

[1] H.P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics* (North-Holland, Amsterdam, 1981).

[2] M. Felleisen and D.P. Friedman, Control operators, the SECD-machine, and the λ-calculus, in: *Formal Description of Programming Concepts III* (North-Holland, Amsterdam, 1986) 193-217.

[3] M. Felleisen, D.P. Friedman, E. Kohlbecker and B. Duba, Reasoning with continuations, in: *Proc. First Symp. on Logic in Computer Science* (1986) 131-141.

[4] M.J. Fischer, Lambda calculus schemata, in: *Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices* **7**(1) (1972) 104-109.

[5] D.P. Friedman, C.T. Haynes and E. Kohlbecker, Programming with continuations, in: P. Pepper, ed., *Program Transformations and Programming Environments* (Springer, Berlin, 1985) 263-274.

[6] C.T. Haynes and D.P. Friedman, Embedding continuations in procedural objects, *TOPLAS* **9**(4) (1987).

[7] C.T. Haynes, Logic continuations, *J. Logic Programming* **4**(2) (1987) 157-176.

[8] P.J. Landin, An abstract machine for designers of computing languages, in: *Proc. IFIP Congress* (1965) 438-439.

[9] P.J. Landin, The mechanical evaluation of expressions, *Comput. Journal* **6**(4) (1964) 308-320.

[10] G.A. Magó, A network of microprocessors to execute reduction languages, *Internat. J. Comput. Inform. Sci.* **8** (1979) 349-385; 435-471.

[11] A.W. Mazurkiewicz, Proving algorithms by tail functions, *Inform. and Control* **18** (1971) 220-226.

[12] C. Mellish and S. Hardy, Integrating Prolog in the POPLOG environment, in: J. A. Campbell, ed., *Implementations of Prolog* (Ellis Horwood Series in Artificial Intelligence, 1984) 147-162.

[13] J.H. Morris, Lambda-calculus models of programming languages, Ph.D. Thesis, Project MAC, MAC-TR-57, MIT, 1968.

[14] G.D. Plotkin, Call-by-name, call-by-value, and the λ-calculus, *Theoret. Comput. Sci.* **1** (1975) 125-159.

[15] J. Rees and W. Clinger, eds., The revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21**(12) (1986) 37-79.

[16] J.C. Reynolds, Definitional interpreters for higher-order programming languages, *Proc. ACM Annual Conference* (1972) 717-740.

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

[18] T.B. Steel, ed., *Formal Language Description Languages for Computer Programming* (North-Holland, Amsterdam, 1966).

[19] C. Strachey and C.P. Wadsworth, Continuations: A mathematical semantics for handling full jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

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

[21] C. Talcott, The essence of rum—a theory of the intensional and extensional aspects of Lisp-type computation, Ph.D. Dissertation, Stanford University, 1985.

[22] M. Wand, Continuation-based program transformation strategies, *J. ACM* **27**(1) (1980) 164-180.