Reasoning With Continuations

by

Matthias Felleisen, Daniel P. Friedman,
Eugene Kohlbecker, Bruce Duba

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 191

Reasoning With Continuations

by

M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba

May, 1986

This material is based on work supported by the National Science Foundation under grants number MCS 83-03325 and DCR 85-01277.

Reasoning With Continuations

Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, Bruce Duba

Indiana University, Bloomington, IN 47405, USA

Abstract

The λ-calculus is extended with two operations and the corresponding reduction rules: $\mathcal{C}$, which gives access to the current continuation, and $\mathcal{A}$, which is an *abort* or *stop* operator. The extended system is a sound and consistent calculus. We prove a standardization theorem and adopt the standard reduction function as an operational semantics. Based on it, we study the access to and invocation of continuations in a purely syntactic setting. With the derived rules, programming with continuations becomes as easy as programming with functions.

1. Deficiencies of the λ-calculus

"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] 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 was also a major obstacle in the further development of the calculus as a programming language since it was based on simplicity rather than convenience.

The one and only means of computation in the pure calculus is the β-reduction rule which directly models function application. Although this

[1] [1], p.3

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 erroneous 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.

The most general solution of the control problem within the functional realm originated in denotational semantics. A program can be evaluated by evaluating its pieces and combining the results. When one particular component is being evaluated, one can think of the remaining sub-evaluations and the combination step, *i.e.* the rest of the computation, as the *continuation* of the current sub-evaluation. The crucial idea is to write programs in such a style that functions can be used to *simulate* continuations. In other words, these programs always pass around and explicitly invoke (a functional representation of) the continuation. They are thus able to direct the evaluation process: they may decide *not* to use the current continuation, to save it in a data-structure for later use, or to resume a continuation from some other point in time. However, such programs look clumsy and are hard to construct. 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 [5], label values [8], escape functions [9], *call-with-current-continuation* (abbreviated as *call/cc*) [2], and *catch* and *throw* [11].

Non-functional control operators "provide a way of pruning unnecessary computation and allow certain computations to be expressed by more com-

[2] C. Talcott about the introduction of *note* into Rum, a lexically-scoped dialect of Lisp [12], p.68.

pact and conceptually manageable programs." [3] If they make continuations available as first-class objects (unlike in COMMON LISP), 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 [4],[6], intelligent backtracking [3], and coroutining [12]. Even though this is widely recognized, control operators are still regarded with skepticism. Their addition seems rather *ad hoc*, since it only advances a particular implementation of the calculus as a programming language but leaves the algebraic side behind. 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 [12]. In this paper we show that the λ-calculus as an equational system can incorporate control operators and that non-functional control may be characterized in a purely syntactic manner.

Since we are interested in reasoning about a call-by-value language, *i.e.* Scheme [2], we use Plotkin's λ-value-calculus [7] as the starting point. We do not expect that the reader knows this variant, but we assume familiarity with the notation and terminology of the conventional λ-calculus [1]. In the next section we extend the basic set of operations by two new ones that give access to and control over the current continuation of a program evaluation. The extended system 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 provides the means to tackle the major goal of this paper: to prove theorems about how to reason with continuations as programming tools. The four theorems of Section 4 show that one can understand access to and resumption of continuations as syntactic operations of terms on their contexts. The last section before the conclusion contains examples demonstrating how to use the theorems.

---

[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 [12], p.16.

2. The λ_c-calculus

Plotkin's λ_v-calculus constitutes the basis of our control calculus, λ_c. For the sake of simplicity we concentrate on constant-free expressions and the β_v-reduction. The inclusion of constants and an associated δ-rule would make the calculus more realistic but not more interesting.

The set of expressions of λ_c, denoted by Λ_c, subsumes the original set of λ-expressions and includes two new types of applications: $\mathcal{C}$- and $\mathcal{A}$-applications. The formal definition of Λ_c is displayed in Definition 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.

The notion of free and bound variables in a term M carries over directly from the *pure* λ-calculus under the provision that $\mathcal{C}$ and $\mathcal{A}$ are symbols which are neither free nor bound. Terms with no free variables are called *closed terms* or *programs*. Since we do not want to get involved in syntactic issues, we adopt Barendregt's convention of identifying terms that are equal modulo 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: $\mathcal{C}$- and $\mathcal{A}$-applications are treated like applications where the function part is simply ignored.

The intention behind the two operations $\mathcal{C}$ and $\mathcal{A}$ can easily be explained informally. $\mathcal{A}$ represents an *abort* or *stop* operation which terminates the program and returns the value of its argument. Whereas some operation like $\mathcal{A}$ is commonly found in traditional languages, $\mathcal{C}$ and its relatives are only available in λ-calculus based languages. It is a form of the *call/cc*-mechanism in Scheme. The operation applies its argument to the current continuation, *i.e.* an abstraction of what has to 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. The continuation is represented by an abstraction; we generally

Definition 1: The term sets Λ_c and Λ

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

- *variables:* x if $x \in V$;
- *abstractions:* $(\lambda x.M)$ if $M \in \Lambda_c$ and $x \in V$;
- *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*.

Λ, the term set of the traditional λ-calculus, stands for Λ_c restricted to variables, applications, and abstractions.

refer to it as a *continuation function*. It 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 a minor point: *call/cc* implicitly invokes the current continuation on the value of its argument; $\mathcal{C}$ leaves this to its argument. Given $\mathcal{C}$, one can define *call/cc* as $\lambda f.\mathcal{C}(\lambda\kappa.\kappa(f\kappa))$.

The formal semantics of Λ_c is defined by a continuation-passing style translation (abbreviated cps) into the λ-calculus:

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

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

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

$$[\![(\mathcal{C}M)]\!] = \lambda\kappa.[\![M]\!](\lambda m.m(\lambda v\kappa'.\kappa v)\mathrm{I}), \qquad (cps4)$$
$$[\![(\mathcal{A}M)]\!] = \lambda\kappa.[\![M]\!]\mathrm{I}. \qquad (cps5)$$

The third equation, $(cps3)$, expresses the left-to-right and by-value evaluation of applications. The equations $(cps4)$ and $(cps5)$ reflect the informal definition of $\mathcal{C}$ and $\mathcal{A}$: $\mathcal{C}$ applies its argument to a functional object, encapsulating the current continuation κ; $\mathcal{A}$ throws away the current continuation.

An alternative, but equivalent definition of $\mathcal{C}$ and $\mathcal{A}$ highlights their application character:

$$[\![\mathcal{C}M]\!] = \lambda\kappa.(\lambda\kappa'.\kappa'\underline{(\lambda f\kappa.(f(\lambda v\kappa'.(\kappa v))\mathrm{I}))})(\lambda f.[\![M]\!](\lambda.fm\kappa)) \qquad (cps4')$$
$$[\![\mathcal{A}M]\!] = \lambda\kappa.(\lambda\kappa'.\kappa'\underline{(\lambda v\kappa.v)})(\lambda f.[\![M]\!](\lambda m.fm\kappa)). \qquad (cps5')$$

The two equations treat $\mathcal{C}$- and $\mathcal{A}$-applications as if they were applications and as if $\mathcal{C}$ and $\mathcal{A}$ were ordinary abstractions in Λ_c which map to the underlined parts, respectively. Yet, it is easy to see that neither $\mathcal{C}$ nor $\mathcal{A}$ are the images of values in Λ_c. Hence, both operations cannot be explained by the ordinary β-reduction.

With these definitions in mind we turn to the reduction rules. First, we recall Plotkin's call-by-value version of the β-rule:

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

Restricted to Λ, it is the basis of the λ_v-calculus, which is an accurate reflection of a higher-order applicative language with a by-value semantics [7]. For the control operations $\mathcal{C}$ and $\mathcal{A}$, we need new reduction rules.

Given the expression $(\mathcal{C}M)N$, we know that M should be applied to a function which simulates the continuation and that the continuation should be replaced by the initial one. The expression $M(\lambda f.(fN))$ almost satisfies the requirement when the application is not nested within other expressions. We need to know the continuation of the entire application in order to send

the result of fN to the rest of the computation. So the reductions are:

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

$$M(\mathcal{C}N) \xrightarrow{C_R} \mathcal{C}\lambda\kappa.N(\lambda v.\kappa(Mv)) \qquad (C_R)$$

provided that M is a value.

To derive the $\mathcal{A}$-reductions, we proceed in the same manner. The $\mathcal{A}$-application must abort all pending computations. Suppose that $(\mathcal{A}M)$ is in the argument position of an application, *e.g.* $N(\mathcal{A}M)$. $\mathcal{A}$ should prohibit this application and make M the result of the program. Since we want reductions that can be applied to subterms, we must assure that M is not only the result of this particular application but also that of the whole expression. $(\mathcal{A}M)$ achieves this effect. The reasoning for the dual case of $(\mathcal{A}M)N$ is similar and so we define the following reductions:

$$(\mathcal{A}M)N \xrightarrow{A_L} \mathcal{A}M, \qquad (A_L)$$

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

provided that M is a value.

So far our relations can deal with programs where $\mathcal{C}$- and $\mathcal{A}$-applications are proper subterms. Next we have to consider occurrences of $\mathcal{C}$- and $\mathcal{A}$-applications at the root of a term, such as $\mathcal{C}\lambda\kappa.\mathbf{K}(\kappa\mathbf{I})$ or $\mathcal{A}(\mathbf{KI})$. The above definitions of $\mathcal{C}$ and $\mathcal{A}$ stipulate that these programs can be further reduced, but neither of the above rules can evaluate them any further. We need two *top-level* evaluation rules.

Intuitively, the program $\mathcal{C}\lambda\kappa.\mathbf{K}(\kappa\mathbf{I})$ is about to grab the current continuation and pass it to $\lambda\kappa.\mathbf{K}(\kappa\mathbf{I})$. But what is the current continuation? In principle, there is nothing left to do after evaluating the $\mathcal{C}$-argument, and that is exactly what we model. The top-level continuation object should, when invoked, stop the evaluation and make its argument the final value of the entire program, *e.g.* $\mathcal{C}\lambda\kappa.\mathbf{K}(\kappa\mathbf{I})$ should evaluate to $\mathbf{I}$. A natural representation for this *abort* continuation is $\lambda x.\mathcal{A}x$. A quick check shows that the

sample program almost reduces to the desired value:

$$\mathcal{C}\lambda\kappa.\mathbf{K}(\kappa\mathbf{I}) \rightarrow (\lambda\kappa.\mathbf{K}(\kappa\mathbf{I}))(\lambda x.\mathcal{A}x) \rightarrow \mathbf{K}((\lambda x.\mathcal{A}x)\mathbf{I}) \rightarrow \mathcal{A}\mathbf{I},$$

except that we still don't know how to evaluate programs of the form $\mathcal{A}\mathbf{I}$.

The case $\mathcal{A}(\mathbf{KI})$ is easy to deal with: the program should abort and deliver the value of the $\mathcal{A}$-argument as the final result. On the other hand, there is nothing else left to do but to evaluate the $\mathcal{A}$-argument. Therefore, it is quite natural to say that $\mathcal{A}(\mathbf{KI})$ results in $\mathbf{KI}$.

Although it seems that we have the basis for an adequate calculus, there is still a problem: neither of the top-level rules is *compatible*[4] with the syntactic constructions. Put differently, the top-level relations are not applicable to subterms. If they were, the equational system would not be confluent. Consider the program $\mathbf{K}(\mathcal{A}\mathbf{I})$: the rule $\mathcal{A}_R$ leads to $\mathcal{A}\mathbf{I}$, which in turn would evaluate to $\mathbf{I}$, but an application of the top-level rule to the subterm $(\mathcal{A}\mathbf{I})$ results in $\mathbf{KI}$ and a final value of $\lambda xy.y$. On the other hand, the top-level relations are needed to reflect the semantics of $\mathcal{C}$ and $\mathcal{A}$. We therefore admit them with a special status: instead of making them first-class reduction rules, we define them to be *computation* rules and indicate this by using $\triangleright$ in place of $\rightarrow$:

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

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

Since we need both reductions and computations for a strong enough calculus, care must be taken in formulating the equivalence relations. Congruence relations are only formed over the reduction relations; the notion of compatibility is extended to $\mathcal{C}$- and $\mathcal{A}$-applications. The result is a subcalculus of λ_c. By throwing in the additional computation rules we obtain the complete control calculus. The formal definition is shown in Definition 2.

[4] [1], p.50

Definition 2: The λ_c-calculus

Let $\xrightarrow{c} = \xrightarrow{\mathcal{C}_L} \cup \xrightarrow{\mathcal{C}_R} \cup \xrightarrow{\mathcal{A}_L} \cup \xrightarrow{\mathcal{A}_R} \cup \xrightarrow{\beta_v}$. Then define the *one step C-reduction* $\rightarrow_c$ as the compatible closure of $\xrightarrow{c}$:

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

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

When we refer to equations in the traditional λ-calculus, we use $=_\beta$ instead of $=_c$ and $=_k$.

We now have an extended λ-calculus programming language that can handle first-class continuations. The meaning of the programs is defined by the cps-transformation; furthermore, we have derived computation and reduction relations that we claim describe equivalences and evaluations among Λ_c-programs. In the next section we investigate the major syntactic proper-

ties of the λ_c-calculus.

3. Fundamental Properties of the λ-calculus

The development of the λ-calculus raises three questions:

— Are the rules sound, *i.e.* do they preserve meaning?

— Is the equational theory consistent?

— And, do the relations define an operational semantics?

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 [7]:

Theorem 1 (Soundness). *Let $L \in \Lambda$ be an abstraction and let $M \in \Lambda_c$ be a closed term, let $\rightarrow$ stand for either $\xrightarrow{C_L}, \xrightarrow{C_R}, \xrightarrow{A_L}$, or $\xrightarrow{A_R}$, and let $\triangleright$ stand for either $\triangleright_C$ or $\triangleright_A$:*

$$\text{If } M \rightarrow N, \text{ then } [\![M]\!]L =_\beta [\![N]\!]L \text{ and,}$$
$$\text{if } M \triangleright N, \text{ then } [\![M]\!]\mathbf{I} =_\beta [\![N]\!]\mathbf{I}.$$

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 C_R and A_R cases one then needs the assumption that L is an abstraction; in all other cases, the respective left-hand and right-hand terms are already equivalent. □

The consistency problem is equivalent to proving the confluence of reductions in $\rightarrow_c$ and $\triangleright_k$, respectively. The proof of the Church-Rosser property for $\xrightarrow{c}$ is an application of Martin-Löf's method for showing the corresponding result for $\xrightarrow{\beta}$. Since our presentation follows rather closely the one of Barendregt, 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 for $\xrightarrow{c}$. For the proof of the standardization theorem we also need to define a notion of

the length or size of a parallel reduction: see Definition 3. Note that if M is a value and $M \underset{1}{\twoheadrightarrow} N$ then N is a value.

The following lemma shows the relationship between $\twoheadrightarrow_c$ and $\underset{1}{\twoheadrightarrow}$. Its proof is obvious and omitted.

Lemma 2. $\overset{c}{\rightarrow} \subset \rightarrow_c \subset \underset{1}{\twoheadrightarrow} \subset \twoheadrightarrow_c$.

Next we need to prove that in $\underset{1}{\twoheadrightarrow}$ 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', respectively. However, for the proof of the standardization theorem we will 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 exact same structure and, therefore,—following Plotkin's example—we have put them together:

Lemma 3. *Suppose $M \underset{1}{\twoheadrightarrow} M'$, $N \underset{1}{\twoheadrightarrow} N'$, and N is a value. Then the following are true:*

(i) $M[x := N] \underset{1}{\twoheadrightarrow} M'[x := N']$

(ii) $s_R = s_{M[x:=N] \underset{1}{\twoheadrightarrow} M'[x:=N']} < s_L = s_{(\lambda x.M)N \underset{1}{\twoheadrightarrow} M'[x:=N']}$

Proof. The proof is a structural induction on the reduction $M \underset{1}{\twoheadrightarrow} M'$. We have omitted the cases which are similar to the given ones or are the same as in Plotkin's proof.

(P1) $M \underset{1}{\twoheadrightarrow} M' \equiv M$.

The result follows by induction on the structure of M. We demonstrate it for the subcase $M \equiv \mathcal{C}P \equiv \mathcal{C}P' \equiv M'$.

(i) $(\mathcal{C}P)[x := N] \equiv (\mathcal{C}P[x := N]) \underset{1}{\twoheadrightarrow} (\mathcal{C}P'[x := N'])$ since P is smaller than M.

(ii)

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

(P3) $M \equiv (\mathcal{A}P)Q \underset{1}{\twoheadrightarrow} M' \equiv \mathcal{A}P'$ and $P \underset{1}{\twoheadrightarrow} P'$.

Definition 3 : The parallel reduction $\underset{1}{\twoheadrightarrow}$.

The *parallel reduction* over Λ_c is denoted by $\underset{1}{\twoheadrightarrow}$. $s_{M \underset{1}{\twoheadrightarrow} N}$ or just s is the function of the *size of the derivations* $M \underset{1}{\twoheadrightarrow} N$. $n(x, M)$ is the number of free occurrences of x in M.

(P1) $$M \underset{1}{\twoheadrightarrow} M, \quad s = 0$$

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

(P3) $$M \underset{1}{\twoheadrightarrow} M' \Rightarrow (\mathcal{A}M)N \underset{1}{\twoheadrightarrow} (\mathcal{A}M'), \quad s = s_{M \underset{1}{\twoheadrightarrow} M'} + 1$$

(P4) $$N \underset{1}{\twoheadrightarrow} N', M \text{ is a value} \Rightarrow M(\mathcal{A}N) \underset{1}{\twoheadrightarrow} \mathcal{A}N', \quad s = s_{N \underset{1}{\twoheadrightarrow} N'} + 1$$

(P5) $$M \underset{1}{\twoheadrightarrow} M', N \underset{1}{\twoheadrightarrow} N' \Rightarrow (\mathcal{C}M)N \underset{1}{\twoheadrightarrow} \mathcal{C}\lambda\kappa.M'(\lambda f.\kappa(fN')), \quad s = s_{M \underset{1}{\twoheadrightarrow} M'} + s_{N \underset{1}{\twoheadrightarrow} N'} + 1$$

(P6) $$M \underset{1}{\twoheadrightarrow} M', N \underset{1}{\twoheadrightarrow} N', \quad M \text{ is a value} \Rightarrow M(\mathcal{C}N) \underset{1}{\twoheadrightarrow} \mathcal{C}\lambda\kappa.N'(\lambda v.\kappa(M'v)), \quad s = s_{M \underset{1}{\twoheadrightarrow} M'} + s_{N \underset{1}{\twoheadrightarrow} N'} + 1$$

(P7) $$M \underset{1}{\twoheadrightarrow} N \Rightarrow \lambda x.M \underset{1}{\twoheadrightarrow} \lambda x.N, \quad s = s_{M \underset{1}{\twoheadrightarrow} N}$$

(P8) $$M \underset{1}{\twoheadrightarrow} N \Rightarrow \mathcal{C}M \underset{1}{\twoheadrightarrow} \mathcal{C}N, \quad s = s_{M \underset{1}{\twoheadrightarrow} N}$$

(P9) $$M \underset{1}{\twoheadrightarrow} N \Rightarrow \mathcal{A}M \underset{1}{\twoheadrightarrow} \mathcal{A}N, \quad s = s_{M \underset{1}{\twoheadrightarrow} N}$$

(P10) $$M \underset{1}{\twoheadrightarrow} M', N \underset{1}{\twoheadrightarrow} N' \Rightarrow MN \underset{1}{\twoheadrightarrow} M'N', \quad s = s_{M \underset{1}{\twoheadrightarrow} M'} + s_{N \underset{1}{\twoheadrightarrow} N'}$$

(i) $((\mathcal{A}P)Q)[x := N] \equiv (\mathcal{A}P[x := N])Q[x := N] \to_1^* \mathcal{A}P'[x := N']$ by inductive hypothesis for $P[x := N] \to_1^* P'[x := N']$.

(ii)

$$\begin{aligned} s_R &= s_{P[x:=N] \to_1^* P'[x:=N']} + 1 \\ &\leq s_{P \to_1^* P'} + n(x, P')s_{N \to_1^* N'} + 1 \text{ by inductive hypothesis} \\ &< (s_{P \to_1^* P'} + 1) + n(x, P')s_{N \to_1^* N'} + 1 = s_L. \end{aligned}$$

(P6) $M \equiv P(\mathcal{C}Q) \to_1^* M' \equiv \mathcal{C}\lambda\kappa.Q'(\lambda v.\kappa(P'v))$ and $P \to_1^* P'$, $Q \to_1^* Q'$, and P is a value.

(i) $(P(\mathcal{C}Q))[x := N] \equiv P[x := N](\mathcal{C}Q[x := N]) \to_1^* \mathcal{C}\lambda\kappa.Q'[x := N'](\lambda v.\kappa(P'[x := N']v))$
by inductive hypothesis for P and Q and the fact that $P[x := N]$ is a value.

(ii)

$$\begin{aligned} s_R &= s_{P[x:=N](\mathcal{C}Q[x:=N]) \to_1^* \mathcal{C}\lambda\kappa.Q'[x:=N'](\lambda v.\kappa(P'[x:=N']v))} \\ &= s_{P[x:=N] \to_1^* P'[x:=N']} + s_{Q[x:=N] \to_1^* Q'[x:=N']} + 1 \\ &\leq s_{P \to_1^* P'} + n(x, P')s_{N \to_1^* N'} + s_{Q \to_1^* Q'} + n(x, Q')s_{N \to_1^* N'} + 1 \\ &\quad \text{by inductive hypothesis for } s_{P[x:=N] \to_1^* P'[x:=N']} \\ &\quad \text{and } s_{Q[x:=N] \to_1^* Q'[x:=N']} \\ &< (s_{P \to_1^* P'} + s_{Q \to_1^* Q'} + 1) + n(x, M')s_{N \to_1^* N'} + 1 = s_L. \end{aligned}$$

(P9) $M \equiv \mathcal{C}P \to_1^* M' \equiv \mathcal{C}P'$ and $P \to_1^* P'$.

(i) $(\mathcal{C}P)[x := N] \equiv \mathcal{C}P[x := N] \to_1^* \mathcal{C}P'[x := N']$ by inductive hypothesis.

(ii)

$$\begin{aligned} s_R &= s_{P[x:=N] \to_1^* P'[x:=N']} \\ &\leq s_{P \to_1^* P'} + n(x, P')s_{N \to_1^* N'} \text{ by inductive hypothesis} \\ &< s_L. \square \end{aligned}$$

In addition to Lemma 3. we have to show that two contractums of $\mathcal{C}_L$- or $\mathcal{C}_R$-redexes reduce to each other in one $\twoheadrightarrow_1$-step if the respective subterms do. Again, the second and fourth claim of the following lemma are actually needed for the standardization theorem:

Lemma 4. *Suppose $M \twoheadrightarrow_1 M'$ and $N \twoheadrightarrow_1 N'$. Then the following are true:*

(i) $\mathcal{C}\lambda\kappa.M(\lambda f.\kappa(fN)) \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.M'(\lambda f.\kappa(fN'))$

(ii) $s_R < s_L$ *where*

$$s_R = s_{\mathcal{C}\lambda\kappa.M(\lambda f.\kappa(fN)) \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.M'(\lambda f.\kappa(fN'))}$$
$$s_L = s_{(\mathcal{C}M)N \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.M'(\lambda f.\kappa(fN'))}.$$

And if M is furthermore a value then

(iii) $\mathcal{C}\lambda\kappa.N(\lambda v.\kappa(Mv)) \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.N'(\lambda v.\kappa(M'v))$

(iv) $s_R < s_L$ *where*

$$s_R = s_{\mathcal{C}\lambda\kappa.N(\lambda v.\kappa(Mv)) \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.N'(\lambda v.\kappa(M'v))}$$
$$s_L = s_{M(\mathcal{C}N) \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.N'(\lambda v.\kappa(M'v))}.$$

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

(i) $N \twoheadrightarrow_1 N'$ hence $\lambda f.(\kappa(fN)) \twoheadrightarrow_1 \lambda f.(\kappa(fN'))$,
$M \twoheadrightarrow_1 M'$ hence $M(\lambda f.(\kappa(fN))) \twoheadrightarrow_1 M'(\lambda f.(\kappa(fN')))$,
and $\lambda\kappa.(M(\lambda f.(\kappa(fN)))) \twoheadrightarrow_1 \lambda\kappa.(M'(\lambda f.(\kappa(fN'))))$,
and therefore $\mathcal{C}\lambda\kappa.M(\lambda f.\kappa(fN)) \twoheadrightarrow_1 \mathcal{C}\lambda\kappa.M'(\lambda f.\kappa(fN'))$.

(ii) $s_R = s_{M \twoheadrightarrow_1 M'} + s_{N \twoheadrightarrow_1 N'} < s_{M \twoheadrightarrow_1 M'} + s_{N \twoheadrightarrow_1 N'} + 1 = s_L$.

The propositions (iii) and (iv) are proven in the same manner. □

Now everything is in place to state and prove the diamond lemma:

Lemma 5. *The relation $\twoheadrightarrow_1$ satisfies the diamond property, i.e. if $M \twoheadrightarrow_1 L_i$ for $i = 1, 2$ then there exists an N such that for each i $L_i \twoheadrightarrow_1 N$.*

Proof. Again, the proof is an induction on the structure of the reduction $M \twoheadrightarrow_1 L_1$. We will 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) $M \equiv P(\mathcal{C}Q) \twoheadrightarrow_1 L_1 \equiv \mathcal{C}\lambda\kappa.Q_1(\lambda v.\kappa(P_1 v))$ and $P \twoheadrightarrow_1 P_1, Q \twoheadrightarrow_1 Q_1$, and P is a value.
There are two possible cases for the reduction from M to L_2 since P is a value and $\mathcal{C}Q$ is not:

a) $L_2 \equiv P_2(\mathcal{C}Q_2)$ and $P \twoheadrightarrow_1 P_2, Q \twoheadrightarrow_1 Q_2$.
Note that P_2 is a value. An application of Lemma 4 gives us $N \equiv \mathcal{C}\lambda\kappa.Q_3(\lambda v.\kappa(P_3 v))$ where P_3 and Q_3 are the terms which must exist for $P \twoheadrightarrow_1 P_i$ and $Q \twoheadrightarrow_1 Q_i$, for $i = 1, 2$ according to the inductive hypothesis.

b) $L_2 \equiv \mathcal{C}\lambda\kappa.Q_2(\lambda v.\kappa(P_2 v))$ and $P \twoheadrightarrow_1 P_2, Q \twoheadrightarrow_1 Q_2$.
Again, an application of Lemma 4 and of the inductive hypothesis for $P \twoheadrightarrow_1 P_i, Q \twoheadrightarrow_1 Q_i, i = 1, 2$ produces terms P_3, Q_3 such that we can take $N \equiv \mathcal{C}\lambda\kappa.Q_3(\lambda v.\kappa(P_3 v))$.

(P10) $M \equiv PQ \twoheadrightarrow_1 L_1 \equiv P_1 Q_1$ and $P \twoheadrightarrow_1 P_1, Q \twoheadrightarrow_1 Q_1$.
This time we have to distinguish 6 possible subcases:

a) $L_2 \equiv R_2[x := Q_2]$ and $P \equiv \lambda x.R, R \twoheadrightarrow_1 R_2, Q \twoheadrightarrow_1 Q_2$, and Q is a value.
But then $P_1 \equiv \lambda x.R_1, R \twoheadrightarrow_1 R_1$, and Q_1 is a value. By inductive hypothesis we must be able to find R_3 and Q_3 such that—using Lemma 3—we can take $N \equiv R_3[x := Q_3]$.

b) $L_2 \equiv \mathcal{A}R_2$ and $P \equiv \mathcal{A}R, R \twoheadrightarrow_1 R_2$.
Again, $P_1 \equiv \mathcal{A}R_1$ and $R \twoheadrightarrow_1 R_1$. By inductive hypothesis we can find an R_3 such that we can take $N \equiv \mathcal{A}R_3$.

c) $L_2 \equiv \mathcal{A}R_2$ and $Q \equiv \mathcal{A}R, R \twoheadrightarrow_1 R_2$, and P is a value.
This case is like b).

d) $L_2 \equiv \mathcal{C}\lambda\kappa.R_2(\lambda f.\kappa(fQ_2))$ and $P \equiv \mathcal{C}R, Q \twoheadrightarrow_1 Q_2, R \twoheadrightarrow_1 R_2$.
Now we must have that $P_1 \equiv \mathcal{C}R_1, R \twoheadrightarrow_1 R_1$. An application of the inductive hypothesis and Lemma 4 quickly shows that we can

take $N \equiv \mathcal{C}\lambda\kappa.R_3(\lambda f.\kappa(fQ_3))$.

e) $L_2 \equiv \mathcal{C}\lambda\kappa.R_2(\lambda v.\kappa(P_2 v))$ and $Q \equiv \mathcal{C}R, P \xrightarrow[1]{} P_2, R \xrightarrow[1]{} R_2$, 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 $\xrightarrow{c}$:

Theorem 6. *The relation $\xrightarrow{c}$ is Church-Rosser.*

Proof. $\twoheadrightarrow_c$ is the transitive closure of $\xrightarrow[1]{}$. Since $\xrightarrow[1]{}$ 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 6 we can easily show that $\triangleright_k$ satisfies the diamond property which is sufficient to establish consistency:

Theorem 7 (Consistency). *The relation $\triangleright_k$ satisfies the diamond property, i.e. if $M \triangleright_k L_i$ for $i = 1, 2$ then there exists an N such that for each i $L_i \triangleright_k N$.*

Proof. We proceed by a case analysis on $M \triangleright_k L_1$.

($\mathcal{A}_T$) $M \triangleright_{\mathcal{A}} L_1$ 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) $M \triangleright_{\mathcal{C}} L_2$.
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 ($\mathcal{A}_T$).

($\twoheadrightarrow_c$) $M \twoheadrightarrow_c L_1$.

Here 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 $\xrightarrow{c}$. □

The proof of the theorem shows that $\rightarrow_c$ and $\triangleright_{\mathcal{C}}$ or $\triangleright_{\mathcal{A}}$, respectively, commute. The theorem establishes the following traditional corollary:

Corollary.

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

(ii) If M has a C-normal form N then $M \triangleright_k^* N$.

(iii) A term has at most one C-normal form.

Proof. (i) relies on the previous theorem. (ii) and (iii) use (i) and the fact that a normal form cannot be further reduced. □

Furthermore, we can now prove:

Theorem 1′ (Incompleteness). *There are M and N such that for all abstractions L, $[M]L =_\beta [N]L$ but $M \neq_k N$.*

Proof. The proposition is a consequence of Theorem 7 and the fact that Plotkin's value calculus is a sub-calculus. An example is given by: $M \equiv (\omega\omega)y$ and $N \equiv (\lambda x.xy)(\omega\omega)$ where $\omega \equiv (\lambda x.xx)$. □

More interesting, from a computational perspective, is the existence of standard reduction sequences since they fix an operational semantics for the programming language independent of a machine. 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 gave an equivalent, but much more elegant and intuitive definition. It requires the definition of a standard reduction function which reduces the first—top-down and left-to-right—redex in a Λ_c-term not inside an abstraction, a $\mathcal{C}$-application, or an $\mathcal{A}$-application. This function is then extended to standard reduction sequences by forming something like a compatible closure: see Definition 4.

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

Definition 4: Standard reduction sequences and functions

The *standard reduction function*, $\mapsto_{sc}$, *for* $\xrightarrow{c}$ is defined:

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

A *standard reduction sequence of type* C, abbreviated C-srs, is defined by:

$$x \in V \Rightarrow x \text{ is a C-srs;}$$
$$N_1, \ldots, N_k \text{ is a C-srs} \Rightarrow$$
$$\lambda x.N_1, \ldots, \lambda x.N_k, \mathcal{C}N_1, \ldots, \mathcal{C}N_k, \text{ and } \mathcal{A}N_1, \ldots, \mathcal{A}N_k \text{ are C-srs's;}$$
$$M \mapsto_{sc} N_1, \text{ and } N_1, \ldots, N_k \text{ is a C-srs} \Rightarrow M, N_1, \ldots, N_k \text{ is a C-srs}$$
$$M_1, \ldots, M_j \text{ and } N_1, \ldots, N_k \text{ are C-srs's} \Rightarrow$$
$$M_1N_1, \ldots, M_jN_1, \ldots, M_jN_k \text{ is a C-srs.}$$

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

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

A *standard reduction sequence of type* K, K-srs, is defined by:

$$N_1, \ldots, N_k \text{ is a C-srs} \Rightarrow N_1, \ldots, N_k \text{ is a K-srs;}$$
$$M \mapsto_{sk} N_1 \text{ and } N_1, \ldots, N_k \text{ is a K-srs} \Rightarrow M, N_1, \ldots, N_k \text{ is K-srs.}$$

The notation $\mapsto_{sk}^{+}$ and $\mapsto_{sk}^{*}$ stands for the transitive and transitive-reflexive closure of $\mapsto_{sk}$, respectively; $\mapsto_{sk}^{i}$ indicates i applications of $\mapsto_{sk}$, $M \mapsto_{sk}^{\{i,j\}} N$ means that either i or j applications of $\mapsto_{sk}$ reduce M to N.

Theorem 8 (Standardization). *$M \triangleright_k^* N$ if and only if there exists a K-srs $L_1, \ldots, L_n$ 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 the $\rightarrow_c$-reduction. Second, we give a method to reshuffle $\triangleright_k$-reduction sequences into K-srs's. It is based on the first standardization theorem and the fact that the top-level reductions and $\rightarrow_c$ commute. The standardization theorem for the $\rightarrow_c$-reductions is:

Theorem 9. *$M \twoheadrightarrow_c N$ 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 $\twoheadrightarrow_1$. This follows from Lemma 2. Then we show with the following lemma that one can recursively transform the resulting sequence of $\twoheadrightarrow_1$ reductions into a C-srs. □

Lemma 10. *If $M \twoheadrightarrow_1 N_1$ 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, the length of the C-srs $N_1, \ldots, N_j$, on the size of the proof $M \twoheadrightarrow_1 N_1$, and on the structure of M. We proceed by a case analysis on the last step in $M \twoheadrightarrow_1 N_1$ and omit all the cases which are similar to the presented ones or which are treated by Plotkin:

(P3) $M \equiv (\mathcal{A}P)Q \twoheadrightarrow_1 N_1 \equiv \mathcal{A}P_1$ and $P \twoheadrightarrow_1 P_1$.
But then we also have $M \mapsto_{ec} \mathcal{A}P$ and $\mathcal{A}P \twoheadrightarrow_1 \mathcal{A}P_1$ by a proof which is shorter than the proof $M \twoheadrightarrow_1 N_1$. 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) $M \equiv P(\mathcal{C}Q) \twoheadrightarrow_1 N_1 \equiv \mathcal{C}\lambda\kappa.Q_1(\lambda v.\kappa(P_1 v))$ and $P \twoheadrightarrow_1 P_1, Q \twoheadrightarrow_1 Q_1$, and P is a value.
Again, M can immediately be reduced to $\mathcal{C}\lambda\kappa.Q(\lambda v.\kappa(Pv))$ by $\mapsto_{ec}$.

By Lemma 4 we know that

$$\mathcal{C}(\lambda\kappa.(Q(\lambda v.(\kappa(Pv)))))\twoheadrightarrow_1 \mathcal{C}\lambda\kappa.Q_1(\lambda v.\kappa(P_1 v))$$

by a proof that is shorter than the one for $M\twoheadrightarrow_1 N_1$. Therefore, by inductive hypothesis, we can find a C-srs from $\mathcal{C}\lambda\kappa.Q_1(\lambda v.\kappa(P_1 v))$ to N_j from which we build the required C-srs from M to N_j.

(P9) $M \equiv \mathcal{C}P \twoheadrightarrow_1 M' \equiv \mathcal{C}P'$ and $P\twoheadrightarrow_1 P'$.
Here $N_i \equiv \mathcal{C}N_i'$ for all $i, 1 \le i \le j$. Now consider $P\twoheadrightarrow_1 N_1', \ldots, N_j'$. 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 into $(\mathcal{C}\ \_)$ we have constructed the required reduction sequence.

(P10) This case does not differ from Plotkin's corresponding case but it requires that $M\twoheadrightarrow_1 N \mapsto_{sc} L$ can be transformed into $M \mapsto_{sc} K\twoheadrightarrow_1 L$ for some appropriate term K. This is proven in a separate lemma. □

The next lemma shows that $\twoheadrightarrow_1$ and $\mapsto_{sc}$ commute as required by Lemma 10, case (P10):

Lemma 11. *If $M\twoheadrightarrow_1 M' \mapsto_{sc} M''$ then there exists an L such that $M \mapsto_{sc} L\twoheadrightarrow_1 M''$.*

Proof. Plotkin's proof—an induction on $M\twoheadrightarrow_1 M'$—of his Lemma 8, section IV goes through with almost no change. For the cases (P5) and (P6) we need Lemma 4 but structurally they are just like case (P2). Case (P10)—$M \equiv (PQ)\twoheadrightarrow_1 M' \equiv (P'Q')$— deserves some explanation. Consider the following subcases:

a) $M' \equiv (\mathcal{A}P_1')Q' \mapsto_{sc} M'' \equiv \mathcal{A}P_1'$.
So we have $P\twoheadrightarrow_1 \mathcal{A}P_1'$. But then we claim that there exists an L such that $P \mapsto_{sc}^* L\twoheadrightarrow_1 \mathcal{A}P_1'$ and L is an $\mathcal{A}$-application. With the rules for $\mapsto_{sc}$ we get that $(PQ) \mapsto_{sc}^* (LQ) \mapsto_{sc} L\twoheadrightarrow_1 \mathcal{A}P_1' \equiv M''$.

b) $M' \equiv (\mathcal{C}P_1')Q' \mapsto_{sc} M'' \equiv \mathcal{C}\lambda\kappa.P_1'(\lambda f.\kappa(fQ'))$.
We proceed just like in a). Given that $P\twoheadrightarrow_1 \mathcal{C}P_1'$ we again claim that

there is an L such that $P \mapsto_{sc}^{*} L \twoheadrightarrow_{1} \mathcal{C}P_1'$ and L is a $\mathcal{C}$-application. The rest is similar to a).

All other cases for $\mathcal{C}$- and $\mathcal{A}$-applications are treated similarly. □

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 $M \twoheadrightarrow_{1} (\mathcal{A}N)$ where M is an application then there exists an L which is a $\mathcal{A}$-application and $M \mapsto_{sc}^{+} L \twoheadrightarrow_{1} (\mathcal{A}N)$.*

Lemma. *If $M \twoheadrightarrow_{1} (\mathcal{C}N)$ where M is an application then there exists an L which is a $\mathcal{C}$-application and $M \mapsto_{sc}^{+} L \twoheadrightarrow_{1} (\mathcal{C}N)$.*

Lemma. *If $M \twoheadrightarrow_{1} (\lambda x.N)$ where M is an application then there exists an L which is an abstraction and $M \mapsto_{sc}^{+} L \twoheadrightarrow_{1} (\lambda x.N)$.*

Lemma. *If $M \twoheadrightarrow_{1} x$ where x is a variable then $M \mapsto_{sc}^{+} x$.*

Equipped with this first standardization theorem for $\rightarrow_c$, it is easy to finish the proof of Theorem 8. But before, we need to clarify one more fact:

Lemma 12. *If $N_1, \ldots, N_k$ is a C-srs then there exists a $j, 1 \leq j < k$ such that for all $i, 1 \leq i < j, N_i \mapsto_{sc} N_{i+1}$, and for all $i, j \leq i < k, N_i \rightarrow_c N_{i+1}$ and $N_i \not\mapsto_{sc} N_i + 1$.*

Proof. A straightforward induction on k. □

And here is finally the proof of the main result of this section:

Proof of Theorem 8. The proof is an induction on the number of computations, $\triangleright_{\mathcal{C}}$ and $\triangleright_{\mathcal{A}}$, used in the evaluation of M to N. If there are no top-level 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 $\triangleright_{\mathcal{A}}$. Then we have the following situation:

$$M \equiv M_1 \triangleright_k \ldots \triangleright_k M_k \equiv \mathcal{A}M_k' \triangleright_{\mathcal{A}} M_{k+1} \equiv M_k' \triangleright_k \ldots \triangleright_k M_n \equiv N.$$

By forming the C-srs for the reduction from M_1 to M_k we get by Lemma 12:

$$M \equiv M_1 \mapsto_{sc} \ldots \mapsto_{sc} M_l \equiv \mathcal{A}M_l' \to_c \ldots \to_c M_k \equiv \mathcal{A}M_k'$$
$$\triangleright_{\mathcal{A}} M_{k+1} \equiv M_k' \triangleright_k \ldots \triangleright_k M_n \equiv N.$$

Now, since $\to_c$ and $\triangleright_{\mathcal{A}}$ commute—see the remark following Theorem 6—we can move the top-level computation forward:

$$M \equiv M_1 \mapsto_{sc} \ldots \mapsto_{sc} M_l \equiv \mathcal{A}M_l' \triangleright_{\mathcal{A}} M_l' \to_c \ldots \to_c M_k' \triangleright_k \ldots \triangleright_k M_n \equiv N.$$

By inductive hypothesis we get a similar reduction sequence for the reduction from M_l' to N. Since $M \mapsto_{sk}^{+} M_l'$ we can form the desired K-srs from M to N.

The dual case of $\triangleright_{\mathcal{C}}$-computations is treated similarly. □

Beyond the satisfaction of a theoretical need, standard reduction sequences are interesting from a practical point of view. A standard reduction function determines an operational semantics for the programming language of the calculus. A chain of $\mapsto_{sk}$-applications leads from the program to its value if and only if the program is reducible to a value. We therefore consider a series of $\mapsto_{sk}$-applications to a program as an evaluation. In the next section we study the behavior of continuations with respect to evaluations.

4. A syntactic characterization of continuations

All previous attempts to reason about the usage of continuation functions in programs relied upon a cps-like interpretation of programs [3], [12]. The λ_c-calculus and its standard reduction semantics allows us to understand the labeling and invocation of continuations in terms of symbolic evaluations. However, it is obviously quite difficult to work directly with the calculus and the sk-evaluation function. The basic system not only has seven reduction rules but one must also keep in mind that two of them are only applicable to the root of the term. Fortunately, we can prove some regularities about the evaluation of $\mathcal{C}$- and $\mathcal{A}$-redexes which can be formed into meta-reduction

Definition 5: Sk-redexes, the depth of a redex, and sk-contexts.

The *sk-redex of a term* M and its *depth* d_M^{sk} is defined as:

(SK1) M if M is a $\mathcal{C}_T$- or an $\mathcal{A}_T$-redex and $d_M^{sk} = 0$;

(SK2) P if P is the sc-redex of M and $d_M^{sk} = d_M^{sc}$.

The *sc-redex of a term* M and its *depth* d_M^{sc} is defined as:

(SC1) M if M is a $\xrightarrow{c}$-redex and $d_M^{sc} = 1$;

(SC2) P if $M \equiv KL$, K is an application and P is the sc-redex of K and $d_M^{sc} = d_K^{sc} + 1$;

(SC3) P if $M \equiv KL$, K is a value and P is the sc-redex of L and $d_M^{sc} = d_L^{sc} + 1$.

An *sk-context* is a term with one hole:

(C1) [ ] is an sk-context, or

(C2) if $C[\ ]$ is an sk-context and P is any Λ_c-term, then $C[\ ]P$ is an sk-context, or

(C3) if $C[\ ]$ is an sk-context and P is a value, then $PC[\ ]$ is an sk-context.

rules. These meta-rules are simple and greatly facilitate the reasoning with continuations. First we need more terminology to talk about evaluations with the sk-function.

Given a term, the sk-function picks *the* outermost C-redex not inside an abstraction and reduces it. The textual context of this redex stays the same. We refer to this outermost redex for the sk-function as the *sk-redex*; its *depth* is the distance from the root of the term to the redex. Both notions are formally defined in Definition 5. We sometimes prefix sk-redex with $\mathcal{C}$, $\mathcal{A}$, or β for clarification.

The concept of a textual context of a redex is formalized in the notion of a Λ_c-*sk-context* which is a Λ_c-term with a hole; see Definition 5. $C[\ ], C'[\ ]$, *etc.* denote contexts. $C[M]$ is the term where the hole of the context $C[\ ]$ is filled with M. If $C[M]$ contains an sk-redex, the redex must be in M or M is at least a part of the redex. Although in general, free variables of M may become bound through the filling-in operation for contexts, this is not true for contexts. By definition of $\mapsto_{sk}$ an sk-redex cannot be within the scope of an abstraction. The next two lemmas connect contexts with sk-reductions.

Lemma 1. *If $M \mapsto_{sk} N$ then for some sk-context $C[\ ]$, $M \equiv C[P], N \equiv C[Q]$, and $P \to_c Q$, $P \rhd_{\mathcal{C}} Q$, or $P \rhd_{\mathcal{A}} Q$. Similarly for $\mapsto_{sc}$.*

The proof of this first lemma is trivial and omitted.

Lemma 2. *Let $C[\ ]$ be an sk-context and let $\otimes$ stand for either $\mathcal{A}$ or $\mathcal{C}$, and let $Q \equiv (\otimes R)$ for some R. Then*

(i) $C[Q]$ contains an $\otimes$-sk-redex

(ii) Q is the $\otimes$-application part of $C[Q]$'s sk-redex, and,

(iii) if $P \xrightarrow{\otimes_l} Q$ or $P \xrightarrow{\otimes_r} Q$, then $d^{sk}_{C[Q]} < d^{sk}_{C[P]}$.

Proof. (i) and (ii) are trivial. The argument for (iii) is an induction on the structure of the sk-context $C[\ ]$. To make the proof more readable, define $M \equiv C[P]$ and $N \equiv C[Q]$.

(C1) $M \equiv P$. But then $N \equiv Q \equiv (\otimes R)$ and N is a $\otimes$-sk-redex with $d^{sk}_N = 0$.

(C2) $M \equiv M_1 M_2, M_1$ is an application and contains M's sc-redex. By inductive hypothesis $M_1 \to_c N_1$ such that N_1 contains the $\otimes$-sk-redex with $d^{sk}_{N_1} < d^{sk}_{M_1}$. If $N_1 \equiv Q$ then $N \equiv QM_2$ is the sk-redex we are looking for and $d^{sk}_N = 1 < d^{sk}_M$. Otherwise there is an sk-context $C'[\ ] \neq [\ ]$ such that $M_1 \equiv C'[P]$ and $N_1 \equiv C'[Q]$. Since $C'[\ ]$ does not change during the reduction, $N \equiv N_1 M_2$ contains its redex in N_1, the redex in N_1 is of the desired form, $d^{sk}_N = d^{sk}_{N_1} + 1 < d^{sk}_{M_1} + 1 = d^{sk}_M$, and the case is finished.

(C3) $M \equiv M_1 M_2, M_1$ is a value and M_2 contains M's sc-redex. This case is just like (C2). □

Equipped with this lemma we can prove that $\mathcal{A}$ behaves like an *abort* operation. Once an $\mathcal{A}$-redex becomes the sk-redex, the $\mathcal{A}$-argument is the final result:

Theorem 3. *Let $M \equiv C_M[\mathcal{A}L]$ for some term L and sk-context $C_M[\ ]$, i.e. M's sk-redex is an $\mathcal{A}$-redex whose $\mathcal{A}$-application is $\mathcal{A}L$. Then M evaluates to the $\mathcal{A}$-argument, i.e., $M \mapsto_{sk}^{+} L$.*

Proof. The proof is an induction on the depth of the sk-redex and uses Lemma 2 for the induction step. □

The situation with a $\mathcal{C}$-sk-redex is similar. Sample evaluations indicate that the respective $\mathcal{C}$-application is removed from a term and that at the same time the current continuation is computed from the context of the $\mathcal{C}$-application. More precisely, a $\mathcal{C}$-sk-redex causes a sequence of evaluation steps with two halves: a construction phase, where the continuation is built piecemeal, and a collection phase, where the fragments are put together.

The construction phase propagates the $\mathcal{C}$-application from deep inside the term to the root via $\mathcal{C}_R$- and $\mathcal{C}_L$-reductions. Unlike with an $\mathcal{A}$-application, this process does not remove but rewrites the context. In case the $\mathcal{C}$-application is part of a $\mathcal{C}_L$-redex, say $(\mathcal{C}P)Q$, it builds a *continuation fragment* of the form $\lambda f.\kappa(fQ)$ and applies P to it. The free variable κ hereby represents the rest of the continuation which is yet to be computed from the rest of the context. The symmetric case of a $\mathcal{C}_R$-redex is dealt with in a similar manner. When the $\mathcal{C}$-application finally reaches the root of the term, it is removed, and its argument is applied to $\lambda x.\mathcal{A}x$.

The collection phase is just a series of β_v-reductions. The argument is always the continuation which has been built up so far; the function part matches either $\lambda\kappa.P(\lambda f.\kappa(fQ))$ or $\lambda\kappa.Q(\lambda v.\kappa(Pv))$. In both cases, the β_v-reduction results in the application of a former $\mathcal{C}$-argument to its current continuation. The collection phase comes to an end when the argument of

the original $\mathcal{C}$-redex is reached. Then the current continuation has been computed and is passed to the expression which requested it.

Since every single step of the construction and collection phase is determined by the structure of the context around the $\mathcal{C}$-application, one can define a function which computes the current continuation of a term M with respect to its $\mathcal{C}$-redex:

$$\begin{aligned}
&[\![(\mathcal{C}P), \kappa]\!]_c = \kappa \\
&[\![(PQ), \kappa]\!]_c = [\![P, \lambda f.\kappa(fQ)]\!]_c \text{ where } P \text{ is an application} \\
&[\![(PQ), \kappa]\!]_c = [\![Q, \lambda v.\kappa(Pv)]\!]_c \text{ where } P \text{ is a value.}
\end{aligned}$$

This function simulates in a top-down fashion the chain of $\mathcal{C}$-reductions in a construction phase. At the same time it collects all the pieces in its second argument which represents the continuation that has been built up so far. If the function's second parameter is the initial continuation $\lambda x.\mathcal{A}x$, the result is indeed the current continuation requested by the $\mathcal{C}$-sk-redex:

Theorem 4. *Let $M \equiv C_M[\mathcal{C}L]$ for some term L and sk-context $C_M[\ ]$, i.e. M's sk-redex is a $\mathcal{C}$-redex whose $\mathcal{C}$-application is $\mathcal{C}L$. Then the current continuation of $\mathcal{C}L$ is $[\![M, \lambda x.\mathcal{A}x]\!]_c$ and $M \mapsto_{sk}^{+} L[\![M, \lambda x.\mathcal{A}x]\!]_c$.*

Proof. We prove our claim by induction on the depth of the sk-redex in M. Assume $d_M^{sk} = 0$. Then $M \equiv \mathcal{C}L \mapsto_{sk}^{+} L(\lambda x.\mathcal{A}x)$ and $[\![M, \lambda x.\mathcal{A}x]\!]_c = \lambda x.\mathcal{A}x$.

Otherwise $d_M^{sk} > 0$. Without loss of generality assume the redex is a $\mathcal{C}_L$-redex. By Lemma 1 and Lemma 2 we know that for some sk-context $C[\ ]$,

$$M \equiv C[(\mathcal{C}L)Q] \mapsto_{sk} N \equiv C[\mathcal{C}\lambda\kappa.L(\lambda f.\kappa(fQ))]$$

that $\mathcal{C}\lambda\kappa.L(\lambda f.\kappa(fQ))$ is the $\mathcal{C}$-application of a new $\mathcal{C}$-sk-redex in N, and

that $d_N^{ek} < d_M^{ek}$. Hence, we can apply the inductive hypothesis to N and get:

$$\begin{aligned} N &\mapsto_{ek}^{+} (\lambda\kappa.L(\lambda f.\kappa(fQ)))[\![N, \lambda x.\mathcal{A}x]\!]_c \\ &\mapsto_{ek} L(\lambda f.[\![N, \lambda x.\mathcal{A}x]\!]_c(fQ)) \\ &\qquad \text{since } [\![\_, \_]\!]_c \text{ is always a value} \\ &= L[\![M, \lambda x.\mathcal{A}x]\!]_c \\ &\qquad \text{by Lemma 6 below.} \square \end{aligned}$$

From this theorem we can immediately deduce a corollary about the structure of continuation functions:

Corollary 5. *All continuation functions have one of the following forms:*

K1: $\lambda x.\mathcal{A}x$

K2: $\lambda v.K(Mv)$ *where* K *is a continuation function and* M *is a value*

K3: $\lambda f.K(fM)$ *where* K *is a continuation function and* M *is an aribtrary term.*

The proof of Theorem 4 depends on:

Lemma 6. *Let* $C[\ ]$ *be an sk-context.*

(i) If $M \equiv C[(\mathcal{C}P)Q]$, $N \equiv C[\mathcal{C}\lambda\kappa.P(\lambda f.\kappa(fQ))]$, *then* $[\![M, \lambda x.\mathcal{A}x]\!]_c = \lambda f.[\![N, \lambda x.\mathcal{A}x]\!]_c(fQ)$.

(ii) If $M \equiv C[P(\mathcal{C}Q)]$, $N \equiv C[\mathcal{C}\lambda\kappa.Q(\lambda v.\kappa(Pv))]$, *and* P *is a value, then* $[\![M, \lambda x.\mathcal{A}x]\!]_c = \lambda v.[\![N, \lambda x.\mathcal{A}x]\!]_c(Pv)$.

Proof. Let us assume for the moment that $[\![C[P], \kappa]\!]_c = [\![P, [\![C[\mathcal{C}X], \kappa]\!]_c]\!]_c$ holds for all P and any arbitrary term X. Then we can establish the claims by straightforward calculations:

(i)

$$\begin{aligned}
[\![M, \lambda x.\mathcal{A}x]\!]_c &= [\![C[(\mathcal{C}P)Q], \lambda x.\mathcal{A}x]\!]_c \\
&= [\![(\mathcal{C}P)Q, [\![C[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c]\!]_c \\
&\qquad \text{for an arbitrary term } X \\
&= [\![\mathcal{C}P, \lambda f.[\![C[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c(fQ)]\!]_c \\
&= \lambda f.[\![C[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c(fQ) \\
&= \lambda f.[\![C[\mathcal{C}\lambda\kappa.P(\lambda f.\kappa(fQ))], \lambda x.\mathcal{A}x]\!]_c(fQ) \\
&\qquad \text{since } X \text{ is arbitrary} \\
&= \lambda f.[\![N, \lambda x.\mathcal{A}x]\!]_c(fQ).
\end{aligned}$$

(ii) Similarly.

Our auxiliary claim is verified by induction on the structure of $C[\ ]$:

(C1) $C[\ ] = [\ ]$. Then $[\![C[P], \kappa]\!]_c = [\![P, \kappa]\!]_c = [\![P, [\![\mathcal{C}X, \kappa]\!]_c]\!]_c$.

(C2) $C[\ ] = C'[\ ]Q$ for some sk-context $C'[\ ]$. Now we have:

$$\begin{aligned}
[\![C'[P]Q, \kappa]\!]_c &= [\![C'[P], \lambda f.\kappa(fQ)]\!]_c \\
&= [\![P, [\![C'[\mathcal{C}X], \lambda f.\kappa(fQ)]\!]_c]\!]_c \\
&\qquad \text{by inductive hypothesis} \\
&= [\![P, [\![C'[\mathcal{C}X]Q, \kappa]\!]_c]\!]_c \\
&= [\![P, [\![C[\mathcal{C}X], \kappa]\!]_c]\!]_c.
\end{aligned}$$

(C3) Similar to (C2).□

Theorem 4 and Corollary 5 characterize how programs label continuations and how these continuation functions are constructed. Every continuation is built inductively and always contains an $\mathcal{A}$-application at the bottom. Hence, if a continuation is invoked and if it ever reaches its bottom in the course of its evaluation, the context of the invocation is—according to Theorem 3—irrelevant. Alternatively, one of the continuation fragments could go into an infinite loop, grab a continuation, or abort the computation. In the first and the third case the continuation invocation is again independent of its context. The second case needs some investigation.

As we know from Theorem 4, the process of computing the current continuation depends on the context of the $\mathcal{C}$-application. We cannot expect *a priori* that the invocation can forget its context in this particular case. On the other hand, from Theorem 4 we also know that with or without context the evaluation results in an application of the $\mathcal{C}$-argument to a new continuation. The $\mathcal{C}$-arguments are the same in either case; the newly generated continuation functions are different. The question is whether the difference is significant.

An example will shed some light on the situation. Suppose the continuation $K \equiv \lambda v.K^*((\lambda y.\mathcal{C}y)v)$ is about to be applied to a value Q in the sk-context $C[\ ]$. The evaluation proceeds as follows:

$$\begin{aligned} C[KQ] &\mapsto_{sk} C[K^*((\lambda y.\mathcal{C}y)Q)] \\ &\mapsto_{sk} C[K^*(\mathcal{C}Q)] \\ &\mapsto^+_{sk} Q(\lambda v.[\![C[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c(K^*v)). \end{aligned}$$

Without context the intermediate result would look different:

$$\begin{aligned} KQ &\mapsto_{sk} K^*((\lambda y.\mathcal{C}y)Q) \\ &\mapsto_{sk} K^*(\mathcal{C}Q) \\ &\mapsto^+_{sk} Q(\lambda v.(\lambda x.\mathcal{A}x)(K^*v)). \end{aligned}$$

However, one can see that the two new continuations would *behave* similarly if they were invoked by Q. Both would immediately call the continuation K^* and, if this continuation reaches its bottom, the two expressions yield final results which are equivalent up to the continuations which occur in them. Thus—by an induction from this specific case—if the final result does not contain continuations, the two results are the same and the context of a continuation invocation does not make any difference at all:

Theorem 7. *Let $M \equiv C_M[KL]$ for some value L, continuation K, and sk-context $C_M[\ ]$, and let N be a value which does not contain a continuation as a subterm:*

$$KL \mapsto^+_{sk} N \text{ if and only if } M \mapsto^+_{sk} N.$$

Since the proof is rather long and tedious, we first outline the proof strategy. We would like to prove the theorem by an induction on the unique number of steps in the respective evaluations or, equivalently, by an induction on the structure of the continuation K. But the statement is too weak to serve as an inductive assumption.

For a motivation of our strategy it is instructional to see why the induction fails. We resume the investigation of the above example since it resembles the crucial instance of the induction step. Given the intermediate results QK_1 and QK_2 where $K_1 \stackrel{df}{=} \lambda v.[\![C[\mathcal{C}Q], \lambda x.\mathcal{A}x]\!]_c(K^\bullet v)$ and $K_2 \stackrel{df}{=} \lambda v.(\lambda x.\mathcal{A}x)(K^\bullet v)$, one sees immediately that the final results of the two expressions will be the same if Q does not invoke its argument and the final result does not contain a continuation.

The interesting case is when Q does invoke its argument at some point. Then the two respective evaluations must look like:

$$QK_i \mapsto_{sk}^{+} C_i[K_iQ_i] \quad \text{for } i = 1, 2$$

where the $C_i[\ ]$'s are sk-contexts and the Q_i's are values. The arguments Q_1 and Q_2 may differ in their occurrences of K_1 and K_2, respectively. We recall from the above discussion that the two continuations behave similarly. Both immediately invoke the continuation $K^\bullet$:

$$C_1[K_1Q_1] \mapsto_{sk} C_1[[\![\mathcal{C}Q, \lambda x.\mathcal{A}x]\!]_c(K^\bullet Q_1)]$$

and

$$C_2[K_2Q_2] \mapsto_{sk} C_2[(\lambda x.\mathcal{A}x)(K^\bullet Q_2)].$$

Now suppose we were proving the direction from right to left. Then the two expressions would almost satisfy the inductive hypothesis: $K^\bullet Q_1$ in some context reduces to N and $K^\bullet$ is clearly smaller than K. However, the statement of Theorem 7 as inductive hypothesis would only allow us to conclude that $K^\bullet Q_1$ in the empty context $[\ ]$ reduces to N. It says nothing about the case where the continuation is invoked on a slightly different value

in a non-empty context. This would not hurt too much if we could first prove the direction from left to right, but, as the reader may check for himself, the induction for this direction also fails in this case.

We need to strengthen the inductive hypothesis so that it takes this situation into account. To this end we formalize the concept of behavioral equivalence in order to compare the continuations K_1 and K_2 and the terms Q_1 and Q_2 which differ by occurrences of K_1 and K_2. Given this notion, we can rephrase our theorem so that it is strong enough for an inductive proof:

> If a continuation is invoked on a value in some sk-context and the entire term evaluates to some continuation-free value, then the invocation yields the same result independent of this particular context for all behaviorally equivalent invocation arguments.

It is obvious that this hypothesis takes care of the above example and that it implies both directions of Theorem 7.

Definition 6a: Behavioral Equivalence of Continuations

Two *continuations* K and K^* are *behaviorally equivalent* iff there exists a continuation K' and two sk-contexts $C[\ ]$ and $C^*[\ ]$ such that for all values L,

$$KL \mapsto_{sk}^{\{0,1\}} C[K'L] \text{ and } K^*L \mapsto_{sk}^{\{0,1\}} C^*[K'L].$$

Notation: $K \approx_c K^*$ or $K \approx_c K^* \ (mod\ K')$ if we want to indicate the common continuation.

Two continuations are said to be behaviorally equivalent if they are the same or if they invoke the same continuation after *one* reduction step. The definition is formalized in Definition 6a. An alternative characterization is captured by:

Corollary 8. *Let K, K^*, and K' be continuations. $K \approx K^* \,(mod\, K')$ if and only if for some K_1 and K_1^**

- $K \equiv K^* \equiv K$, *or*
- $K \equiv \lambda v.K_1(K'v)$ *and* $K^* \equiv K'$, *or*
- $K \equiv K'$ *and* $K^* \equiv \lambda v.K_1^*(K'v)$, *or*
- $K \equiv \lambda v.K_1(K'v)$ *and* $K^* \equiv \lambda v.K_1^*(K'v)$.

The extension of the equivalence relation to terms is shown in Definition 6b. Informally, two terms are behaviorally equivalent if they are the same up to occurrences of continuations which must be behaviorally equivalent w.r.t. $\approx_c$. It is obvious that continuations which are equivalent under $\approx_c$ are also behaviorally equivalent as terms but the converse is false. Continuations which are $\approx_t$-equivalent must be the same modulo some continuation fragments which may never get invoked or only after many evaluation steps. In any case, the $\approx_t$-equivalent continuations also behave equivalently. We can therefore confuse the two notions whenever a distinction is unnecessary.

Before we can tackle our main lemma, we need to verify that behavioral equivalence is preserved by sk-evaluations until one of the common continuations is invoked and that common continuations are therefore invoked on behaviorally equivalent values:

Lemma 9. *Suppose $M_1 \approx M_2 \,(mod\, K_1, \ldots, K_n)$ and that $M_1 \mapsto_{sk} M_2$ such that none of the K_i's gets invoked. Then we have either*

- $N_1 \mapsto_{sk} N_2$ *and* $N_2 \approx M_2$, *or*
- *for some K_i, sk-contexts $C_M[\ ]$, $C_N[\ ]$, and values L_M, L_N such that $L_M \approx L_N$, M_2 invokes some K_i, i.e. $M_2 \equiv C_M[K_iL_M]$, and either $N_1 \equiv C_N[K_iL_N]$ or $N_1 \mapsto_{sk} N_2 \equiv C_N[K_iL_N]$.*

Proof. By Lemma 1, the reduction has to be a primitive step within an sk-context. Hence, the claim needs to be checked for all seven base cases of $\mapsto_{sk}$. All but the $\xrightarrow{\beta_v}$-step are trivial, since they only re-arrange equivalent subterms. If none of the continuations which are different in M_1 and N_1 is

Definition 6b: Behavioral equivalence of terms

Two *terms* M and M^* are *behaviorally equivalent*

— if $M \equiv K$ and $M^* \equiv K^*$ for two continuations K and K^* such that $K \approx_c K^* \,(mod\, K')$, or,

— if $M \equiv x \equiv M^*$ for some variable x, or,

— if there are terms M_i, M_i^* such that M_i is behaviorally equivalent to M_i^* for $i = 1, 2$ and

- $M \equiv M_1 M_2$, $M^* \equiv M_1^* M_2^*$, or
- $M \equiv \lambda x.M_1$, $M^* \equiv \lambda x.M_1^*$, or
- $M \equiv \mathcal{C} M_1^*$, $M^* \equiv \mathcal{C} M_1^*$, or
- $M \equiv \mathcal{A} M_1$, $M^* \equiv \mathcal{A} M_1^*$.

Notation: $M \approx_t M^*$ or $M \approx_t M^* \,(mod\, K_1', \ldots, K_n')$ where the K_i's are the common sub-continuations of the continuations in M and M^* which are behaviorally equivalent w.r.t. $\approx_c$.

invoked, the case is characterized by the following matrix:

$$\begin{array}{ccc} M_1 \equiv C_M[(\lambda x.P)R] & \xrightarrow{\beta_x} & C_M[P[x := R]] \equiv M_2 \\ \approx & & \approx \\ N_1 \equiv C_N[(\lambda x.Q)S] & \xrightarrow{\beta_x} & C_N[Q[x := S]] \equiv N_2 \end{array}$$

with $P \approx Q$ and $R \approx S$. The claim that substitution preserves behavioral equivalence is easily proved by an induction on the structure of a term.

Otherwise, assume that $M_1 \equiv C'_M[(\lambda v.K_M(K_i v))L_M]$ and $M_2 \equiv C'_M[K_M(K_i L_M)]$ for some sk-context $C'_M[\ ]$. Since N must contain a behaviorally equivalent continuation in an equivalent sk-context, by Corollary 8 it must be the case that either $N_1 \equiv C'_N[(\lambda v.K_N(K_i v))L_N]$ or $N_1 \equiv C'_N[K_i L_N]$ for some sk-context $C'_N[\ ]$ and value L_N. Since continuation functions can

only occur within L_N or outside of it, L_N is equivalent to L_M. In both cases, the conclusion is immediate. □

The lemma generalizes to longer evaluation sequences:

Corollary 10. *Suppose* $M_1 \approx N_1 \,(mod\, K_1, \ldots, K_n)$.

(i) If $M_1 \mapsto_{ok}^{j} M_j$ *for any* $j \geq 1$, *none of the* K_i *is ever invoked and* M_j*'s redex does not contain a* K_i *in function position, then* $N_1 \mapsto_{ok}^{j} N_j$ *and* $M_k \approx N_k$ *for all* k, $1 \leq k \leq j$.

(ii) If $M_1 \mapsto_{ok}^{j} M_2 \equiv C_M[K_i L_M]$ *for a first such* $j \geq 1$, *an sk-context* $C_M[\ ]$ *and a value* L_M, *then* $N_1 \mapsto_{ok}^{\{j-1,j\}} N_2 \equiv C_N[K_i L_N]$ *and* $L_M \approx L_N$.

We are now ready to prove the main lemma for Theorem 7:

Lemma 11. *Let* K *be a continuation and let* N *be a value which does not contain a continuation. If there exists an sk-context* $C'[\ ]$ *and a value* L' *such that* $C'[KL'] \mapsto_{ok}^{+} N$, *then* $C[KL] \mapsto_{ok}^{+} N$ *for all sk-contexts* $C[\ ]$ *and for all values* $L \approx L'$.

Proof. Assume the hypothesis. We proceed by an induction on the unique number n such that $C'[KL'] \mapsto_{ok}^{n} N$ or, equivalently, on the structure of K:

(K1) $K \equiv \lambda x.\mathcal{A}x$. This case is trivial:

$$C'[KL'] \mapsto_{ok} C'[\mathcal{A}L'] \mapsto_{ok}^{+} L'$$

and for any $C[\ ]$, $L \approx L'$

$$C[KL] \mapsto_{ok} C[\mathcal{A}L] \mapsto_{ok}^{+} L.$$

Since both L and L' are values, $L \approx L' \equiv N$, and N does not contain continuations, we have: $L \equiv L'$.

(K2) $K \equiv \lambda v.K^*(Pv)$ for some continuation K^* and function P. The first step of the reduction is then:

$$C'[KL'] \mapsto_{ok} C'[K^*(PL')];$$

on the other hand, for $C[\ ]$ and L the development is:

$$C[KL] \mapsto_{sk} C[K^*(PL)].$$

The sk-redexes for the two terms are PL' and PL, respectively, and they are behaviorally equivalent: $PL \approx PL' (mod\, K_1, \ldots, K_n)$. Thus, according to Corollary 10, there are two sc-evaluations possible: the evaluation of PL' either invokes a continuation or it does not:

a) Suppose $PL' \mapsto^*_{sk} C'_{L'}[K_iQ']$ for some continuation K_i, some value Q' and some sk-context $C'_{L'}[\ ]$. K_i may be P or it may be some of the continuations in L'. In either case, $PL \mapsto^*_{sk} C_L[K_iQ]$ for some sk-context $C_L[\ ]$ and value Q such that $Q \approx Q'$. But we know that $C'_{L'}[K_iQ'] \mapsto^+_{sk} N$ and that this reduction sequence is at least one step shorter than the one for $C'[KL']$. Thus we can invoke the inductive hypothesis and this finishes the subcase.

b) Now, assume that no continuation gets invoked in the course of the sc-evaluation. Then we know by the assumption, that the sc-evaluation for PL' must be finite. Suppose $PL' \mapsto^+_{sc} Q'$ for some Q' which cannot be sc-reduced any further. By Corollary 10 we know that $PL \mapsto^+_{sc} Q$ for some Q such that $Q \approx Q'$. Since the sc-reductions do not involve top-level computation rules, there are three cases possible:

   b1) Q' and Q are values. The two reduction sequences are:

$$C'[K^*(PL')] \mapsto^+_{sk} C'[K^*Q'] \mapsto^+_{sk} N$$

   and

$$C[K^*(PL)] \mapsto^+_{sk} C[K^*Q],$$

   and by inductive hypothesis the conclusion follows.

   b2) $Q' \equiv \lambda R'$ and $Q \equiv \lambda R$ for $R' \approx R$. From Theorem 3 it follows that

$$C'[K^*(PL')] \mapsto^+_{sk} C'[K^*(\lambda R')] \mapsto^+_{sk} R'$$

and

$$C[K^{\bullet}(PL)] \mapsto^{+}_{ek} C[K^{\bullet}(\mathcal{A}R)] \mapsto^{+}_{ek} R.$$

R' may now reduce to N without invoking a continuation. In this case, by Corollary 10, R must reduce to N, since N does not contain continuations. Otherwise, also by Corollary 10, R' reduces to a first continuation invocation and R must invoke the same continuation on an equivalent value:

$$R' \mapsto^{\bullet}_{ek} C'_{R'}[K_i S'] \mapsto^{+}_{ek} N$$

and

$$R \mapsto^{\bullet}_{ek} C_R[K_i S] \text{ and } S \approx S'.$$

But then by inductive hypothesis we have $C_R[K_i S] \mapsto^{+}_{ek} N$.

b3) $Q' \equiv \mathcal{C}R'$ and $Q \equiv \mathcal{C}R$ for $R \approx R'$. By Theorem 4 the two reduction sequences must look like:

$$C'[K^{\bullet}(PL')] \mapsto^{+}_{ek} C'[K^{\bullet}(\mathcal{C}R')] \mapsto^{+}_{ek} R'[\![C'[K^{\bullet}(\mathcal{C}R')], \lambda x.\mathcal{A}x]\!]_c$$

and

$$C[K^{\bullet}(PL)] \mapsto^{+}_{ek} C[K^{\bullet}(\mathcal{C}R)] \mapsto^{+}_{ek} R[\![C[K^{\bullet}(\mathcal{C}R)], \lambda x.\mathcal{A}x]\!]_c.$$

By Lemma 6 the two continuations are behaviorally equivalent modulo $K^{\bullet}$:

$$[\![C'[K^{\bullet}(\mathcal{C}R')], \lambda x.\mathcal{A}x]\!]_c = \lambda v.[\![C'[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c(K^{\bullet}v) \stackrel{df}{=} S'$$

and

$$[\![C[K^{\bullet}(\mathcal{C}R)], \lambda x.\mathcal{A}x]\!]_c = \lambda v.[\![C[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c(K^{\bullet}v) \stackrel{df}{=} S.$$

Thus, $RS \approx R'S'$ and the rest of this subcase is just like the rest of the previous one.

Notice that S and S' have $K^\bullet$ as a common continuation which is not necessarily in the modulo set of L and L'.

(K3) $K \equiv \lambda f.K^\bullet(fP)$ for some continuation $K^\bullet$ and an arbitrary term P. This case can quickly be reduced to (K2). Again we have:

$$C'[KL'] \mapsto_{ok} C'[K^\bullet(L'P)]$$

and

$$C[KL] \mapsto_{ok} C[K^\bullet(LP)].$$

The rest is a case analysis of the possible results of the sc-evaluation of the arbitrary term P:

a) $P \mapsto_{oc}^{+} Q$ for some value Q. Then the above evaluations continue as follows:

$$C'[K^\bullet(L'P)] \mapsto_{ok}^{+} C'[K^\bullet(L'Q)]$$

and

$$C[K^\bullet(LP)] \mapsto_{ok}^{+} C[K^\bullet(LQ)].$$

Clearly $L'Q \approx LQ$ and thus we have reduced the subcase to (K2).

b) $P \mapsto_{oc}^{+} \mathcal{A}Q$ for some Q. Trivially,

$$C'[K^\bullet(L'P)] \mapsto_{ok}^{+} C'[K^\bullet(L'(\mathcal{A}Q))] \mapsto_{ok}^{+} Q$$

and

$$C[K^\bullet(LP)] \mapsto_{ok}^{+} C[K^\bullet(L(\mathcal{A}Q))] \mapsto_{ok}^{+} Q.$$

c) $P \mapsto_{ok}^{+} \mathcal{C}Q$ for some Q. This case is a bit tricky:

$$C'[K^\bullet(L'P)] \mapsto_{ok}^{+} C'[K^\bullet(L'(\mathcal{C}Q))] \mapsto_{ok} C'[K^\bullet(\mathcal{C}\lambda\kappa.Q(\lambda v.\kappa(L'v)))]$$

and

$$C[K^\bullet(LP)] \mapsto_{ok}^{+} C[K^\bullet(L(\mathcal{C}Q))] \mapsto_{ok} C[K^\bullet(\mathcal{C}\lambda\kappa.Q(\lambda v.\kappa(Lv)))].$$

But now we have $\lambda\kappa.Q(\lambda v.\kappa(L'v)) \approx_t \lambda\kappa.Q(\lambda v.\kappa(L'v))$ since by assumption $L' \approx L$ and, thus, we have the same situation as in subcase b3) of (K2). □

The preceding lemma immediately implies:

Proof of Theorem 7. Both directions of Theorem 7 are special cases of Lemma 11. For the direction from left to right, take $C'[\ ] = [\ ]$, for the opposite direction $C'[\ ] = C_M[\ ]$. □

As an aside we note that Lemma 9 and Lemma 11 also show:

Corollary 12. *Let M_1 and M_2 be terms, L_1 and L_2 values.*
If $M_1 \approx M_2$, $M_i \mapsto_{ok}^+ L_i$ for $i = 1, 2$, then $L_2 \approx M_2$.

Remark. The proofs of Lemma 9 and Lemma 11 rely on the fact that there are only two operations on continuations: $\mathcal{C}$ captures continuations and an application with a continuation in its function position invokes it. In other words, there is only one way to find out something about a continuation, namely, one can invoke it and test the result and the beahvior. This assumption would be invalidated by the presence of Church's δ-rule, which allows a program to compare closed normal forms. **End of remark.**

Let us now summarize what we have found out so far about the labeling and invoking of continuation functions. Theorem 4 tells us that a continuation abstracts the context of the $\mathcal{C}$-application. According to Theorem 7, the invocation of a continuation generally forgets about the current context and runs an abstraction of a former program context. The next theorem shows that we can get around the intermediate representation in terms of functional abstractions and work directly with contexts:

Theorem 13. *Let $K = [\![C[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c$ for some sk-context $C[\ ]$ and some term X, let L be a value, and let N be a value which does not contain a continuation as a subterm:*

$$C[L] \mapsto_{ok}^* N \text{ if and only if } KL \mapsto_{ok}^+ N.$$

Proof. The equivalence is shown by an induction on the structure of the continuation K:

(K1) $K \equiv \lambda x.\mathcal{A}x$. This implies that $C[\ ] = [\ ]$ and *vice versa* and the statement is obviously true.

(K2) $K \equiv \lambda v.K^*(Pv)$ for a continuation K^* and some function P. From Lemma 6 we know that $C[\ ] = C^*[P[\ ]]$ for some context $C^*[\ ]$ and $K^* = [\![C^*[\mathcal{C}X], \lambda x.\mathcal{A}x]\!]_c$ for any term X. Then the two evaluations must begin with $KL \mapsto_{sc} K^*(PL)$ and $C[L] = C^*[PL]$, respectively, such that PL is the β_v-sk-redex in both terms. There are three possible cases for the $\mapsto_{sc}$-evaluation of PL:

a) $PL \mapsto_{sc}^+ L^*$ for some value L^*. But then we can apply the inductive hypothesis:

$$KL \mapsto_{sc}^+ K^*L^* \mapsto_{sk}^+ N$$

and

$$C[L] \mapsto_{sc}^+ C^*[L^*] \mapsto_{sk}^+ N.$$

b) $PL \mapsto_{sc}^+ (\mathcal{A}L^*)$ for some L^*. By Theorem 3, the two evaluations continue with the same term:

$$KL \mapsto_{sc}^+ K^*(\mathcal{A}L^*) \mapsto_{sk} L^*$$

and

$$C[L] \mapsto_{sc}^+ C^*[(\mathcal{A}L^*)] \mapsto_{sk}^+ L^*.$$

c) $PL \mapsto_{sc}^+ (\mathcal{C}L^*)$ for some L^*. By Theorem 4 and the inductive hypothesis we get:

$$KL \mapsto_{sk}^+ K^*(\mathcal{C}L^*) \mapsto_{sk}^+ L^*(\lambda v.(\lambda x.\mathcal{A}x)(K^*v))$$

and

$$C[L] \mapsto_{sk}^+ C^*[\mathcal{C}L^*] \mapsto_{sk}^+ L^*K^*.$$

Since the two intermediate results are behaviorally equivalent terms, we obtain the conclusion using Corollary 12.

(K3) $K \equiv \lambda f.K^*(fQ)$ for a continuation K^* and some term Q. This case is the same as (K2). □

Putting Theorem 7 and Theorem 13 together yields a simple, but effective rule for the invocation of a continuation: throw away the entire term and replace it by the context which the continuation stands for, filling the hole with the argument. This rule works as long as the computation is in an infinite loop or the final value does not contain a continuation. If it does, we can generalize the above proofs to show that the result according to the rule is behaviorally equivalent to the *real* result obtained from a (pure) sk-evaluation. However, it is not quite clear what this really means. In fact, it is not even clear what it means to get a continuation as (part of) a result. The answer to these questions lies in the nature of the interaction between the evaluating machine and the programmer.

For a *batch* computation one will indeed want to interpret the final result as a number, truth value, *etc.* One is definitely not interested in getting back a continuation as (part of) the result, since continuations abstract machine behavior. Depending on the actual evaluation mechanism, *i.e.* machine, it is even unlikely that the answer would say any more than "THIS IS A CONTINUATION." In this case the above theorem and lemmas suffice. If however, the computation proceeds *interactively*, it makes sense to save an intermediate result which abstracts control for later use. But the stress is on future use: these intermediate results are only useful for a potential application in the future and then the fact that they are behaviorally equivalent assures us that there will be no way to find out the difference. Hence, we can use the above theorems and the rule without hesitation when symbolically evaluating programs.

5. Programming with continuations

The λ_c-calculus *per se* gave us the possibility to symbolically evaluate programs which use continuation-based control facilities. The theorems of the preceding section reduce the complexity of the rewrite rule system to the level of the β_v-rule:

The Continuations-as-Contexts Rule

When grabbing a continuation, remove and remember the current sk-context; when invoking a continuation, replace the current program by the respective sk-context, filling the hole with the argument.

With this rule in mind, programming and reasoning with continuations becomes an acceptable alternative to working within the purely functional λ-calculus. The examples in the sequel of this section illustrate simple applications of this context rule. They may appear rather trivial at a first glance, but without the above rule—which we use as if it were part of the definition of the sk-function—they would be less tractable. We urge the skeptical reader to translate the programs via cps into the λ-calculus and carry out the corresponding validations there.

The first example is the program $\omega(\mathcal{C}\omega)$ where $\omega \equiv \lambda x.xx$. It is interesting in its own right since continuations are passed out of their original scope as the evaluation shows:

$$\begin{aligned}\omega(\mathcal{C}\omega) &\mapsto^{+}_{sk} kk \text{ where } k \sim \omega[\ ] \\ &\mapsto^{+}_{sk} \omega k \\ &\mapsto^{+}_{sk} kk \\ &\mapsto^{+}_{sk} \omega k \ldots\end{aligned}$$

The notation $k \sim \omega[\ ]$ stands for "k represents the context $\omega[\ ]$." The program obviously loops forever. That means, the program is also in an infinite loop when evaluated with the pure sk-function, but it may be looping in a different context.

As our programming language is rather primitive we introduce some common combinators and syntactic forms[5]. The functions $\mathbf{T}_v \equiv \lambda xy.x\mathbf{I}$ and $\mathbf{F}_v \equiv \lambda xy.y\mathbf{I}$ stand for the truth values *true* and *false*, respectively. Given truth values, we can implement a call-by-value version of the form (**if** $N_1\, N_2\, N_3$) with the obvious behavior. Assuming that N_1 reduces to a

[5] Compare the corresponding call-by-name forms in [1], p.129.

boolean value, the form stands for $N_1(\lambda d.N_2)(\lambda d.N_3)$ where d is a dummy variable.

The function $[\_,\_] \equiv \lambda mn.\lambda x.(\mathbf{if}\ x\, m\, n)$ stands for the pairing function, $(\_)_0 \equiv \lambda p.p\mathbf{T}_v$ and $(\_)_1 \equiv \lambda p.p\mathbf{F}_v$ for the left and right selectors: $([M_0, M_1])_i \mapsto_{sc} M_i$. The abbreviation $(\mathbf{let}\ ([x,y]N)M)$ is to be read as $((\lambda p.((\lambda xy.M)(p)_0(p)_1))N)$.

For our next example we want to extend the above loop so that it iterates a function f over a value x. We assume that successive applications of f to a value X always sc-reduce to some value:

$$fX \mapsto^+_{sc} X_f$$
$$fX_f \mapsto^+_{sc} X_{ff}$$
$$fX_{ff} \mapsto^+_{sc} X_{fff}\ \textit{etc.}$$

Then the function $\mathbf{L}^\infty$ defines a loop for f which iteratively generates the values of fx, $f(fx)$, *etc.*

$$\mathbf{L}^\infty \equiv \lambda fx.(\mathbf{let}\ ([\kappa, x](\mathcal{C}\lambda\kappa.\kappa[\kappa, x]))(\kappa[\kappa, fx])).$$

This claim can easily be checked by a symbolic evaluation:

$$\begin{aligned}\mathbf{L}^\infty fx &\mapsto^+_{sk} (\mathbf{let}\ ([\kappa, x](\mathcal{C}\lambda\kappa.\kappa[\kappa, x]))(\kappa[\kappa, fx])) \\ &\mapsto^+_{sk} \kappa[\kappa, X_f] \text{ where } \kappa \sim (\mathbf{let}\ ([\kappa, x][\ ])(\kappa[\kappa, fx])) \\ &\mapsto^+_{sk} (\mathbf{let}\ ([\kappa, x][\kappa, X_f])(\kappa[\kappa, fx])) \\ &\mapsto^+_{sk} (\mathbf{let}\ ([\kappa, x][\kappa, X_{ff}])(\kappa[\kappa, fx])) \ldots\end{aligned}$$

We have generated this loop without a full-blown recursion combinator. With continuations it is also possible to construct recursive functions without using a classical recursion combinator.

Now, suppose that f is as above and that p is a predicate returning a truth value for every $X_f, X_{ff}, \ldots$ Then we claim that

$$\mathbf{L}^* \equiv \lambda pfx.\mathcal{C}(\lambda\kappa.\mathbf{L}^\infty(\lambda y.(\mathbf{if}\ (py)\ (\kappa y)\ (fy)))(fx))$$

implements an iteration combinator which generates the values $X_f, X_{ff}, \ldots$ and returns the first one for which p evaluates to true. Again, the proposition can be validated by a symbolic reduction.

For our next example we adopt Barendregt's numeral system for the λ_c-calculus: $\ulcorner 0 \urcorner \equiv \mathbf{I}$, $\ulcorner n+1 \urcorner \equiv [\mathbf{F}_v, \ulcorner n \urcorner]$, $\mathbf{zero?} \equiv \lambda x.x\mathbf{T}_v$, and $\mathbf{S}^+ \equiv \lambda x.[\mathbf{F}_v, x]$. We also use $+$ for the addition function. Using the pairing operator one can now talk about (finite) sequences of natural numbers. We let $\mathrm{nil} \equiv [\mathbf{T}_v, \mathbf{F}_v]$ stand for an *end-of-sequence* marker. The function $\mathbf{null?} \equiv \lambda x.(\mathbf{if}\ (\mathbf{zero?}\ x)\ \mathbf{F}_v\ (x)_0)$ is then an adequate test on the nil sequence. The expression $[1,2,3]$ is shorthand for $[1,[2,[3,\mathrm{nil}]]]$.

The function Σ which takes a list of numbers and returns their sum is obviously a trivial exercise in recursive function writing:

$$\Sigma \equiv \mathbf{Y}_v \lambda s.(\lambda l.(\mathbf{if}\ (\mathbf{null?}\ l)\ \ulcorner 0 \urcorner\ (+(l)_0(s(l)_1)))).$$

$\mathbf{Y}_v \equiv \lambda f.(\lambda x.f(\lambda z.xxz))(\lambda x.f(\lambda z.xxz))$ represents the call-by-value recursion combinator [8]. It is less trivial to change this function—without using $\mathcal{C}$—so that it returns $\ulcorner 0 \urcorner$ if any of the elements is $\ulcorner 0 \urcorner$ although this is only a minor modification to the original specification. With the $\mathcal{C}$-operator the new function Σ_0 is a straightforward extension of the Σ-function:

$$\begin{aligned}\Sigma_0 \equiv\ & \lambda l. \\ & \mathcal{C}\lambda\kappa.\kappa \\ & (\mathbf{Y}_v\lambda s.(\lambda l.(\mathbf{if}\ (\mathbf{null?}\ l)\ \ulcorner 0 \urcorner\ (\mathbf{if}\ (\mathbf{zero?}\ (l)_0)\ (\kappa \ulcorner 0 \urcorner)\ (+(l)_0(s(l)_1)))))))\ l).\end{aligned}$$

We illustrate the working of Σ_0 by a symbolic evaluation of the program

$\mathbf{S}^+(\Sigma_0[\ulcorner 1\urcorner, \ulcorner 2\urcorner, \ulcorner 0\urcorner, \ulcorner 3\urcorner])$:

$$\begin{aligned}
\mathbf{S}^+(\Sigma_0[\ulcorner 1\urcorner, \ulcorner 2\urcorner, \ulcorner 0\urcorner, \ulcorner 3\urcorner]) &\mapsto^+_{sk} \kappa(s_{fp}[\ulcorner 1\urcorner, \ulcorner 2\urcorner, \ulcorner 0\urcorner, \ulcorner 3\urcorner]) \\
&\quad \text{where } s_{fp} \text{ is defined by} \\
&\quad\quad \mathbf{Y}_v(\lambda s.\lambda l.\ \ldots) \mapsto_{sk} s_{fp} \\
&\quad \text{and } \kappa \sim \mathbf{S}^+[\ ] \\
&\mapsto^+_{sk} \mathbf{S}^+(+\ulcorner 1\urcorner(s_{fp}[\ulcorner 2\urcorner, \ulcorner 0\urcorner, \ulcorner 3\urcorner])) \\
&\mapsto^+_{sk} \mathbf{S}^+(+\ulcorner 1\urcorner(+\ulcorner 2\urcorner(s_{fp}[\ulcorner 0\urcorner, \ulcorner 3\urcorner]))) \\
&\mapsto^+_{sk} \mathbf{S}^+(+\ulcorner 1\urcorner(+\ulcorner 2\urcorner(\kappa\ulcorner 0\urcorner))) \\
&\mapsto^+_{sk} \mathbf{S}^+\ulcorner 0\urcorner \mapsto^+_{sk} \ulcorner 1\urcorner.
\end{aligned}$$

With our last example we get even closer to real-world programming problems. We extend the Σ and Σ_0 functions to work on *trees* of numbers. In other words, we design a function $\Sigma^\bullet$ that sums up the numbers in all the nodes of a tree and another function $\Sigma^\bullet_0$ which only sums up the elements of the tree if there is no $\ulcorner 0\urcorner$ in the tree, but returns $\ulcorner 0\urcorner$ otherwise.

For this exercise we assume that trees are either empty trees or non-empty trees consisting of a *left-son* tree, a number, and a *right-son* tree. We represent empty trees by $\langle\rangle$ and non-empty ones by $\langle M, \ulcorner n\urcorner, N\rangle$ where M and N are the left- and right-son and n is the node information. The functions **mt?**, **lson**, **rson**, and **num** are the respective predicates and selector functions. It is obvious that this tree representation is λ-definable by a straightforward extension of the above list representation, but we omit the details.

The function $\Sigma^\bullet$ resembles Σ:

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

Σ_0^* in turn is related to Σ_0:

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

For a comparisson we give an intensionally equivalent definition[6] of this function in the pure λ-calculus:

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

It is an interesting exercise to trace and compare the evaluation of an application of Σ_0^* to some tree like $\langle\langle\langle\rangle, \ulcorner 0\urcorner, \langle\rangle\rangle, \ulcorner 1\urcorner, \langle\langle\rangle, \ulcorner 2\urcorner, \langle\rangle\rangle\rangle$ for both versions. We recommend it to the still unconvinced readers.

6. Conclusions

In the preceding sections we have shown how the λ-calculus can be extended to a control calculus. The resulting system is sound and consistent. A standardization theorem determines the operational semantics for λ_c and allows us to talk about evaluations, in particular, about the behavior of continuations during evaluations.

---

[6] By this we mean that the function is a single-pass function which only performs additions if necessary and escapes as soon as a $\ulcorner 0\urcorner$ is discovered. There are alternatives to the $\mathcal{C}$-free version but they can all be derived from the cps definition of Σ_0^*.

The essence of Section 4 is a rule which expresses all continuation operations in terms of contexts. It makes it possible to reason about non-functional control in the same manner that we reason about functional programming. Since first-class continuations can imitate any sequential control strategy, one can modify the calculus, theorems, and rules to deal with other constructs.

The control calculus also raises the question of what kind of objects continuations really are. In denotational semantics they are represented by functions. But this only works because the definitions are expressed—should we say programmed?—in a particular style, namely cps. Their true nature remains concealed. We expect that a further investigation of the λ_c-calculus will deepen our understanding of continuation objects and the nature of control operations in programminng languages.

Acknowledgement. We wish to thank Mitchell Wand for his helpful discussions and comments on earlier drafts of this paper. Michael Dunn helped to clarify the Soundness and Incompleteness Theorem. We are also grateful to John Gateley, Chris Haynes, and Carolyn Talcott for their comments.

This material is partly based on work supported by the National Science Foundation under grants DCR 85-01277 and DCR 85-03279. Eugene Kohlbecker is an IBM Graduate Fellow.

7. References

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

[2] Clinger, W.D., et. al., The revised revised report on Scheme, *Joint Technical Report Indiana University 174 and MIT Laboratory for Computer Science 848*, 1985.

[3] Friedman, D.P., C.T. Haynes, E. Kohlbecker, Programming with continuations, in P. Pepper (ed.), *Program Transformations and Programming Environments*, Springer Verlag, 1985.

[4] Haynes, C. T., Logic continuations, Technical Report No. 183, Indiana University Computer Science Department, to appear in the proceedings

of the *Third International Conference on Logic Programming,* London (July, 1986), Springer-Verlag, 1986.

[5] Landin, P.J., The mechanical evaluation of expressions, *Computer Journal* **6** (4), 1964.

[6] Mellish, C., S. Hardy, Integrating Prolog in the POPLOG environment, in J.A. Campbell (ed.), *Implementations of Prolog*, Ellis Horwood, 1984.

[7] Plotkin, G., Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science* **1**, pp. 125–159, 1975.

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

[9] Reynolds, J.C., Definitional interpreters for higher-order programming languages, *Proc. ACM National Convention*, 717–740, 1972.

[10] Steele, G., *COMMON LISP - The Language*, Digital Press, 1984.

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

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