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 axioms: $\mathcal{C}$, 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 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 suffices from a purist's point of view, it is in many cases 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 [4], label values [7], *call-with-current-continuation* (abbreviated as *call/cc*) [2], and *catch* and *throw* [9].

Non-functional control operators "provide a way of pruning unnecessary computation and allow certain computations to be expressed by more compact and conceptually manageable programs." [3] If they make continuations

[1] [1], p.3

[2] C. Talcott about the introduction of *note* into Rum, a lexically-scoped dialect of Lisp [10], p.68.

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, and 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.

available as first-class objects, as in Scheme or ISWIM, it is easy to imitate any desired sequential control construct with little effort, *e.g.* escapes, error stops, search strategies as applied in logic programming [5], intelligent backtracking [3], and coroutining [10]. 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 [10]. 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 [6] 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 3 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 [10], 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 catching—of continuations with reference to label values in more traditional languages. The continuation is represented by a function; we generally 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:

$$[\![x]\!] = \lambda\kappa.\kappa x, \qquad (cps1)$$

$$[\![(\lambda x.M)]\!] = \lambda\kappa.\kappa(\lambda x.[\![M]\!]), \qquad (cps2)$$

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

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

$$[\![(\mathcal{A}M)]\!] = \lambda\kappa.[\![M]\!]\mathbf{I}. \qquad (cps5)$$

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. It is easy to see from these equations that neither $\mathcal{C}$ nor $\mathcal{A}$ are the images of combinators in Λ.

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 [6]. 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. The expression $\lambda f.(fN)$ 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:

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

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

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:

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

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

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:

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

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 $\mathbf{KI}$.

Definition 2: The λ_c-calculus

Let $\xrightarrow{c} = \xrightarrow{\mathcal{C}_1} \cup \xrightarrow{\mathcal{C}_2} \cup \xrightarrow{\mathcal{A}_1} \cup \xrightarrow{\mathcal{A}_2} \cup \xrightarrow{\beta_v}$. 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.

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 sub-calculus of λ_c. By throwing in the additional computation rules we obtain the complete control calculus. The formal definition is shown in Definition 2. When we refer to equations in the traditional λ-calculus, we use $=_\beta$ instead of $=_c$ and $=_k$.

[4] [1], p.50

We 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. This immediately 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 [6].

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 $\xrightarrow{\mathcal{C}_1}$, $\xrightarrow{\mathcal{C}_2}$, $\xrightarrow{\mathcal{A}_1}$, or $\xrightarrow{\mathcal{A}_2}$, and let $\triangleright$ stand for either $\triangleright_{\mathcal{C}}$ or $\triangleright_{\mathcal{A}}$:*

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

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 $\xrightarrow{\beta}$:

Theorem 2. *The relation $\xrightarrow{c}$ is Church-Rosser.*

Based on this, we can easily show:

Definition 3: 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.

Theorem 3 (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* $L_i \triangleright_k N$ *for* $i = 1, 2$.

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.

Furthermore, we can now prove:

Theorem 1' (Incompleteness). *Let* $L \in \Lambda$ *be an abstraction. Then there are* M *and* N *in* Λ_c *such that* $[M]L =_\beta [N]L$ *but* $M \neq_k N$.

The proposition is a consequence of Theorem 3 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. We use Plotkin's elegant method and first define 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. Then we extend it to standard reduction sequences by forming something like a compatible closure: see Definition 3. The standardization theorem follows from this definition by adapting Plotkin's corresponding proof:

Theorem 4 (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* $L_n \equiv N$

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.

3. 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], [10]. The λ_c-calculus allows us to understand the labeling and invocation of continuations in terms of program code. The major tool for this analysis is the operational semantics of the λ_c-calculus.

A closer look at the reduction and computation rules for $\mathcal{C}$- and $\mathcal{A}$-applications makes it clear that $\mathcal{C}$ is eliminated in favor of $\mathcal{A}$ and that $\mathcal{A}$ removes a term by replacing the term with the $\mathcal{A}$-argument. The removal of a $\mathcal{C}$-application ($\mathcal{C}L$) from a term M involves an extensive rewriting of the whole term and always leads to the application of L to some λ-abstraction of the rest of M, *i.e.* the textual context of ($\mathcal{C}L$). When the continuation is eventually invoked, the rest of the reduction process is determined by the former context of ($\mathcal{C}L$). Before we proceed to prove some interesting properties about these processes, we need some terminology. The *sk-redex* of a term M is *the* redex such that $M \mapsto_{sk} N$ for some N. The *depth* of an sk-redex is the distance from the root of M to the redex. Both notions are formally defined in Definition 4. We sometimes prefix sk-redex with $\mathcal{C}$, $\mathcal{A}$, or β for clarification.

The concept of a textual context is formalized in the notion of a Λ_c-*context* which is a Λ_c-term with zero or more holes in it. The definition is just like the one for Λ_c except that the base case also includes [ ]. We denote contexts with $C[\ ], C'[\ ]$, *etc.* $C[M]$ is the term where all holes of the context $C[\ ]$ are filled with M. Free variables of M may become bound through the filling in. An *sk-context* is a one-hole context such that only the hole could possibly contain or be part of an sk-redex, *i.e.*, [ ] is an sk-context, if P is any term and $C[\ ]$ is an sk-context then $C[\ ]P$ is an sk-context, and, if Q is a value and $C[\ ]$ is an sk-context, then $QC[\ ]$ is an sk-context. For reasons of hygiene we state:

Proposition. *An sk-context cannot bind free variables.*

Proof. An sk-redex cannot be within the scope of an abstraction by definition of $\mapsto_{sk}$. □

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. Define $M \equiv C[P]$ and $N \equiv C[Q]$. The argument for (iii) is an induction on the structure of M; points (i) and (ii) fall out automatically.

SC1: $M \equiv P$. But then $N \equiv Q \equiv (\otimes R)$ and N is a $\otimes$-sk-redex with $d^{sk}_N = 0$.

SC2: $M \equiv M_1 M_2, M_1$ is an application and contains M's sc-redex. By inductive hypothesis $M_1 \mapsto_{sc} N_1$ 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 we know from Lemma 1 that 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.

Definition 4: Sk-redexes, sc-redexes, and the depth of a redex.

The *sk-redex of a term M* and its *depth* d^{sk}_M is defined as:
(SK1)M if M is a $\mathcal{C}_T$- or an $\mathcal{A}_T$-redex and $d^{sk}_M = 0$;
(SK2)P if P is the sc-redex of M and $d^{sk}_M = d^{sc}_M$.

The *sc-redex of a term M* and its *depth* d^{sc}_M is defined as:
(SC1)M if M is a C-redex and $d^{sc}_M = 1$;
(SC2)P if $M \equiv KL, K$ is an application and P is the sc-redex of K and $d^{sc}_M = d^{sc}_K + 1$;
(SC3)P if $M \equiv KL, K$ is a value and P is the sc-redex of L and $d^{sc}_M = d^{sc}_L + 1$.

SC3: $M \equiv M_1 M_2, M_1$ is a value and M_2 contains M's sc-redex. This case is just like (SC2). □

Equipped with this machinery we can prove that $\mathcal{A}$ behaves like an *abort* operation:

Theorem 3. *Suppose the sk-redex of a term M 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. □

Next we turn our attention to the labeling of continuations. The theorem we have in mind structurally resembles the previous one. It shows how a $\mathcal{C}$-sk-redex is removed from a term and how at the same time the current continuation is computed from the context of the $\mathcal{C}$-application.

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 an $\mathcal{A}$-application, it 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 function which takes a continuation κ as an argument and then applies P to $\lambda f.\kappa(fQ)$, *i.e.*, it extends the continuation κ to the current continuation; otherwise it is part of a $\mathcal{C}_R$-redex, say $P(\mathcal{C}Q)$, and then invokes Q on the continuation fragment $\lambda v.\kappa(Pv)$. 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 has one of the above mentioned two forms: $\lambda\kappa.P(\lambda f.\kappa(fQ))$ or $\lambda\kappa.Q(\lambda v.\kappa(Pv))$. In either case it 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 function which requested it.

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

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

Intuitively 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 the current continuation requested by the $\mathcal{C}$-sk-redex:

Theorem 4. *Let the sk-redex of a term M be a $\mathcal{C}$-redex whose $\mathcal{C}$-application is $\mathcal{C}L$. Then M evaluates to an application of the $\mathcal{C}$-argument to the current continuation:*

$$M \mapsto_{sk}^{+} L[\![M, \lambda x.\mathcal{A}x]\!]_c.$$

Proof. We prove our claim by induction on the depth of the sk-redex in M:

$d_M^{sk} = 0$: $M \equiv \mathcal{C}L \mapsto_{sk}^{+} L(\lambda x.\mathcal{A}x)$ and $[\![M, \lambda x.\mathcal{A}x]\!]_c = \lambda x.\mathcal{A}x$.

$d_M^{sk} > 0$: 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, and that $d_N^{sk} < d_M^{sk}$. Hence, we can apply the inductive hypothesis to N and get:

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

From this theorem we can immediately deduce a corollary about the continuation functions obtained by a $\mathcal{C}$-redex:

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.*

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 the sk-context $C[\ ]$:

SC1: $C[\ ] = [\ ]$. Then $[\![C[P], \kappa]\!]_c = [\![P, \kappa]\!]_c = [\![P, [\![\mathcal{C}X, \kappa]\!]_c]\!]_c$.

SC2: $C[\ ] = C'[\ ]Q$ for some sk-context $C'[\ ]$. Now we have:

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

SC3: Similar to (SC2). □

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. According to Theorem 3, if the continuation ever reaches the bottom in the course of an evaluation, it can forget about the context of its invocation. The question is, what happens if the continuation does not reach its initial $\mathcal{A}$-application.

There are two possible cases in which a continuation function does not complete the functional control path that leads to the $\mathcal{A}$-application. It may invoke an $\mathcal{A}$-application at one of the intermediate stages, or it may grab a continuation with a $\mathcal{C}$-application. In the former case we have no problem at all: any $\mathcal{A}$-application which becomes the sk-redex will remove the 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 evaluation of a continuation invocation can neglect 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. But, how different are the newly generated continuation functions?

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 $\mathbf{I}$ in the sk-context $C[\ ]$. The evaluation proceeds as follows:

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

Without context the result would look different:

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

However, one can see that the two results would behave similarly if they were invoked. Both would immediately call the continuation K' and, if this continuation reaches its bottom, the two would yield the same result.

By induction from this specific case, we can argue that the results are the same except for some continuations, but those must *behave equivalently.* The reason is that according to Theorem 4 a $\mathcal{C}$-application will always encode the rest of the term as an abstraction. Hence, if during the evaluation of some continuation a new continuation is labeled, the remainder of the old one will be the beginning of the new one. Thus the rest of the context cannot play an active role; it can never change the course of an evaluation. If the final result does not contain continuations, the context of a continuation invocation does not make any difference at all:

Theorem 7. *Let the β_v-sk-redex of M be of the form KL where K and L stand for a continuation function and a value, respectively, and let N be a value which does not contain a continuation as a subterm:*

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

Proof Idea. Since the proof is rather long and tedious, we only indicate the major proof idea. The proof is conceptually an induction on the number of steps in $KL \mapsto^{+}_{sk} N$, but in order to make the induction work

smoothly we need to prove a stronger statement:

> If $C'[KL']$ evaluates to N for some sk-context $C'[\ ]$ and value L' then $C[KL]$ reduces to N for all sk-contexts $C[\ ]$ and all values L which are behaviorally equivalent to L'.

Behavioral equivalence essentially means that any further evaluation of the two terms either returns the same value or returns two functions which are behaviorally equivalent. Hence, there is no possibility within the λ-calculus to differentiate the two resulting functions since there is no equality predicate on arbitrary terms. The syntactic difference between them is that wherever one term contains a continuation the other must contain a behaviorally equivalent continuation. Two continuations are said to behave the same way if, when invoked, they reduce in less than two steps to the invocation of the same continuation. □

Let us summarize what we have found out so far about 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 8. *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 $KL \mapsto_{oc} K'(PL)$ 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 $\mapsto_{oc}$-evaluation of PL:

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

$$KL \mapsto_{oc}^{+} (K'L') \mapsto_{ok}^{+} N$$

and

$$C[L] \mapsto_{oc}^{+} C'[L'] \mapsto_{ok}^{+} N.$$

b) $PL \mapsto_{oc}^{+} (\mathcal{A}L')$ for some L'. This time by Theorem 3 we get

$$KL \mapsto_{oc}^{+} K'(\mathcal{A}L') \mapsto_{ok} L'$$

and

$$C[L] \mapsto_{oc}^{+} C'[(\mathcal{A}L')] \mapsto_{ok}^{+} L'.$$

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

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

and

$$C[L] \mapsto_{ok}^{+} C'[\mathcal{C}L'] \mapsto_{ok}^{+} L'K'.$$

Since the two intermediate results are behaviorally equivalent terms, we obtain the conclusion using the strong version of Theorem 7.

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

The preceding theorem is a complete and simple rule for thinking about continuations: 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 programming with continuations becomes much easier.

4. Programming with continuations

The examples in this section illustrate simple applications of our context rule for continuations. They may appear rather trivial at a first glance, but without the above rule 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 proofs 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:

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

The program obviously loops forever. The notation $k \sim \omega[\ ]$ stands for "k represents the context $\omega[\ ]$."

As our language is rather primitive we introduce some common combinators and syntactic forms. The function $[\_,\_]$ stands for the pairing function, $(\_)_0$ and $(\_)_1$ for the left and right selectors: $([M_0, M_1])_i \mapsto_{sc} M_i$. The abbreviation **(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.*

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

This claim can easily be checked by a symbolic evaluation:

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

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

Before we present our final example we introduce some more syntax and combinators. 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 boolean value, the form stands for $N_1(\lambda d.N_2)(\lambda d.N_3)$ where d is a dummy variable.

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

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

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 is validated by a symbolic reduction.

5. 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.

The essence of Section 3 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 about 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 comments on the final draft of the paper.

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.

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

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

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

[7] 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, 1971.

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

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

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