A Syntactic Theory of Sequential Control

by

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

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 215

A Syntactic Theory of Sequential Control

by

M. Felleisen, D.P. Friedman, E. Kohlbecker, & B. Duba

February, 1987

This material is based on work supported in part by IBM Fellowship Awards to Matthias Felleisen and Eugene Kohlbecker, and by National Science Foundation grants number DCR 85-01277 and DCR 85-03325.

A Syntactic Theory of Sequential Control

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

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405, USA

Abstract

Sequential control operators like J and call/cc are often found in implementations of the λ-calculus as a programming language. Their semantics is always defined by the evaluation function of an abstract machine. We show that, given such a machine semantics, one can derive an algebraic extension of the λ_v-calculus. The extended calculus satisfies the diamond property and contains a Church-Rosser subcalculus. This underscores that the interpretation of control operators is to a certain degree independent of a specific evaluation strategy. We also prove a standardization theorem and use it to investigate the correspondence between the machine and the calculus. Together, the calculus and the rewriting machine form a syntactic theory of control, which provides a natural basis for reasoning about programs with non-functional control operators.

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

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

The one and only means of computation in the calculus is the β-reduction rule which directly models function application. Although this suffices from a purist's point of view, it is in many cases insufficient with respect to expressiveness and inefficient with respect to the evaluation process. For example, when a recursive program discovers the final result in the middle of the computation process, it should be allowed to immediately *escape* and report its value. Similarly, in an

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

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 these operations make continuations

---

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

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

[3] C. Talcott wrote this remark in the context of escape mechanisms, but the spirit of her dissertation makes

available as first-class objects, as in Scheme or ISWIM, it is easy to imitate any desired sequential control construct, *e.g.* escapes, error stops, search strategies as applied in logic programming [7, 12], intelligent backtracking [5], and coroutining [21]. Even though this is widely recognized, control operators are still regarded with skepticism. Their addition seems rather *ad hoc*, because it only advances the calculus as a programming language. On the algebraic side there are no rules reflecting the new operations; proofs of program properties can no longer be carried out in the syntactic domain. They must be based upon a semantic interpretation in terms of abstract machines or denotational definitions [11, 21].

In this paper, an expanded revision of two preliminary reports [2, 3], we show that the λ-calculus as an equational system can incorporate control operators and that non-functional control may be characterized in a purely syntactic manner. Starting from an idealized Scheme with an operational semantics, we design an extended calculus-like system that captures the behavior of the additional control operators. The theory is consistent in the sense that two different derivations starting with the same term are confluent. Hence, it permits algebraic calculations in the familiar style. A standardization theorem shows that the calculus defines a programming language in its own right. Comparing the calculus and the programming language semantics, we find that the two truly correspond to each other. The standard computation function agrees with the evaluation function except for some negligible differences and equality in the calculus implies indistinguishability on the machine. Together with the machine semantics, the calculus provides a powerful framework for reasoning about programs with non-functional control operators. We exemplify and discuss this in more depth in the last section.

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

2. Λ_c

The programming language Scheme [15] is the starting point of our language design. Of all the λ-calculus-based languages, its continuation accessing operation, call/cc, is independent of the rest of the primitive language concepts. For example,

clear that it is also applicable to jump operations in general [21, p.16].

the J-operator [8] interferes with functional abstraction; the **escape**-construct [16] introduces a second binding facility. Thus, call/cc, which simply takes an argument and applies it to the current continuation, appears to be the ideal candidate. However, it too complicates the development of a calculus. When call/cc is used, it not only transfers control over the current continuation to the program, it also installs this continuation. This implies that a program has only partial control over the continuation.

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

2.1. Syntax

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

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

$$\begin{aligned} &x[x := N] \equiv N, \quad y[x := N] \equiv y \qquad (x \neq y), \\ &(\lambda y.M)[x := N] \equiv (\lambda y.M[x := N]), \\ &(LM)[x := N] \equiv (L[x := N]M[x := N]) \end{aligned}$$

and

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

2.2. Operational semantics

The intention behind the two operations $\mathcal{C}$ and $\mathcal{A}$ can be explained informally. An $\mathcal{A}$-application represents an *abort* or stop operation, which terminates the program

Definition 2.1: The term sets Λ_c and Λ

The improper symbols are λ, (,), ., $\mathcal{C}$, and $\mathcal{A}$. *Var* is a countable set of variables; the symbols x, k, f, v, *etc.* range over *Var* as meta-variables but are also used as if they were elements of *Var*. $L, M, N, \ldots$ are meta-variables for Λ_c. 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*; $U, V, \ldots$ are meta-variables for values.

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

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

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

The formal semantics of Λ_c is defined via an abstract machine. We started from an operational interpretation of a denotational semantics in the style of Reynolds's continuations-as-data structures interpreter [16]. Such a machine has three state

components: terms as control strings, environments for the evaluation of free variables, and continuation structures to remember the rest of a computation. The elimination of environments by quasi-substitution and the replacement of continuation codes by a special kind of context leads to an equivalent quasi-term rewriting system [2]. Since this rewriting system is more appropriate for the derivation of a calculus, we take it as the basis of our development.

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

Definition 2.2: The C-rewriting system

The term language Λ_p:

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

The C-transition function:

$$C[(\lambda x.M)V] \xmapsto{C} C[M[x := V]] \quad \text{(C1)}$$
$$C[\mathcal{C}M] \xmapsto{C} M\langle \mathbf{p}, C[\ ] \rangle \quad \text{(C2)}$$
$$C[\langle \mathbf{p}, C_0[\ ] \rangle V] \xmapsto{C} C_0[V] \quad \text{(C3)}$$
$$C[\mathcal{A}M] \xmapsto{C} M. \quad \text{(C4)}$$

The transition function of the C-rewriting system is displayed in Definition 2.2. It performs a single evaluation step on an entire program. The unique partitioning of a program into an applicative context and a C-redex determines the next rewriting

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

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

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

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

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

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

With an operational semantics à la SECD-machine [9] operational equivalence is rather opaque. Verifying the equivalence of two useful program pieces is almost impossible. With the C-rewriting system this becomes simpler. The transition function only uses one extraneous concept, namely applicative contexts, and this is naturally related to terms. However, the system only accounts for entire program