Beyond Continuations

by

Matthias Felleisen, Daniel P. Friedman,
Bruce Duba & John Merrill

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 216

Beyond Continuations

by

M. Felleisen, D.P. Friedman, B. Duba & J. Merrill

February, 1987

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

BEYOND CONTINUATIONS

—PRELIMINARY REPORT—

Matthias Felleisen, Daniel P. Friedman, Bruce Duba, John Merrill

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

Abstract

The simplification of the extended λ-calculus for program continuations leads to a new, powerful programming language control construct. Conventional continuation semantics turns out to be inadequate for a denotational definition. We introduce the concept of compositional continuation functions. Based on this solution, we conjecture that a denotational semantics can entirely rely on the domain of function values for the modelling of control constructs.

1. Continuations, Calculi, and Denotational Semantics

First-class program continuations are powerful additions to functional programming languages. They abstract a by-hand simulation of continuations [12] in a reliable way [10] and provide non-functional control devices like function exits and exception handlers. A major obstacle to the use of continuations has been the lack of a purely symbolic reasoning system for extended functional programming languages. The λ-calculus provides such a system for functional programs, eliminating extraneous concepts from program manipulations, for example, environments or interpreter continuations. Recently we have shown [2, 3] that a calculus is derivable for extended λ-calculus-based programming languages. This development has led to a fundamentally different way of perceiving interpreter continuations and continuation-accessing operations.

Convention. Functions in a denotational semantics that model the rest of the computation are called *continuation functions* or, when we think of the semantics as an interpreter, *interperter continuations.* When continuation functions are accessed via programming facilities and brought under control of a program, they become *continuation objects* or just *continuations.* **End**

As our starting point we had taken the functional subset of Scheme extended with the call-with-current-continuation (abbreviated call/cc) operator [6]. When applied to a function of one argument, call/cc encodes the current interpreter continuation into a function-like object, a *continuation object*, passes this object to its argument, and immediately installs the interpreter continuation. Upon application, a continuation object *discards the current interpreter continuation* and reinstalls the encapsulated one, so that unlike a true function, a continuation does *not* return a result to the point of application.

Although this strategy is efficient because of the *de facto* programming style with call/cc, it complicates the axioms for the extended λ-calculus. A first variation on call/cc, called $\mathcal{C}$ [2, 3], delegates the responsibility of installing the continuation to the program. It is equivalent to call/cc, but reduces the number of additional axioms from six to three and the remaining three are shortened. A further investigation reveals that another simplification of the axioms is possible if the continuation-accessing operation produces a *truly functional* encoding of the remaining computation. This new operator $\mathcal{F}$ can syntactically model $\mathcal{C}$ and call/cc and it can express programs that are not conceivable in the original call/cc-language.

Convention. Functional encodings of interpreter continuations under the control of a program are called *functional continuation objects* or just *functional continuations.* Their semantic counterparts are *compositional continuation functions.* **End**

At first glance, the concept of a continuation object that behaves like a function seems counterintuitive, but the simplification of the calculus is a significant argument in its favor. Programming language calculi are closer to the programmer's understanding of a language, and the simpler the calculus description for a specific programming facility, the easier it is to reason with the underlying concept. An argument against functional continuation objects is that the obvious attempt at a construction of a denotational continuation semantics fails. This, however, is a major inadequacy of continuation functions: the functional composition of two continuation functions does not yield a continuation function. A solution to the problem is therefore a denotational semantics with *compositional continuation functions.* We arrive at such a semantics by modifying the original derivation of the $\mathcal{C}$-λ_c-calculus from its denotational semantics in an appropriate manner for $\mathcal{F}$. The crucial idea in the development is the concept of a *functional abstraction of a continuation.* Roughly speaking, a functional abstraction of a continuation is the semantic image of a syntactic abstraction that performs the same actions as the respective continuation function, but upon completion resumes the current continuation. Since $\mathcal{F}$ subsumes $\mathcal{C}$ and $\mathcal{C}$ in turn is capable of explaining all traditional control facilities as syntactic abstractions, we conjecture that the domain of functional values suffices to model programming language control.

The main body of the paper is devoted to a presentation of the $\mathcal{C}$-calculus, its simplification to an $\mathcal{F}$-calculus, and the derivation of the denotational semantics for the latter. In the final section we summarize our work and address its relevance to language design. We assume some familiarity with denotational semantics [9] and Barendregt's notation and terminology [1] for the syntactic aspects of the λ-calculus.

2. A Variation on the Calculus for $\mathcal{C}$

The core of our programming language is an applicative-order concretization of the λ-calculus. The term set Λ is defined inductively over a set of variables *Var*:

$$L ::= x \mid \lambda x.M \mid MN.$$

We use the symbols $x, \ldots$ to range over *Var* but also as if they were elements of the set. $L, M, \ldots$ stand for Λ-terms.[1] Variables and λ-abstractions are collectively referred to as (syntactic) *values*.

The semantics of our programming language is defined by a denotational continuation semantics. Thus we need domains for the (semantic) values of abstractions, *Val*, continuation functions, *Cont*, and environments, *Env*:

$$\begin{aligned} \rho \in \mathit{Env} &= \mathit{Var} \longrightarrow \mathit{Val} \quad \text{(Environments)} \\ m, n, v \in \mathit{Val} &= \mathit{Val} \longrightarrow \mathit{Cont} \longrightarrow \mathit{Val} \quad \text{(Function Values)} \\ \kappa \in \mathit{Cont} &= \mathit{Val} \longrightarrow \mathit{Val} \quad \text{(Continuation Functions)} \end{aligned}$$

The semantic function is

$$\mathcal{E} : \Lambda \longrightarrow \mathit{Env} \longrightarrow \mathit{Cont} \longrightarrow \mathit{Val}$$

$$\begin{aligned} \mathcal{E}[\![x]\!] &= \lambda\rho\kappa.\kappa(\rho[\![x]\!]) && (\mathcal{E}.\text{var}) \\ \mathcal{E}[\![\lambda x.M]\!] &= \lambda\rho\kappa.\kappa(\lambda v.\mathcal{E}[\![M]\!]\rho[x \leftarrow v]) && (\mathcal{E}.\text{lam}) \\ \mathcal{E}[\![MN]\!] &= \lambda\rho\kappa.\mathcal{E}[\![M]\!]\rho(\lambda m.\mathcal{E}[\![N]\!]\rho(\lambda n.mn\kappa)) && (\mathcal{E}.\text{app}) \end{aligned}$$

[1] We consider Λ-terms equal if they are the same modulo the renaming of bound variables; we assume that free and bound variables never interfere [1].

Plotkin [5] has shown that the by-value variation of the λ-calculus is the correct medium for reasoning about programs in this language. Its basic notion of reduction is the β-value rule:

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

This notion of reduction can be extended to a congruence relation in the usual way and we write $M =_{\beta_v} N$ if M is convertible to N via some number of β_v-steps. The λ_v-calculus is correct with respect to the above denotational semantics in the following sense [5]:

- the standard reduction order of the λ_v-calculus produces the same program results as [an operational interpretation of] the denotational definition, and
- two terms that are equal under $=_{\beta_v}$ are operationally indistinguishable by an interpreter for $\mathcal{E}$, that is, the interpreter produces congruent answers for all computational contexts.

The principal extensions of our syntax are

$$L ::= \mathcal{C}M \text{ and } L ::= \mathcal{F}M.$$

$\mathcal{C}$ refers to our modified call/cc, $\mathcal{F}$ to the version that produces functional continuation objects. The denotational semantics of $\mathcal{C}$ is defined by extending $\mathcal{E}$ with

$$\mathcal{E}[\![\mathcal{C}M]\!] = \lambda\rho\kappa.\mathcal{E}[\![M]\!]\rho(\lambda m.m(\lambda v\kappa'.\kappa v)\mathrm{I}). \qquad (\mathcal{E}.\mathcal{C})$$

The derivation of a calculus proceeds in two major steps. First, based on well-known transformations [7, 11], we reinterpret the denotational semantics as an abstract machine. Second, we eliminate the environment and continuation component from this machine [2]. This yields a program rewriting system, the transition rules of which can be transformed into notions of reduction for a calculus.

The resulting calculus for $\mathcal{C}$ differs from the traditional λ_v-calculus in that it has two classes of term relations. First, there are the usual *notions of reduction*:

$$(\mathcal{C}M)N \rightarrow \mathcal{C}(\lambda k.M(\lambda f.(\mathcal{C}\lambda d.(k(fN))))) \qquad (\mathcal{C}_L)$$
$$M(\mathcal{C}N) \rightarrow \mathcal{C}(\lambda k.N(\lambda v.(\mathcal{C}\lambda d.(k(Mv))))) \quad \text{where } M \text{ is a value} \qquad (\mathcal{C}_R)$$

But, in order to imitate the machine correctly, the calculus includes a relation that is only applicable to entire terms. We refer to this kind of relation as *computation* and denote it with $\triangleright$ instead of the customary $\rightarrow$:

$$\mathcal{C}M \triangleright M(\lambda x.(\mathcal{C}\lambda d.x)). \qquad (\mathcal{C}_T)$$

If two terms M and N are equal according to (β_v), $(\mathcal{C}_L)$, $(\mathcal{C}_R)$, or $(\mathcal{C}_T)$, and any arbitrary mixture thereof, we write $M =_c N$. We refer to the calculus as λ_c-calculus. The calculus is Church-Rossser and has a standard reduction order. It is correct with respect to the denotational semantics in the above sense after some appropriate adaptations.

The additional reductions and computations of the λ_c-calculus are rather complex. Each right-hand side of the relation schemata contains a subterm of the form $\mathcal{C}(\lambda d.M)$ where d is not free in M. When evaluated, these subterms cause the reduction context to be collected and thrown away, acting like an *abort* operation. These aborts occur at the beginning of a continuation object invocation and thus achieve the proper discontinuous effect of continuation objects according to $(\mathcal{E}.\mathcal{C})$.

A brief investigation shows that a removal of these abort operations preserves an intact calculus. This new calculus has the same properties, except that, because we did not base it on a denotational semantics, we cannot claim its correctness with respect to such a semantics. The term relations are:

$$\mathcal{F}M \triangleright M(\lambda x.x) \qquad (\mathcal{F}_T)$$

$$(\mathcal{F}M)N \rightarrow \mathcal{F}(\lambda k.M(\lambda f.k(fN))) \qquad (\mathcal{F}_L)$$

$$M(\mathcal{F}N) \rightarrow \mathcal{F}(\lambda k.N(\lambda v.k(Mv))) \qquad \text{where } M \text{ is a value} \qquad (\mathcal{F}_R)$$

According to these equations, $\mathcal{F}$ constructs a functional abstraction from the context of an $\mathcal{F}M$-expression. Aborts are still equivalent to $\mathcal{F}(\lambda d.\_\_)$ and thus, $\mathcal{C}$ can be defined as a syntactic abstraction:

$$\mathcal{C}M \equiv \mathcal{F}(\lambda k.M(\lambda v.\mathcal{F}(\lambda d.kv))).$$

To prove the correctness of this implementation without a denotational semantics for $\mathcal{F}$, we must show that the syntactic equivalent of $\mathcal{C}M$ satisfies the axioms of $\mathcal{C}$ in the $\mathcal{F}$-λ_c-calculus. More precisely, the syntactic expansion of each of the two sides of the $\mathcal{C}$-term relations must be operationally indistinguishable in the $\mathcal{F}$-λ_c-calculus. Because of the lack of a semantics for $\mathcal{F}$, we hereby use the standard reduction order of the calculus as an operational semantics. Operationally indistinguishable then means that the standard reduction order for the two expressions leads to equivalent results. We call this *operational satisfaction of the axioms*:

Proposition 2.1. *$\mathcal{F}(\lambda k.M(\lambda v.\mathcal{F}(\lambda d.kv)))$ operationally satisfies the reduction and computation relations of $\mathcal{C}M$ in the $\mathcal{F}$-λ_c-calculus.*

It is impossible to model $\mathcal{F}$ with $\mathcal{C}$ by a syntactic equivalence. There is no semantically equivalent expression to $\mathcal{F}M$ that is obtained without knowledge of the structure of M and that otherwise is solely based on $\mathcal{C}$, λ-abstraction, and application. This is due to the abortive effect of continuation objects. If at any time during the simulation of a functional continuation a continuation object were used, it would discard the rest of the computation and hence, violate the semantics for $\mathcal{F}$. On the other hand, the functionality of the continuation, *i.e.*, the program context of $\mathcal{F}M$, is needed and can only be acquired through a $\mathcal{C}$-application. Therefore, $\mathcal{F}$ is not a syntactic variation on $\mathcal{C}$. If we admit a restructuring of M, then an implementation strategy could be based on a continuation-passing-like programming style. This question leads us to the problem of constructing a denotational semantics for $\mathcal{F}$.

The obvious denotational definition for $\mathcal{F}$ seems to be a modification to $(\mathcal{E}.\mathcal{C})$ such that the two continuations in the continuation object are functionally composed into a new continuation:

$$\mathcal{E}[\![\mathcal{F}M]\!] = \lambda\rho\kappa.\mathcal{E}[\![M]\!]\rho(\lambda m.m(\lambda v\kappa'.(\kappa' \circ \kappa)v)\mathrm{I}),$$

where $(\kappa' \circ \kappa)v \equiv \kappa'(\kappa v)$. However, this is wrong. Continuation functions must always encompass the *entire* remaining computation. In the expression $\kappa' \circ \kappa$ this no longer holds for κ as computations in there are not aware of κ'. Thus, if some program piece in κ grabs a continuation, it only has access to the continuation-information in κ and the rest in κ' is lost.

This problem is inherent in continuation semantics. The use of continuation functions in a denotational semantics is restricted syntactically. If a continuation function is applied to a value, then this application must happen at the root of the term. In programming jargon, the call must be tail-recursive. This guarantees that the continuation function really represents the entire rest of the computation. The above equation violates this restriction. When the modified continuation object is used, the κv application is embedded in the application of κ' to a value. In the next two sections we demonstrate how to construct a continuation-free semantics for $\mathcal{F}$, thus avoiding the above pitfall.

3. A Denotational Semantics for $\mathcal{F}$

In our previous work [2] we derived the $\mathcal{C}$-λ_c-calculus from a denotational-style machine for $\mathcal{C}$-Λ. The transition function is displayed in Figure 1. The initial configuration for the evaluation of a program M is $\langle M, \emptyset, (\mathbf{stop})\rangle$; the machine stops when it reaches the state $\langle \ddagger, \emptyset, ((\mathbf{stop})\ \mathbf{ret}\ V)\rangle$ for some value V. Implementing $\mathcal{F}$

Figure 1: The CEK-transition function

$$\langle x, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa\,\mathbf{ret}\ \rho(x))\rangle \quad (1)$$

$$\langle \lambda x.M, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa\,\mathbf{ret}\ \langle \lambda x.M, \rho\rangle)\rangle \quad (2)$$

$$\langle MN, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle M, \rho, (\kappa\,\mathbf{arg}\ N\rho)\rangle \quad (3)$$

$$\langle \ddagger, \emptyset, ((\kappa\,\mathbf{arg}\ N\rho)\,\mathbf{ret}\ F)\rangle \overset{CEK}{\longmapsto} \langle N, \rho, (\kappa\,\mathbf{fun}\ F)\rangle \quad (4)$$

$$\langle \ddagger, \emptyset, ((\kappa\,\mathbf{fun}\ \langle \lambda x.M, \rho\rangle)\,\mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle M, \rho[x := V], \kappa\rangle \quad (5)$$

$$\langle \mathcal{C}M, \rho, \kappa\rangle \overset{CEK}{\longmapsto} \langle Mc, \rho[c := \langle \mathrm{p}, \kappa\rangle], (\mathbf{stop})\rangle, \quad c \notin Dom(\rho) \quad (6)$$

$$\langle \ddagger, \emptyset, ((\kappa\,\mathbf{fun}\ \langle \mathrm{p}, \kappa_0\rangle)\,\mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa_0\,\mathbf{ret}\ V)\rangle \quad (7)$$

requires a modification of the transitions that specify the behavior of continuations. One solution is changing (CEK7) so that the invoked continuation is concatenated to the current one:

$$\langle \ddagger, \emptyset, ((\kappa\,\mathbf{fun}\ \langle \mathrm{p}, \kappa_0\rangle)\,\mathbf{ret}\ V)\rangle \overset{CEK}{\longmapsto} \langle \ddagger, \emptyset, (\kappa \otimes \kappa_0\,\mathbf{ret}\ V)\rangle \qquad (\mathrm{CEK}_{\mathcal{F}}7).$$

The result of $\kappa \otimes \kappa_0$ is like κ_0 with the $(\mathbf{stop})$-continuation replaced by κ. Although this machine is a correct implementation of the $\mathcal{F}$-λ_c-calculus, this construction does not lead to a denotational semantics. Unlike continuation data structures in the CEK-machine, a continuation function in denotational semantics is opaque. There is no algorithm comparable to $\otimes$ that produces the correct combination of two continuation functions.

Since we know that in the $\mathcal{F}$-λ_c-calculus $\mathcal{F}$ constructs a functional abstraction of the current continuation, a second implementation strategy is feasible. Instead of passing the $\mathcal{F}$-argument an encoded continuation structure and reinterpreting the application of those, the CEK$_{\mathcal{F}}$-machine can supply a machine closure[2] to the $\mathcal{F}$-argument. This closure must simulate the λ-abstraction in the $\mathcal{F}$-λ_c-calculus that corresponds to the current continuation. The necessary and guiding condition for

[2] A machine closure is the operational value of an abstraction.

the construction of the new machine is that the modified CEK$_{\mathcal{F}}$-machine and the original one produce congruent values.

A transliteration of this second operational definition leads to a natural denotational specification of $\mathcal{F}$:

$$\mathcal{E}[\![\mathcal{F}M]\!] = \lambda\rho\kappa.\mathcal{E}[\![M]\!]\rho(\lambda m.m\gamma\mathrm{I}), \tag{$\mathcal{E}.\mathcal{F}$}$$

where γ stands for a function value which corresponds to the continuation κ. Intuitively, correspondence means, that γ, when invoked, acts like κ but resumes the current continuation upon completion. This latter condition holds by definition for true images of λ-abstractions. Given that $\mathcal{E}$ maps abstractions to function values, we can formalize the notion of correspondence:

Definition 3.1. *A function value* γ corresponds to a continuation κ, $\gamma \sim \kappa$, *if there exists a closed λ-abstraction c such that $\kappa'\gamma = \mathcal{E}[\![c]\!]\rho\kappa'$ for all κ', ρ and $\gamma v\mathrm{I} = \kappa v$ for all v. We refer to* γ *as* a functional abstraction of the continuation κ.

This definition precludes the trivial correspondence $\lambda x\kappa'.\kappa x \sim \kappa$.

The definition of $\sim$ is implicit. In order to reconstruct the corresponding functional abstraction γ for a continuation function κ, we need to know how the continuation function was built. Fortunately, there are only three possible ways to construct a continuation in the semantics of our base language:

$$\lambda x.x,\ \ \lambda m.\mathcal{E}[\![N]\!]\rho(\lambda n.mn\kappa),\ \ \text{and}\ \lambda n.mn\kappa,$$

where the free κ is a continuation and m is some value. If we let c be an abstraction that denotes a function value γ and γ corresponds to κ, then the three pre-images

$$\lambda x.x,\ \ \lambda m.c(mN),\ \ \text{and}\ \lambda n.c(mn),$$

respectively, define the corresponding function values. In other words, we can now use $\mathcal{E}$ to find the right functional abstractions of the continuations:

Proposition 3.2. *If c denotes γ and $\gamma \sim \kappa$, then for the appropriate ρ we have*

$$\begin{aligned}\mathcal{E}[\![\lambda x.x]\!]\rho\mathrm{I} &= \lambda x\kappa'.\kappa' x \sim \lambda x.x\\ \mathcal{E}[\![\lambda m.c(mN)]\!]\rho\mathrm{I} &= \lambda m\kappa'.\mathcal{E}[\![N]\!]\rho(\lambda m.mn(\lambda x.\gamma x\kappa')) \sim \lambda m.\mathcal{E}[\![N]\!]\rho(\lambda n.mn\kappa)\\ \mathcal{E}[\![\lambda n.c(mn)]\!]\rho\mathrm{I} &= \lambda n\kappa'.mn(\lambda x.\gamma x\kappa') \sim \lambda n.mn\kappa\end{aligned}$$

With this proposition we have explicated the relationship between functional abstractions and continuations, but the definition $\mathcal{E}.\mathcal{F}$ is still implicit because of the opaqueness of continuation functions. The crucial idea is to replace continuations by a pair consisting of the old-style continuation function and its corresponding functional abstraction. The induced adaptation of the notion of correspondence is negligible since the evaulation of abstractions ignores the additional component. The necessary change to *Val* is

$$Val = Val \longrightarrow Cont \times Val \longrightarrow Val.$$

We call the new semantic function $\mathcal{E}_{cf}$ and define it as:

$$\mathcal{E}_{cf} : \Lambda \longrightarrow Env \longrightarrow Cont \times Val \longrightarrow Val$$

$$\mathcal{E}_{cf}[\![x]\!] = \lambda\rho\kappa\gamma.\kappa(\rho[\![x]\!]) \qquad (\mathcal{E}_{cf}.\text{var})$$
$$\mathcal{E}_{cf}[\![\lambda x.M]\!] = \lambda\rho\kappa\gamma.\kappa(\lambda v.\mathcal{E}_{cf}[\![M]\!]\rho[x \leftarrow v]) \qquad (\mathcal{E}_{cf}.\text{lam})$$

and

$$\begin{aligned}\mathcal{E}_{cf}[\![(MN)]\!] = \lambda\rho\kappa\gamma.\mathcal{E}_{cf}&[\![\lambda m.c(mN)]\!]\rho[c \leftarrow \gamma] \qquad (\mathcal{E}_{cf}.\text{app})\\ &(\lambda\gamma_m.\mathcal{E}_{cf}[\![M]\!]\rho\\ &\quad(\lambda m.\mathcal{E}_{cf}[\![\lambda n.c(mn)]\!]\rho[c \leftarrow \gamma][m \leftarrow m]\\ &\quad\quad(\lambda\gamma_n.\mathcal{E}_{cf}[\![N]\!]\rho(\lambda n.mn\kappa\gamma)\gamma_n)^*)\gamma_m)^*.\end{aligned}$$

In the last clause, we first build the functional abstractions of the respective continuations. Since this maps abstractions to values, the additional applications of $\mathcal{E}_{cf}$ ignore their corresponding functional abstractions. We use * to indicate this. Furthermore, because these applications yield values, they may be eliminated by explicitly computing the results. The final version of ($\mathcal{E}_{cf}$.app) thus becomes:

$$\begin{aligned}\mathcal{E}_{cf}[\![MN]\!] = \lambda\rho\kappa\gamma.\mathcal{E}_{cf}&[\![M]\!]\rho \qquad (\mathcal{E}_{cf}.\text{app})\\ &(\lambda m.\mathcal{E}_{cf}[\![N]\!]\rho(\mathbf{B}_{cont}\kappa\gamma m)(\mathbf{B}_{val}\gamma m))\\ &(\lambda m\kappa'\gamma'.\mathcal{E}_{cf}[\![N]\!]\rho(\mathbf{B}_{cont}(\mathbf{B}_{cont}\kappa'\gamma'\gamma)(\mathbf{B}_{val}\gamma'\gamma)m)(\mathbf{B}_{val}(\mathbf{B}_{val}\gamma'\gamma)m)).\end{aligned}$$

The two combinators $\mathbf{B}_{val}$ and $\mathbf{B}_{cont}$ compose function values and continuations:

$$\begin{aligned}(\mathbf{B}_{val}fg) &= \lambda x\kappa\gamma.gx(\mathbf{B}_{cont}\kappa\gamma f)(\mathbf{B}_{val}\gamma f),\\ (\mathbf{B}_{cont}\kappa\gamma f) &= \lambda x.fx\kappa\gamma.\end{aligned}$$

We therefore call functional abstractions of continuations ***compositional continuation functions.***

Let $\hat{\mathbf{I}}$ stand for the compositional continuation function that corresponds to the intial continuation $\mathbf{I}(\equiv \lambda x.x)$. Then in the definition for $\mathcal{F}$ we can simply shift the compositional continuation function to the program:

$$\mathcal{E}_{cf}[\![\mathcal{F}M]\!] = \lambda\rho\kappa\gamma.\mathcal{E}_{cf}[\![M]\!]\rho(\lambda m.m\gamma\mathbf{I}\hat{\mathbf{I}})(\lambda m.m\gamma). \qquad (\mathcal{E}_{cf}.\mathcal{F})$$

Now we can state and prove our central correctness theorem:

Theorem 3.3. *The $\mathcal{F}$-λ_c-calculus is correct with respect to the denotational semantics of the $\mathcal{F}$-Λ-language.*

Of course, we can also translate the defining clause for $\mathcal{C}$ into the new style:

$$\mathcal{E}_{cf}[\![\mathcal{C}M]\!] = \lambda\rho\kappa\gamma.\mathcal{E}_{cf}[\![M]\!]\rho(\lambda m.m(\lambda x\kappa'\gamma'.\kappa x)\mathbf{I}\hat{\mathbf{I}})(\lambda m.m(\lambda x\kappa'\gamma'.\kappa x)). \qquad (\mathcal{E}_{cf}.\mathcal{C})$$

As an immediate consequence Proposition 2.1 can be restated and reproved:

Corollary 3.4. *For all ρ, κ, and γ,*

$$\mathcal{E}_{cf}[\![\mathcal{F}(\lambda k.M(\lambda v.\mathcal{F}(\lambda d.(kv))))]\!]\rho\kappa\gamma = \mathcal{E}_{cf}[\![\mathcal{C}M]\!]\rho\kappa\gamma.$$

4. Continuations are Unnecessary

In the previous section we introduced the concept of a compositional continuation function, a functional value corresponding to the current continuation function. The important characteristic of compositional continuation functions is captured in the second clause of Definition 3.1. It says that a functional abstraction of a continuation delivers the same final answer as the continuation when invoked on the initial continuation function, *i.e.*, in the current framework

$$\gamma \sim \kappa \text{ implies } \gamma v\mathbf{I}\hat{\mathbf{I}} = \kappa v.$$

Replacing applications of continuation functions by applications of the corresponding functional abstractions preserves this property with one exception: the initial continuation function. It becomes $\hat{\mathbf{I}} = \lambda x\kappa\gamma.\gamma x\mathbf{I}\hat{\mathbf{I}}$. Whereas for the original $\hat{\mathbf{I}}$ it is true that

$$\hat{\mathbf{I}}x\kappa\gamma = \kappa x, \text{ and, in particular, } \hat{\mathbf{I}}x\mathbf{I}\hat{\mathbf{I}} = \mathbf{I}x = x,$$

this no longer holds for the new version. To avoid this problem, we introduce $\hat{\mathbf{I}}$ as a unique object in the *Val* domain, thus making it testable with a predicate $is\hat{\mathbf{I}}$, and define it by

$$\hat{\mathbf{I}} = \lambda x\kappa\gamma.is\hat{\mathbf{I}}\,\gamma \rightarrow x, \gamma x\mathbf{I}\hat{\mathbf{I}}.$$

This version again satisfies the above equations for $\hat{\mathbf{I}}$. To get the types correct, we must modify the definition of $\mathbf{B}_{val}$ appropriately.

At this point continuation functions are superfluous since they are never invoked. The *Val* domain changes to:

$$Val = [\,Val \longrightarrow Val \longrightarrow Val] + \{\hat{\mathbf{I}}\}.$$

The continuation-free version of $\mathcal{E}_{cf}$ is $\mathcal{E}_f$:

$$\mathcal{E}_f : \Lambda \longrightarrow Env \longrightarrow Val \longrightarrow Val$$

$$\begin{aligned}
\mathcal{E}_f[\![x]\!] &= \lambda\rho\gamma.\gamma(\rho[\![x]\!])\hat{\mathbf{I}} && (\mathcal{E}_f.\text{var})\\
\mathcal{E}_f[\![\lambda x.M]\!] &= \lambda\rho\gamma.\gamma(\lambda v.\mathcal{E}_f[\![M]\!]\rho[x \leftarrow v])\hat{\mathbf{I}} && (\mathcal{E}_f.\text{lam})\\
\mathcal{E}_f[\![MN]\!] &= \lambda\rho\gamma.\mathcal{E}_f[\![M]\!]\rho(\lambda m\gamma_m.\mathcal{E}_f[\![N]\!]\rho(\mathbf{B}_{val}\gamma_m(\mathbf{B}_{val}\gamma m))) && (\mathcal{E}_f.\text{app})\\
\mathcal{E}_f[\![\mathcal{F}M]\!] &= \lambda\rho\gamma.\mathcal{E}_f[\![M]\!]\rho(\lambda m.m\gamma) && (\mathcal{E}_f.\mathcal{F})
\end{aligned}$$

The final definitions of the initial compositional continuation function and the auxiliary combinator $\mathbf{B}_{val}$ are:

$$\begin{aligned}
\hat{\mathbf{I}} &= \lambda x\gamma.is\hat{\mathbf{I}}\;\gamma \rightarrow x, \gamma x\hat{\mathbf{I}}\\
\mathbf{B}_{val} &= \lambda fg.is\hat{\mathbf{I}}\,g \rightarrow f, \lambda xh.gx(\mathbf{B}_{val}hf)
\end{aligned}$$

The equivalence of the two definitions is captured in a congruence theorem:

Theorem 4.1. *$\mathcal{E}_{cf}$ and $\mathcal{E}_f$ yield congruent values.*

5. Conclusions

In the preceding sections we have shown how the simplification of the calculus for continuations leads to a powerful programming language control construct. With it we can write a new class of programs that are impossible to express with classical continuations. To construct a suitable denotational semantics, we have introduced the concept of functional abstractions of continuations. We have thus demonstrated that function values can entirely replace continuations within denotational descriptions of control constructs.

Beyond these immediate consequences our development has also illustrated that the consideration of a programming language calculus is important for the design of a programming language. While a denotational semantics is of importance to the implementor, a calculus is closer to the consumer's perspective. We propose language calculi as yet another dimension along which to measure language design.

References

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

2. FELLEISEN, M., D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus, *Formal Description of Programming Concepts III*, North-Holland, Amsterdam, 1986, to appear.

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

4. LANDIN, P.J. An abstract machine for designers of computing languages, *Proc. IFIP Congress*, 1965, 438–439.

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

6. REES J., W. CLINGER. (Eds.) The revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21**(11), 1986, to appear.

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

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

9. STOY, J.E. *Denotational Semantics: The Scott-Stratchey Approach to Programming Languages*, The MIT Press, Cambridge, Massachusetts, 1981.

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.

11. WAND, M., D.P. FRIEDMAN. Compiling lambda-expressions using continuations and factorizations, *Computer Languages* **3**, 1978, 241–263.

12. WAND, M. Continuation-based program transformation strategies, *JACM* **27** (1), 1980, 164–180.