

# The Theory and Practice of First-Class Prompts

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

**Abstract.** An analysis of the  $\lambda_v$ -C-calculus and its problematic relationship to operational equivalence leads to a new control facility: the *prompt-application*. With the introduction of prompt-applications, the control calculus becomes a traditional calculus all of whose equations imply operational equivalence. In addition, prompt-applications enhance the expressiveness and efficiency of the language. We illustrate the latter claim with examples from such distinct areas as systems programming and tree processing.

## 1 A Problem of the $\lambda_v$ -C-calculus

The  $\lambda_v$ -C-calculus [2, 4] is a conservative extension of Plotkin's  $\lambda_v$ -calculus [11] for reasoning about imperative control operations in call-by-value languages. The theory satisfies modified versions of the Church-Rosser and the Curry-Feys Standardization Theorem. Furthermore, the standardization procedure yields the same answers for programs as the evaluation function of an appropriate abstract machine. However, the calculus also has a major deficiency: equality in the calculus does *not* imply operational equality on the machine. That is, two expressions that are equal

---

\*This work was partly supported by an IBM Fellowship and Dan Friedman's NSF grant (DCR 85 01277). *Author's current address:* Rice University, Herrman Brown Hall, Houston, TX 77251-1892

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

in the calculus do not necessarily behave equivalently in all program contexts. The problem can be fixed with a meta-theorem that distinguishes theorems in the theory as operational equalities, yet, this is not a desirable situation.

In this paper, we present a different and more promising solution. It is based on a new linguistic facility for constraining the extent of control operations: the prompt-application. The introduction of prompt-applications transforms the control calculus into a truly traditional calculus, and, in addition, prompt-applications are a practical tool for a broad variety of programming paradigms. We illustrate the latter claim with examples from such distinct areas as systems programming and tree processing.

In the next section, we formalize the original  $\lambda_v$ -C-calculus and the concept of a first-class prompt facility. We then show that the extended calculus is consistent and that there is a standard reduction function. Based on this, we define an operational semantics and show that theorems in the calculus directly imply operational equivalence. The third section contains an abstract stack-machine for the extended language and a brief discussion of the implementation cost. Section 4 is a collection of programming examples, which illustrates the expressive power of prompt-applications. The fifth section is an overview of related work. In the conclusion, we analyze the contribution of language calculi to the systematic analysis of programming languages.

## 2 The modified $\lambda_v$ -C-calculus

The term set  $\Lambda_{\mathcal{F}}$  of the original  $\lambda_v$ -C-calculus is defined inductively over a set of algebraic constants, *Const*, and a set of variables, *Vars*:

$$L ::= a \mid x \mid \lambda x.M \mid MN \mid \mathcal{F}M,$$

where  $a$  ranges over constants,  $x$  over *Vars*, and  $L$ ,  $M$ , and  $N$  over  $\Lambda_{\mathcal{F}}$ -terms. The first four classes of expres-

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)
---

sions have their original semantics: constants stand for basic and functional constants from some kind of algebraic domain, variables are placeholders,  $\lambda$ -abstractions represent call-by-value procedures, and combinations denote function applications.

As usual, the variable  $x$  in the abstraction  $\lambda x.M$  is called the *bound* variable; a variable that is not bound by some  $\lambda$  is *free*. An expression with no free variables is called *closed*. Programs in this language are closed expressions. Like Barendregt [1:26], we identify terms that are identical modulo bound variables, and we assume that free and bound variables cannot interfere with each other. The substitution of a free variable  $x$  by  $N$  in a term  $M$  is denoted by  $M[x := N]$ .

The new expression type is called  $\mathcal{F}$ -application,<sup>1</sup> its subterm is an  $\mathcal{F}$ -argument. An  $\mathcal{F}$ -application transforms the current control state into a *functional abstraction*, which we call *continuation*.<sup>2</sup> Next, the  $\mathcal{F}$ -application applies its argument to this continuation, thus providing the  $\mathcal{F}$ -argument with total power over the rest of the computation. In particular, if the corresponding actions are to be performed at all, the  $\mathcal{F}$ -argument must invoke this continuation.

The equational theory is specified with a set of term relations for the evaluation of expressions. Constants, variables, and  $\lambda$ -abstractions require no further evaluation and are appropriately referred to as *values*. For the evaluation of an application of a function constant to a basic constant, we assume the existence of an interpretation  $\delta$  on the set of constants:

$$\delta: \text{FuncConst} \times \text{BasicConst} \rightarrow \text{Closed-}\Lambda\text{-Values.}$$

The  $\delta$ -reduction for constant applications is:

$$\delta: fa \rightarrow \delta(f, a).$$

The evaluation of an application of a  $\lambda$ -abstraction to a value is determined by the  $\beta$ -value relation:

$$\beta: (\lambda x.M)V \rightarrow M[x := V] \quad \text{provided } V \text{ is a value.}$$

That is, an application whose first part is an abstraction and whose second part is a value is reduced to a term that is like the function body with all occurrences of the abstracted variable replaced by the value.

The meaning of  $\mathcal{F}$ -applications is determined by two reductions and a special computation relation. The reductions are:

$$\mathcal{F}_L: (\mathcal{F}M)N \rightarrow \mathcal{F}(\lambda k.M(\lambda m.k(mN))),$$

<sup>1</sup> $\mathcal{F}$  is a generalized version of Iswim's J [8], Reynolds's escape [13], and Scheme's [12] call/cc.

<sup>2</sup>WARNING: This usage is inconsistent with Scheme-terminology.

$$\mathcal{F}_R: V(\mathcal{F}M) \rightarrow \mathcal{F}(\lambda k.M(\lambda m.k(Vm)))$$

provided  $V$  is a value.

The purpose of these reductions is to push an  $\mathcal{F}$ -application to the root of a term and to encode the context of the  $\mathcal{F}$ -application as an abstraction. Once the  $\mathcal{F}$ -application has reached the root, the computation relation

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

eliminates the  $\mathcal{F}$ -application by applying the  $\mathcal{F}$ -argument to the identity function, a representation of the empty context. The computation rule is denoted with  $\triangleright$  instead of the customary  $\rightarrow$  because it can only be applied to the root: *an unrestricted use would make the calculus inconsistent*.

The nature of  $\mathcal{F}$ -applications and continuations is best illustrated with examples. Consider the program  $(1^+(\mathcal{F}(\lambda d.0)))$ . It evaluates to 0 after the  $\mathcal{F}$ -application erases the call to  $1^+$ :

$$\begin{aligned} (1^+(\mathcal{F}(\lambda d.0))) &\rightarrow \mathcal{F}(\lambda k.(\lambda d.0)(\lambda v.k(1^+v))) \\ &\triangleright (\lambda k.((\lambda d.0)(\lambda v.k(1^+v))))(\lambda x.x) \\ &\rightarrow (\lambda d.0)(\lambda v.(\lambda x.x)(1^+v)) \\ &\rightarrow 0. \end{aligned}$$

Put more abstractly,  $\mathcal{F}(\lambda d.V)$  where  $d$  does not occur in the value  $V$  behaves like an **abort** operation that returns  $V$ . The conventional **goto** is a straightforward generalization of **abort**. If we replace the value  $V$  with an arbitrary expression  $M$ , the term  $\mathcal{F}(\lambda d.M)$  first eliminates the current state and then installs  $M$  as the new one.

When a continuation is applied, it acts like an ordinary function. It performs the encoded rest of the computation and, unless an  $\mathcal{F}$ -application intervenes, *returns to the point of invocation*:

$$\begin{aligned} &(1^+(\mathcal{F}(\lambda k.(k(k0)))))) \\ &\rightarrow (\mathcal{F}(\lambda k.(\lambda k.(k(k0))))(\lambda x.k(1^+x))) \\ &\triangleright (\lambda k.((\lambda k.(k(k0)))(\lambda x.k(1^+x))))(\lambda x.x) \\ &\rightarrow ((\lambda k.(k(k0)))(\lambda x.(\lambda x.x)(1^+x))) \\ &\rightarrow ((\lambda x.(\lambda x.x)(1^+x))((\lambda x.(\lambda x.x)(1^+x))0)) \\ &\rightarrow ((\lambda x.(\lambda x.x)(1^+x))((\lambda x.x)(1^+0))) \\ &\rightarrow ((\lambda x.(\lambda x.x)(1^+x))1) \\ &\rightarrow (\lambda x.x)(1^+1) \\ &\rightarrow (\lambda x.x)2 \\ &\rightarrow 2. \end{aligned}$$

The problem of this calculus is the separation of computation and reduction relations. Since computation relations can only be applied at the root of a term, an equation that is based on the *computation* relation does not imply operational equivalence. More precisely, even if two terms are *computationally* equivalent, there are generally program contexts that can distinguish the two expressions. A typical example is the pair of terms  $\mathcal{F}(\lambda d.0)$  and  $0$ . Although equivalent according to the computation relation, they behave differently in practically all contexts. Whereas the former aborts a computation when evaluated, the second one simply evaluates to  $0$ .

A possible and obvious fix is the introduction of a unique top-level marker  $\#$  for identifying the root of a program. Then the computation relation can be replaced by a reduction relation:

$$\#_{\mathcal{F}} : (\#(\mathcal{F}M)) \longrightarrow (\#(M(\lambda x.x))).$$

However, this solution pushes the semantic division into the language syntax and does not eliminate the complications of its predecessor: equations for  $\#$ -contained terms cannot predict the behavior of these terms in arbitrary contexts. A true solution must go further:  $\#$ -applications must become a new kind of *first-class* construct that can occur anywhere in the program text.

The term language  $\Lambda_{\mathcal{F}\#}$  for the modified calculus is a minor extension of  $\Lambda_{\mathcal{F}}$ :

$$L ::= x \mid \lambda x.M \mid MN \mid \mathcal{F}M \mid \#M.$$

Programs are now identified as the set of closed  $\#$ -applications.

**Remark.** A program  $M$  in  $\Lambda_{\mathcal{F}}$  corresponds to a program  $\#M$  in  $\Lambda_{\mathcal{F}\#}$ . **End**

Since programs should reduce to proper values, we need an additional reduction relation for returning an evaluated  $\#$ -argument:

$$\#_v : (\#V) \longrightarrow V \quad \text{provided } V \text{ is a value.}$$

If we assume that an interactive system for this language automatically supplies the outermost  $\#$ -part for a program, a dialogue looks like

$$\begin{array}{l} \# M_1 \\ V_1 \\ \# M_2 \\ V_2 \\ \dots \end{array}$$

In other words,  $\#$  corresponds to a prompt, and we therefore call  $(\#M)$  a *prompt-application*.

With this modification, the  $\lambda_v$ -C-calculus becomes a calculus in the traditional sense:

**Definition.** (*The new  $\lambda_v$ -C-calculus*) The basic notion of reduction  $c$  collects all of the above relations:

$$c : c = \delta \cup \beta_v \cup \mathcal{F}_L \cup \mathcal{F}_R \cup \#_{\mathcal{F}} \cup \#_v.$$

The one-step  $c$ -reduction,  $\longrightarrow_c$ , is the compatible closure of  $c$ :

$$\begin{aligned} (M, N) \in c &\Rightarrow M \longrightarrow_c N \\ M \longrightarrow_c N &\Rightarrow \lambda x.M \longrightarrow_c \lambda x.N \\ M \longrightarrow_c N &\Rightarrow \mathcal{F}M \longrightarrow_c \mathcal{F}N \\ M \longrightarrow_c N &\Rightarrow \#M \longrightarrow_c \#N \\ M \longrightarrow_c N &\Rightarrow LM \longrightarrow_c LN, ML \longrightarrow_c NL; \end{aligned}$$

$\longrightarrow_c$  stands for the transitive-reflexive closure of the one-step  $c$ -reduction.

The  $c$ -congruence is defined as:

$$\begin{aligned} M &=_c M \\ M \longrightarrow_c N &\Rightarrow M =_c N \\ M =_c N &\Rightarrow N =_c M \\ L =_c M, M =_c N &\Rightarrow L =_c N. \end{aligned}$$

Formally, the  $\lambda_v$ -C-calculus is defined as the congruence relation  $=_c$ ; informally, we refer to the entire set of relations as  $\lambda_v$ -C-calculus.

**Remark.** There are normal forms that are not values and values that are not normal forms, e.g.,  $\mathcal{F}(\lambda x.x)$  and  $\lambda y.(\lambda x.xx)(\lambda x.xx)$ . **End**

Unlike its predecessor, this calculus has the ordinary Church-Rosser Property:

**Theorem (Church-Rosser for  $\lambda_v$ -C).** *The (modified)  $\lambda_v$ -C-calculus is Church-Rosser, i.e.,  $\longrightarrow_c$  satisfies the diamond property.*

**Proof.** From our previous work [2] we know that

$$c' = \delta \cup \beta_v \cup \mathcal{F}_L \cup \mathcal{F}_R$$

is Church-Rosser. Furthermore, it is obvious that

$$c'' = \#_{\mathcal{F}} \cup \#_v$$

is Church-Rosser, and also that the reductions based on  $c'$  and  $c''$  commute. Given this, it follows from the Hindley-Rosen Lemma [1:64] that the reduction  $\longrightarrow_c$  satisfies the diamond property.  $\square$

Furthermore, the modified calculus has the usual standardization procedure for equations. For every

(one-way) derivation, there exists a standard derivation based on the leftmost-outermost reduction strategy. A convenient way of defining leftmost-outermost reductions is based on the concept of an evaluation context:

**Definition.** (*Contexts and evaluation contexts*) *Contexts* are terms with a hole. Let  $C[ ]$  range over the set of contexts and let  $[ ]$  denote the hole. Then we can define the set of contexts inductively by:

$$C[ ] ::= [ ] \mid \lambda x.C[ ] \mid MC[ ] \mid C[ ]M \mid \mathcal{F}C[ ] \mid \#C[ ].$$

If  $C[ ]$  is a context, then  $C[M]$  is the term where the hole is filled with the term  $M$ . The filling of a context may capture free variables in the fill-in term.

*Evaluation contexts* are special contexts whose hole is not inside a  $\lambda$ -abstraction or an  $\mathcal{F}$ -application. More precisely, the path from the root to the hole leads through applications and  $\#$ -applications only and the terms to the left of the path are values:

$$C[ ] ::= [ ] \mid VC[ ] \mid C[ ]M \mid \#C[ ],$$

where  $M$  is an arbitrary term and  $V$  is a value.

The concept of an evaluation context precisely captures the notion of leftmost-outermost: if  $P$  is a  $\mathbf{C}$ -redex and  $C[ ]$  is an evaluation context, then  $P$  is the unique leftmost-outermost redex in  $C[P]$ . Given this, it is straightforward to define a standard reduction function and a set of standard reduction sequences. The former reduces the leftmost-outermost redex of a given term, the latter combines a series of terms that are related by standard reduction steps, possibly by omitting the reduction of some leftmost-outermost redexes:

**Definition.** (*Standard reduction function and standard reduction sequences*) The standard reduction function maps  $M$  to  $N$ ,  $M \mapsto_{sc} N$ , if there are  $P$ ,  $Q$ , and an evaluation context  $C[ ]$  such that

$$(P, Q) \in \mathbf{c},$$

and

$$M \equiv C[P], \quad N \equiv C[Q].$$

*Standard reduction sequences*, abbreviated SRS-s, are defined by:

1. all constants and variables are SRS-s;
2. if  $M_1, \dots, M_m$  is an SRS and  $N_1, \dots, N_n$  is an SRS, then the following are SRS-s:

$$\lambda x.M_1, \dots, \lambda x.M_m,$$

$$\begin{aligned} &\mathcal{F}M_1, \dots, \mathcal{F}M_m, \\ &\#M_1, \dots, \#M_m, \text{ and} \\ &M_1N_1, \dots, M_mN_n; \end{aligned}$$

3. if  $M \mapsto_{sc} M_1$  and  $M_1, \dots, M_m$  is an SRS, then  $M, M_1, \dots, M_m$  is an SRS.

The Standardization Theorem can now be formulated as:

**Theorem (Standardization for  $\lambda_v$ -C).**

$M \mapsto_c N$  if and only if there is a standard reduction sequence  $L_1, \dots, L_l$  such that  $M \equiv L_1$  and  $L_l \equiv N$ .

**Proof.** A full-fledged proof requires an adaptation of the standardization proof for the original reduction-based  $\lambda_v$ -C-subcalculus. All arguments must be extended to cover the two new reductions,  $\#_{\mathcal{F}}$  and  $\#_v$ . Informally, we can argue along the following lines. Suppose the standard reduction sequence from  $\#P$  to  $Q$  is needed. This sequence can be obtained with the old standardization procedure by replacing all computation steps with  $\#_{\mathcal{F}}$ -reductions. If  $Q$  is of the form  $\#Q_1$ , then we have obtained the required standard reduction sequence. Otherwise  $Q$  is a value. In this case, the tail of the reduction sequence is a transformation of a value inside of a  $\#$ -application. By using a  $\#_v$ -step as early as possible, we standardize the reduction sequence. All other standardization steps stay the same.  $\square$

The Standardization Theorem implies that a program has a value if and only if we can reduce it to a value by always reducing the leftmost-outermost redex:

**Corollary (Standard evaluation).** *If  $M$  has a value, i.e., there is a value  $U$  such that  $\lambda_v\text{-C} \vdash M = U$ , then the transitive-reflexive closure of the standard reduction function relates  $M$  to a value  $V$ :*

$$M \mapsto_{sc}^* V.$$

In other words, the standard reduction function associates a unique value with a program and can thus be construed as an abstract machine semantics. Based on this, we define the following evaluation function:

**Definition.** (*The  $eval_c$ -function*) The (partial) evaluation function  $eval_c$  maps the program  $M$  to a value  $V$ :

$$eval_c(M) = V \text{ if } M \mapsto_{sc}^* V.$$

This evaluation function is equivalent to a programmer's perception of an interpreter. It characterizes the observable behavior of programs and thus

determines an *operational equivalence* relation on programs and terms. The formalization of this equivalence relation follows Plotkin's [11] corresponding work on the  $\lambda_v$ -calculus. Operational equivalence equates two terms if one can replace the other in any arbitrary program context without affecting the final value of the program. In order to make this definition effective, we restrict our attention to basic constants as answers. This is also justified by the fact that a function is generally considered as an intermediate result for subsequent computations. Putting this together, we get:

**Definition. (Operational equivalence)**  $M, N \in \Lambda_{\mathcal{F}\#}$  are operationally equivalent,  $M \simeq_C N$ , if for any context  $C[\ ]$  such that  $C[M]$  and  $C[N]$  are programs,  $eval_c$  is undefined on both, or it is defined on both and if one of the programs yields a basic constant, then the value of the other is the same basic constant.

Operational equivalence is indeed omni-present in the work of a programmer. For any kind of program transformation, e.g., for making a program shorter, faster, better looking, etc., the operational semantics of the program must be preserved: the two versions must be operationally equivalent. Because of this, it is necessary to have an equational theory for reasoning systematically about operational equivalences. The modified  $\lambda_v$ -C-calculus is such a theory. Theorems in the calculus imply operational equivalence for the terms according to the evaluation function:

**Theorem (Correspondence).** *If  $\lambda_v$ -C  $\vdash M = N$  then  $M \simeq_C N$ . The converse does not hold.*

**Proof.** The proof is exactly the same as the one for the  $\lambda_v$ -calculus [11:144]. If  $\lambda_v$ -C  $\vdash M = N$ , then  $\lambda_v$ -C  $\vdash C[M] = C[N]$ . Hence, if  $eval_c(C[M])$  is defined, so is  $eval_c(C[N])$  (though they may not be the same). Finally, if one of the values is a basic constant, the other must be the same basic constant by the Church-Rosser Theorem.  $\square$

The Correspondence Theorem manifests the major theoretical improvement to the control calculus. Only a restricted version holds for the original  $\lambda_v$ -C-calculus, namely, a version for the reduction-based subcalculus. For proving operational equivalences in the old calculus, we needed the additional notion of *safe* (computational) theorems. Safe theorems are characterized by derivations that are independent of their concrete *evaluation* context, and this independence guarantees the operational equivalence on the machine.

In the new  $\lambda_v$ -C-calculus the set of *safe equations* plays a more traditional role. It is a consistent extension of the calculus and thus relates to the  $\lambda_v$ -C-calculus like a  $\lambda$ -theory to the  $\lambda$ -calculus. The definition

of a safe equation directly carries over from the old calculus:

**Definition. (Safe equations)** An equation  $M = N$  is safe if the calculus proves all theorems of the form

$$\lambda_v\text{-C} \vdash (\#C[M]) = (\#C[N])$$

where  $C[\ ]$  is an arbitrary evaluation context.

The adequacy of safe equations is encapsulated in the last theorem of this section:

**Theorem (Safe-ness).** *If  $M = N$  is safe,  $M \simeq_C N$ .*

**Proof.** The proof is an adaptation of the corresponding proof for the old calculus [2].  $\square$

It follows from this theorem that the  $\lambda_v$ -C-calculus can be consistently extended with safe equations. That is, safe equations can be added to the set of axioms of the  $\lambda_v$ -C-calculus. When we rely on such additional axioms, we write

$$\lambda_v\text{-C}^{safe} \vdash M = N,$$

indicating that the enriched calculus proves the theorem on the right-hand side.

Thus far, we have seen that prompt-applications simplify the control calculus and establish the desired relationship to the operational semantics. We henceforth refer to the modified calculus as *the  $\lambda_v$ -C-calculus*. The next challenge is to demonstrate that prompt-applications can easily be implemented and that they constitute a useful programming concept.

### 3 An Abstract Machine Semantics

The operational semantics of  $\Lambda_{\mathcal{F}}$  is based on the CEK-machine. The CEK-machine is a state transition system. It resembles Landin's SECD-machine [9], but is also close in spirit to denotational semantics. An extension to  $\Lambda_{\mathcal{F}\#}$  is straightforward.

The CEK-states are triples of control strings, environments, and control structures. A control string is either a  $\Lambda_{\mathcal{F}\#}$ -term or the unique symbol  $\dagger$ . The environment component is a finite map that assigns values to the free variables of the control string. Finally, the control structure is a stack-based encoding of the rest of the computation.

The initial configuration for the evaluation of a program  $M$  is  $\langle M, \emptyset, (\text{stop}) \rangle$ . A machine yields the value  $U$  if it stops in the terminal state  $\langle \dagger, \emptyset, ((\text{stop}) \text{ret } (V, \rho)) \rangle$  and  $U$  results from  $\langle V, \rho \rangle$  by

recursively replacing all free variables in  $V$  by their environment value in  $\rho$ .

The transition function for the  $\lambda$ -calculus-subset is straightforward. The first three transition rules *return* syntactic values as semantic values to the continuation:

$$\langle a, \rho, \kappa \rangle \xrightarrow{CEK} \langle \dagger, \emptyset, (\kappa \text{ ret } a) \rangle \quad (1)$$

$$\langle x, \rho, \kappa \rangle \xrightarrow{CEK} \langle \dagger, \emptyset, (\kappa \text{ ret } \rho(x)) \rangle \quad (2)$$

$$\langle \lambda x.M, \rho, \kappa \rangle \xrightarrow{CEK} \langle \dagger, \emptyset, (\kappa \text{ ret } \langle \lambda x.M, \rho \rangle) \rangle \quad (3)$$

A constant is returned without any further ado, a variable's value is looked up in the current environment, and a  $\lambda$ -abstraction defines a closure, i.e., a pairing of the abstraction with the current environment.

When the CEK-transition function encounters a combination, it evaluates the two components from left to right. To this end, the continuation is extended with the argument part and the current environment:

$$\langle MN, \rho, \kappa \rangle \xrightarrow{CEK} \langle M, \rho, (\kappa \text{ arg } N \rho) \rangle. \quad (4)$$

The evaluation then continues with the function part. Once the function part is reduced to a semantic value, the machine resumes the evaluation of the argument by exchanging the control-string and environment registers with the top of the control stack:

$$\langle \dagger, \emptyset, ((\kappa \text{ arg } N \rho) \text{ ret } F) \rangle \xrightarrow{CEK} \langle N, \rho, (\kappa \text{ fun } F) \rangle. \quad (5)$$

The last two rules for the  $\lambda$ -calculus-subset explain the effect of an application. If both parts are constants, the machine continues according to the  $\delta$ -function:

$$\langle \dagger, \emptyset, ((\kappa \text{ fun } f) \text{ ret } a) \rangle \xrightarrow{CEK} \langle \dagger, \emptyset, (\kappa \text{ ret } \delta(f, a)) \rangle. \quad (6)$$

Otherwise, if the function value is a closure, the evaluation continues with an evaluation of the abstraction body in an extended closure environment that maps the parameter to the argument value:

$$\langle \dagger, \emptyset, ((\kappa \text{ fun } \langle \lambda x.M, \rho \rangle) \text{ ret } V) \rangle \xrightarrow{CEK} \langle M, \rho[x := V], \kappa \rangle. \quad (7)$$

Upon encountering a  $\#$ -application, the CEK-machine must evaluate the  $\#$ -argument and, furthermore, it must guarantee that the current control stack is intact upon completion of the sub-evaluation. This is achieved by marking the control stack:

$$\langle \#M, \rho, \kappa \rangle \xrightarrow{CEK} \langle M, \rho, (\kappa \text{ mark}) \rangle. \quad (8)$$

All functions that manipulate the stack must respect this mark. The mark is removed after the sub-evaluation terminates:

$$\langle \dagger, \emptyset, ((\kappa \text{ mark}) \text{ ret } V) \rangle \xrightarrow{CEK} \langle \dagger, \emptyset, (\kappa \text{ ret } V) \rangle. \quad (9)$$

As described in Section 2, an  $\mathcal{F}$ -application transforms the control state into a value that corresponds to a functional abstraction. This transformation must account for stack markers since they determine the accessible part of the control stack. Our way to accomplish this is to modify the stack-copy ( $\oplus$ ) and the stack-erase ( $\ominus$ ) function such that they stop at a marker:

$$\begin{aligned} \oplus(\text{stop}) & \text{ is undefined} \\ \oplus(\kappa \text{ arg } N \rho) & = (\oplus \kappa \text{ arg } N \rho) \\ \oplus(\kappa \text{ fun } F) & = (\oplus \kappa \text{ fun } F) \\ \oplus(\kappa \text{ mark}) & = (\text{stop}) \end{aligned}$$

and

$$\begin{aligned} \ominus(\text{stop}) & \text{ is undefined} \\ \ominus(\kappa \text{ arg } N \rho) & = \ominus \kappa \\ \ominus(\kappa \text{ fun } F) & = \ominus \kappa \\ \ominus(\kappa \text{ mark}) & = (\kappa \text{ mark}). \end{aligned}$$

Given this, we can define the transition rule for  $\mathcal{F}$ -applications by

$$\langle \mathcal{F}M, \rho, \kappa \rangle \xrightarrow{CEK} \langle M\gamma, \rho[\gamma := \langle \mathbf{p}, \oplus \kappa \rangle], \ominus \kappa \rangle, \quad (10)$$

where  $\gamma$  is a fresh variable. In other words, the  $\mathcal{F}$ -argument is applied to a new kind of value—a  $\mathbf{p}$ -closure—that represents the currently accessible control state. In order to make this definition work, we need a transition for specifying the result of applying a  $\mathbf{p}$ -closure to a value. A function-like behavior for such applications is achieved by appending the  $\mathbf{p}$ -closure to the current continuation:

$$\langle \dagger, \emptyset, ((\kappa \text{ fun } \langle \mathbf{p}, \kappa_0 \rangle) \text{ ret } V) \rangle \xrightarrow{CEK} \langle \dagger, \emptyset, (\kappa \otimes \kappa_0 \text{ ret } V) \rangle, \quad (11)$$

where the stack-append operation ( $\otimes$ ) is defined by

$$\begin{aligned} \kappa \otimes (\text{stop}) & = \kappa \\ \kappa \otimes (\kappa' \text{ arg } N \rho) & = (\kappa \otimes \kappa' \text{ arg } N \rho) \\ \kappa \otimes (\kappa' \text{ fun } V) & = (\kappa \otimes \kappa' \text{ fun } V). \end{aligned}$$

For a comparison of this machine semantics with the standard reduction semantics, we abstract from the transition function with a CEK-*evaluation function*:

**Definition.** (*The  $eval_{CEK}$ -function*) The (partial) evaluation function  $eval_{CEK}$  maps programs to values according to the transition function:

$$eval_{CEK}(M) = \text{Unload}(V)$$

if  $\langle M, \emptyset, (\text{stop}) \rangle \xrightarrow{CEK^+} \langle \dagger, \emptyset, ((\text{stop}) \text{ret } V) \rangle$ .  
 Unload is specified by:

$$\text{Unload}(\langle V, \rho \rangle) \equiv V[x_1 := \text{Unload}(\rho(x_1))] \dots$$

for the free variables  $x_1, \dots$  in  $V$ .

The correctness of the machine is captured in:

**Theorem (Correspondence II).**

For all programs  $M \in \Lambda_{\mathcal{F}\#}$  and basic values  $a \in \text{Const}$

$$\text{eval}_C(M) = a \text{ iff } \text{eval}_{CEK}(M) = a.$$

**Proof.** The proof follows the original one: each machine component is incorporated into the control string component by a natural transformation. The environment is eliminated in favor of substitutions, evaluation contexts replace the control structures. The correctness of each transformation can be shown by a comparison of the evaluation processes.  $\square$

The CEK-machine can easily be translated into an efficient implementation for ordinary machines. The prompt-application poses no problem at all. It simply marks the control stack. When a value is returned and the top of the control stack is marked, this mark is eliminated.

The modifications to the stack-erasing and stack-moving operations are more difficult to translate. Both must now recognize and account for the mark as an artificial end of the stack. For a fast realization a search for this mark should be avoided. A possible solution is to keep the addresses of the marks in a separate stack register. Then stack-erasing is equivalent to moving a pointer from the top of this mark-stack to a register, and the stack-move operation can be realized as a single move instruction on many machines.

## 4 Programming with Prompt-Applications

As we have seen in the preceding two sections, prompt-applications totally constrain the extent of control actions. Hence, they are applicable when a program must restrict the behavior of sub-computations. Such situations frequently occur in large systems and interactive language implementations. Beyond this, the new construct is helpful in situations where a part of the continuation must be eliminated or saved for later use. In the following three subsections we illustrate these theoretical arguments with examples.

For the sub-sections, we assume that our toy language is embedded in a language like Scheme [12] (or

any other Algol-like language that supports the required constructs). This means in particular that the set of primitives includes such arithmetic and list-processing functions as `cons`, `car`, `+`, `*`, `...`, and that we use such syntactic forms as `if`, `begin`, and `iterate` with their usual semantics.<sup>3</sup>

### 4.1 Constraining Control

In large systems it is often crucial that a service routine has a single exit point. If the underlying implementation language provides for unrestricted transfer of control, this is difficult to guarantee. Consider the specific case of a parameterized file-access handler that is responsible for closing files after a program-specified read- or write-action has taken place. Assuming the existence of files and related operations like `open`, `close`, etc., the code for such a procedure is approximately

$$\lambda p f_{open}.(\text{begin } (p f_{open}) (\text{close } f_{open})),$$

where  $p$  is the program-specified procedure for reading and/or writing from file  $f$ . Unfortunately, this code is insecure in a language with aborts, jumps, and other control operations. If  $p$  transfers control to the outside, the file  $f$  is left open and another call to the file-access handler will cause an error.

Without a `#`-facility, the correct behavior is difficult to enforce. It requires a rather complicated unwind-protect structure for keeping track of domains that a program cannot leave without further action [5, 6]. In the extended language, the procedure call  $(p f)$  is simply embedded in a `#`-application:

$$\lambda p f_{open}.(\text{begin } (\#(p f_{open})) (\text{close } f_{open})).$$

This guarantees that any imperative transfer of control by  $p$  can only erase the part of the continuation within the `#`-application: control is bound to return to the prompt. Thus, the file is properly closed upon termination of the program-specified code  $p$  and the file-access routine cannot be corrupted.

Another example that belongs in the same category is the implementation of an interactive interpreter or compiler. In such language environments, the object language relies on operations in the implementation language. With prompt-applications, this can also work for control operations.

Take, for example, the operation `error`. If `error` were imported into an object language naively, an invocation of `error` in an object-program would terminate an

<sup>3</sup>All of these syntactic forms can be construed as abbreviations of  $\Lambda$ -expressions: see Appendix.

evaluation loop like

```

LOOP ≡ (iterate L (exp (prompt-read ' →))
        (if (eq? exp 'exit) 'good-bye
            (begin
              (evaluate exp base-environment)
              (L (prompt-read ' →)))))).

```

This is, of course, undesirable, and therefore, control-operations must be treated differently from ordinary primitive operations.

The solution is again based on the introduction of a prompt-application. The vulnerable application is (evaluate *exp* base-environment) and it must be protected from control-operations in *exp*:

```

LOOP ≡ (iterate L (exp (prompt-read ' →))
        (if (eq? exp 'exit) 'good-bye
            (begin
              (# (evaluate exp base-environment))
              (L (prompt-read ' →)))))).

```

In this program, the prompt-application guarantees that no matter what the object-program does, the evaluation-loop will safely continue, e.g., the invocation of *error* would abort the rest of the computation for *exp* but would also return to the evaluation-loop as desired. This is equally true for the import of such operations as *halt*, *J* [8], *call/cc* [12], or  $\mathcal{F}$ .

## 4.2 Recursive Programming with Prompt-Applications

Besides providing a means for constraining control, prompt-applications also enhance the expressive power of recursive programming languages. A typical example of a recursive task is a tree walk. Consider a multi-ary B\*-tree, which is either a (n information) leaf or a list of multi-ary B\*-trees. A pre-order traversal of such a tree can be specified by: if the tree is a leaf, enumerate it; otherwise, apply the algorithm to all elements in the list of subtrees from left to right.

Abstracting from the particular enumeration procedure at a leaf, the algorithm can be formalized as:

$$\text{Enumerate} \stackrel{df}{=} \lambda e. \lambda t. (\text{iterate } L (t t) \text{ (if (leaf? } t) (e t) \text{ (for-each } L t)))).$$

The *iterate*-loop almost literally implements the informal description. If the tree is a leaf, it applies the enumeration procedure *e* to the leaf; otherwise, the recursive procedure is applied to every list element from left to right. This is accomplished with the list-processing function *for-each*, which applies a function *f* (for effect only) to all elements of a list *l*.

With a generic schema like the above, it is straightforward to realize a variety of different tree walks. For example, a pre-order print function is realized by

$$\text{PrintPreOrder} \stackrel{df}{=} (\text{Enumerate writeln}).$$

Similarly, the routine can be used for generating an updating function that alters information in (selected) leaves:

$$\text{Update} \stackrel{df}{=} (\text{Enumerate update}).$$

A more interesting and challenging example is a tree walk that returns a leaf at a time *and* a function for enumerating the rest of the tree when appropriate. Such a procedure is useful in situations where the elements of a tree are successively fed into a different computation and/or the information in the rest of the tree may not be needed. The enumeration step now immediately returns a pair of results: the leaf and the function. On one hand, this additional function specifies what the application of the enumeration function would do to the same tree without the current leaf; on the other hand, it is precisely what the current invocation of the pre-order traversal procedure would have to do in order to complete its computation. Put differently, the desired second component of the enumeration step is the part of the continuation that describes the extent of the rest of the tree walk.

This new enumeration procedure is yet another instantiation of the above schema in a language with prompt- and  $\mathcal{F}$ -applications. If the procedure call is embedded in a prompt-application, the second component of the stream can be constructed with an  $\mathcal{F}$ -application. The first step is thus to embed the application of *Enumerate* in a prompt-application:

$$\text{Enumerate2} \stackrel{df}{=} \lambda t. (\# (\text{Enumerate } e t)).$$

The second step is to design an enumeration procedure *e* that pairs the current leaf with the current continuation of the tree walk. This procedure must approximately look like

$$e \stackrel{df}{=} \lambda l. \mathcal{F}(\lambda r. (\text{cons } l \dots r \dots)).$$

The remaining question is what to return as the second component of the pair. The continuation *r* is

a function that will resume the tree walk when invoked on an arbitrary argument. In order to preserve the invariant that this tree walk is always run in a prompt-application, we must encapsulate  $r$  in an abstraction:

$$\lambda x.(\#(r x)).$$

Finally, since the argument to  $r$  is actually irrelevant, we can put the prompt-application in a null-ary function and use NIL as the dummy value:

$$e \stackrel{df}{=} \lambda l.\mathcal{F}(\lambda r.(\text{cons } l \lambda().(\#(r \text{NIL}))))).$$

In the absence of prompt-applications, the formalization of this tree walk becomes more complicated. It requires an implementation schema where `Enumerate` acts like a coroutine with respect to the rest of the program, i.e., it stores away the continuation of the main program upon every resumption for use in  $e$ . The important difference is that the prompt-free solution contains recurring programming patterns. Such patterns clearly call for an appropriate abstraction that hides the details of grabbing partial continuations. The prompt-application is the correct fundamental abstraction for reasoning and programming with parts of the current continuation.

**Remark.** `Enumerate2` is a typical example of a function that makes practical use of first-class continuations in the absence of assignments. Indeed, it is also an example of the usefulness of continuations for stream-programming [9]. Given a lazy `cons`, i.e., an abbreviation à la

$$(\text{cons}\$ a d) \stackrel{df}{=} (\text{cons } a (\lambda().d)),$$

the function `Enumerate2` can be recast as

$$\begin{aligned} \text{Enumerate2} \stackrel{df}{=} & \lambda t.(\text{iterate } L (t t) \\ & (\text{if } (\text{leaf? } t) \\ & (\mathcal{F}(\lambda r.(\text{cons}\$ t (\#(r \text{NIL})))))) \\ & (\text{for-each } L t)). \end{aligned}$$

In other words, the function `Enumerate2` transforms a multi-ary B\*-tree into a list whose rest is computed when necessary. With a lazy programming language, the same algorithm could be expressed as:

$$\begin{aligned} \text{Enumerate2} \stackrel{df}{=} & \lambda t.(\text{iterate } L (t t) \\ & (\text{if } (\text{leaf? } t) t \\ & (\text{apply append}\$ (\text{map}\$ L t)))). \end{aligned}$$

However, this function is not an instantiation of the tree-walk programming schema. Although the algorithm is intuitively a variant of the single-step enumeration, lazy programming forces a rewriting of the `Enumerate`-function to fit this particular need. Also, depending on the management of continuations by the implementation, the  $\mathcal{F}$ - $\#$ -based version avoids the overhead of the lazy version: because `for-each` is used for effect only, it does not require the expensive allocation of extraneous `cons`-cells. **End**

### 4.3 Making Function-Exits More Efficient

Programming languages with operators like `call/cc` and  $\mathcal{F}$  can easily simulate the effect of function exits. A multiplication function  $\Pi$  that computes the product of a list of numbers can thus escape upon encountering 0:

$$\begin{aligned} \Pi_1 \stackrel{df}{=} & (\lambda l.(\mathcal{F} \lambda \epsilon.\epsilon(\text{iterate } p (l l) \\ & (\text{if } (\text{null? } l) 1 \\ & (\text{if } (\text{zero?}(\text{car } l)) (\mathcal{F}(\lambda d.(\epsilon 0))) \\ & (*(\text{car } l)(p(\text{cdr } l))))))))), \end{aligned}$$

or, with `call/cc` and proper Scheme-continuations,

$$\begin{aligned} \Pi_1 \stackrel{df}{=} & (\lambda l.(\text{call/cc } \lambda k.(\text{iterate } p (l l) \\ & (\text{if } (\text{null? } l) 1 \\ & (\text{if } (\text{zero?}(\text{car } l)) (k 0) \\ & (*(\text{car } l)(p(\text{cdr } l))))))). \end{aligned}$$

Unfortunately, this simulation of function-exits manipulates the control structure extensively. It first accesses the continuation of  $\Pi$ , a second time when it eliminates the recursive unfoldings of  $\Pi$  to return 0, and, because there is no indication as to the extent of the manipulation, the escape-jump requires the removal and re-installation of most of the current control stack. Depending on the implementation of continuations and continuation access operations, this can become expensive.

With a  $\#$ -application, we can instead mark the entrance point with a prompt, and when an escape-exit becomes necessary, the partial continuation is erased:

$$\begin{aligned} \Pi_2 \stackrel{df}{=} & (\lambda l.(\# (\text{iterate } p (l l) \\ & (\text{if } (\text{null? } l) 1 \\ & (\text{if } (\text{zero?}(\text{car } l)) (\mathcal{F}(\lambda d.0)) \\ & (*(\text{car } l)(p(\text{cdr } l))))))). \end{aligned}$$

This new schema for the implementation of function-exits is more efficient. As discussed in the preceding section, the `#`-application only marks the control stack and does not require any further action. When the function must escape, the stack is only erased to the next marker. Hence, this second version of function-exits manipulates the control structure only when necessary and only to the extent necessary. It is practically equivalent to `catch` and `throw` in Common Lisp [14] or `exit`-statements in other Algol-like languages.

**Remark.** The correctness of this transformation can be verified with a simple derivation in the calculus. The functions  $\Pi_1$  and  $\Pi_2$  are extensionally equivalent in the extended  $\lambda_v$ -C-calculus:

**Proposition.** *Let  $L$  be a list of numbers. If  $\Pi_1$  is correct, then  $\lambda_v$ -C<sup>safe</sup>  $\vdash (\Pi_1 L) = (\Pi_2 L)$ .*

**Proof.** The correctness of  $\Pi_1$  implies that  $(\Pi_1 L) = n$  is a safe equation for any list  $L$  and some number  $n$ . It follows that

$$\begin{aligned} \lambda_v\text{-C}^{safe} \vdash n &= (\text{iterate } p \ (l \ L) \\ &\quad (\text{if } (\text{null? } l) \ 1 \\ &\quad\quad (\text{if } (\text{zero?}(\text{car } l)) \ (\mathcal{F}(\lambda d.0)) \\ &\quad\quad\quad (*(\text{car } l)(p(\text{cdr } l)))))). \end{aligned}$$

The rest of the proof is a simple manipulation of  $(\Pi_2 L)$ :

$$\begin{aligned} &\lambda_v\text{-C}^{safe} \vdash (\Pi_2 L) \\ =_c &\ (\# \ (\text{iterate } p \ (l \ L) \\ &\quad (\text{if } (\text{null? } l) \ 1 \\ &\quad\quad (\text{if } (\text{zero?}(\text{car } l)) \ (\mathcal{F}(\lambda d.0)) \\ &\quad\quad\quad (*(\text{car } l)(p(\text{cdr } l))))))) \\ = &\ (\# \ ((\lambda x.x)(\text{iterate } p \ (l \ L) \quad \text{by } safe\text{-ness} \\ &\quad (\text{if } (\text{null? } l) \ 1 \\ &\quad\quad (\text{if } (\text{zero?}(\text{car } l)) \ (\mathcal{F}(\lambda d.((\lambda x.x)0)) \\ &\quad\quad\quad (*(\text{car } l)(p(\text{cdr } l)))))))))) \\ =_c &\ (\# \ (\lambda \epsilon.\epsilon(\text{iterate } p \ (l \ L) \\ &\quad (\text{if } (\text{null? } l) \ 1 \\ &\quad\quad (\text{if } (\text{zero?}(\text{car } l)) \ (\mathcal{F}(\lambda d.(\epsilon 0)) \\ &\quad\quad\quad (*(\text{car } l)(p(\text{cdr } l))))))))))(\lambda x.x)) \\ =_c &\ (\# \ \mathcal{F}(\lambda \epsilon.\epsilon(\text{iterate } p \ (l \ L) \\ &\quad (\text{if } (\text{null? } l) \ 1 \\ &\quad\quad (\text{if } (\text{zero?}(\text{car } l)) \ (\mathcal{F}(\lambda d.(\epsilon 0)) \\ &\quad\quad\quad (*(\text{car } l)(p(\text{cdr } l)))))))))) \\ =_c &\ (\# \ (\Pi_1 L)) \\ =_c &\ (\# \ n) \quad \text{for some number } n, \text{ by assumption} \end{aligned}$$

$$=_c \quad n = (\Pi_1 L) \quad \text{by assumption, } safe\text{-ness. } \square$$

End

## 5 Related Practical Work

Prompt-applications are new for the theoretical investigations of programming language control facilities, but, to some extent, they are known in the realm of practical programming. In principle, the prompt-application is a generalization of the Lisp-facility `errorset` [10]. The function `errorset` evaluates an application until it successfully terminates or an error occurs. In the first case, `errorset` resumes its continuation with a list of the result, in the second case, it returns `NIL`. In the same manner, the prompt-application generalizes the `catch`-construct in Common Lisp [14]. Whereas a `catch` only intercepts one kind of abortive action, a prompt-application catches all. Clearly, a prompt-application can simulate a `catch` as a syntactic abbreviation with a simple message-passing protocol. The inverse is impossible.

The novelty of our approach with respect to practical work is to explore the presence of prompt-applications in languages with *full* power over evaluation control. In general, such languages do not provide the means for totally constraining control. Considering the examples in the preceding section, we believe that this is a design omission.

An exception to the above rule is the programming language GL [7]. It contains both powerful imperative control operations and a construct with the capabilities of a prompt-application. It has an operation `CurCont` for accessing the current continuation and two operations for invoking continuations: `continue` and `fork`. The operation `continue` replaces the current control stack with the one that is encoded by the invoked continuation, i.e., it realizes the invoking of continuations as in Iswim or Scheme. The operation `fork` evaluates a continuation like a sub-expression, except that this sub-expression can *not* manipulate the control stack below the point of invocation. In other words, a `fork`-statement uses an *implicit* prompt-application for invoking a continuation.

Due to the implicit occurrence, it is difficult in GL to isolate the prompt-facility as a separate programming abstraction. In  $\lambda_v$ -C, on the other hand, the programmer can exploit and reason about the concept of restricted control independently. As demonstrated above with the tree-processing example, this may lead to novel applications of prompt- and control-facilities.

## 6 Conclusion

In the preceding sections, we discussed a facility for constraining the extent of control operations. Its introduction was motivated by theoretical difficulties in the theory of control. The new facility solved the theoretical problem, and, even more important, it turned out to be a practical programming tool. The development of the prompt-application again illustrated the practical relevance of the analysis of programming language calculi. Besides the present example with the non-theorem as its starting point, this analysis already led to the concept of  $\mathcal{F}$ -applications through a simplification of the reductions and computations for call/cc [3]. A further analysis of calculi for imperative operations will certainly yield more systematic insight into the structure and design of programming languages.

**Acknowledgement.** Dan Friedman and Carolyn Talcott read early drafts of the paper and suggested several improvements. Perry Wagle's quest for a non-lazy enumeration of trees inspired `Enumerate2`. Mitch Wand pointed out the kinship of `errorset` and `#`.

## 7 References

1. BARENDREGT, H.P. *The Lambda Calculus: Its Syntax and Semantics*. rev. ed. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.
2. FELLEISEN, M. *The Calculi of Lambda- $\nu$ -CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Ph.D. dissertation, Indiana University, 1987.
3. FELLEISEN, M., D.P. FRIEDMAN, B. DUBA, AND J. MERRILL. Beyond continuations. Technical Report No 216, Indiana University Computer Science Department, 1987.
4. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, AND B. DUBA. A syntactic theory of sequential control, *Theor. Comput. Sci.*, 1987, to appear.
5. HAYNES, C. AND D.P. FRIEDMAN. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.* 9(4), 1987.
6. HANSON, C., J. LAMPING. Dynamic binding in Scheme, unpublished manuscript, 1984, MIT.
7. JOHNSON, G.F. GL—A language and environment for interactively experimenting with denotational definitions. In *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques SIGPLAN Notices* 22(7), 1987, 165–176.

8. LANDIN, P.J. The next 700 programming languages. *Commun. ACM* 9(3), 1966, 157–166.
9. LANDIN, P.J. The mechanical evaluation of expressions. *Comput. J.* 6(4), 1964, 308–320.
10. MCCARTHY, J. et al. *Lisp 1.5 Programmer's Manual*. sec. ed. MIT Press, Cambridge, 1965.
11. PLOTKIN, G.D. Call-by-name, call-by-value, and the  $\lambda$ -calculus. *Theor. Comput. Sci.* 1, 1975, 125–159.
12. REES J. AND W. CLINGER (Eds.). The revised<sup>3</sup> report on the algorithmic language Scheme. *SIGPLAN Notices* 21(12), 1986, 37–79.
13. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972, 717–740.
14. STEELE, G. *Common Lisp—The Language*. Digital Press, 1984.

## 8 Appendix

The syntactic facilities of Section 4 are abbreviations of  $\Lambda$ -expressions:

— (**if**  $\langle tst \rangle \langle thn \rangle \langle els \rangle$ )  $\equiv \langle tst \rangle (\lambda d. \langle thn \rangle) (\lambda d. \langle els \rangle) 0$   
 where **True**  $\equiv \lambda xy. x$  and **False**  $\equiv \lambda xy. y$ ;  
 (the  $\delta$ -function must respect this representation,  
 e.g.,  $\delta(\text{zero?}, 0) = \text{True}$ )

— (**begin**  $\langle exp \rangle_1 \dots \langle exp \rangle_i$ )  $\equiv$   
 $(\lambda x_1 \dots x_i. x_i) \langle exp \rangle_1 \dots \langle exp \rangle_i$

— (**iterate**  $p (x X) B$ )  $\equiv Y_\nu (\lambda px. B) X$  where

$$Y_\nu \equiv \lambda fx. (\lambda g. f(\lambda x. ggx)) (\lambda g. f(\lambda x. ggx)) x.$$

The  $Y_\nu$ -combinator is a call-by-value recursion operator and satisfies the following equation:

$$\lambda_\nu\text{-C} \vdash Y_\nu F x = F(Y_\nu F) x.$$