Reasoning with Continuations III:
A Complete Calculus of Control

Amr Sabry Matthias Felleisen

Department of Computer Science
Rice University
Houston, TX 77251-1892

Summary

The λ-calculus is the simple and elegant, yet powerful basis for the semantic manipulation of functional programs by programmers and programming tools (compilers, optimizers, semantics-based editors) alike. The theory provides a simple system for evaluating programs in a symbolic manner; the typed version is a *complete* system of program equality for the pure core language. Felleisen *et al.* [6, 9, 8] and Talcott [24, 25] have recently developed extensions of the λ-calculus for reasoning about languages with powerful control operators, the λ_v-C-calculus and the first-order logic IOCC, respectively, but they did not address the issue of completeness.

The central result of this paper is an extension of the λ_v-C-calculus for a complete set of control operators and a correspondence theorem between the new theory and the $\lambda\beta\eta$-calculus. Technically, the theorem shows that the two calculi prove the same equations with respect to the well-known continuation-passing style translation (and its inverse), which is the standard tool for defining control operators via translation into a functional language. As a corollary, the calculus proves *all* program equivalences between terms over the pure language extended *with* control operators. We believe that this work has important consequences for the direct-style compilation of programming languages with control operators.

1 The Expressive Power of CPS

The continuation-passing style (CPS) transformation constitutes an essential component of many program analysis tools. Apparently, compilers benefit from the CPS intermediate representation [1, 21, 23], partial evaluators perform better on CPS programs [3], and abstract interpreters can extract more information from CPS programs [14, 15]. However, the CPS formalism has its disadvantage. It is a *global* transformation that re-structures the entire program, increases its size considerably, and makes it mostly unreadable for the programmer.

Our goal is to capture the expressive power of the CPS formalism in a variant of the λ-calculus. More precisely, we aim at developing a theory for reasoning about control abstractions that is complete and canonical, as the λ-calculus is complete and canonical for reasoning about functional abstractions. We consider two aspects of the expressive power of CPS: the control operators that are definable within the CPS framework, and the equational axioms that govern the behavior of these operators.

Sitaram and Felleisen [22] address the first aspect by analyzing CPS models. They establish that *abort* ($\mathcal{A}$), *call-with-current-continuation* (**callcc**), and control delimiters represent a complete set of control operators. Pragmatically, these control operators can concisely express a wide variety of control abstractions such as error exits, jumps, backtracking, coroutines, and exception handling. However, there are no known equational theories that characterize the behavior of the operators $\mathcal{A}$ and **callcc** in the presence of control delimiters *and simultaneously capture the equations provable in the CPS framework.*

The main technical result of this paper is the development of a theory for reasoning about control abstractions that proves all the equations in the CPS framework and also proves all program equivalences between recursion-free terms. The theory is an extension of the λ_v-C-calculus [6, 9, 8] that completely characterizes the operational behavior of **callcc** and $\mathcal{A}$ in the presence of control delimiters. The theory permits the manipulation of programs in a simple algebraic fashion for the purposes of symbolic evaluation or for proving the correctness of optimizing transformations.

```
(define make-coroutine
  (lambda (f)
    (call/cc
      (lambda (maker)
        (let ([LCS 'any])
          (let ([resume
                  (lambda (dest val)
                    (call/cc (lambda (k)
                               (set! LCS k)
                               (dest val))))])
            (f resume (resume maker
                          (lambda (v) (LCS v))))
            (error "coroutine fell off end")))))))
```

```
(define make-coroutine
  (lambda (f)
    (letrec ([LCS (lambda (x)
                    (f (lambda (dest val)
                         (call/cc (lambda (k)
                                    (set! LCS k)
                                    (dest val))))
                       x)
                    (error "coroutine fell off end"))])
      (lambda (v) (LCS v)))))
```

Figure 1: Coroutines from continuations.

To illustrate the power of our new calculus, we sketch an equivalence proof between the two Scheme programs in Figure 1. The programs implement a coroutine facility using first-class continuations. The left program, due to Haynes, Friedman, and Wand [11], is the result of clear and well-understood design steps but is far more complicated than necessary. The initialization part of the coroutine is non-trivial: it involves capturing a continuation and an artificial use of the function *resume* that assigns the proper value to the local control state (LCS). The second program avoids the clumsy initialization phase and immediately returns a closure. Using our new set of axioms, we can incrementally rewrite the left program and transform it into the second program. This derivation establishes that the two programs are observationally indistinguishable. The main steps of the transformation are in the appendix.

The next section introduces basic concepts and summarizes our previous work about the connection between a pure call-by-value language and its image under the CPS transformation [20]. In the third section, we extend the source language with the control operators: **callcc** and $\mathcal{A}$, and control delimiters. For each extension, we identify a set of reductions that captures the effect of $\beta\eta$-reductions on CPS terms. In Section 4, we address the issue of semantic completeness. Finally, we conclude by discussing related work and the impact of our work on the compilation of languages like Scheme and ML.

2 The λ-calculus and the CPS Transformation

The language Λ consists of function definitions (λ-abstractions) and function applications. The set of terms M is generated inductively over an infinite set of variables, *Vars*:[1]

$$\begin{array}{rcll} M & ::= & V \mid (M\ M) & \qquad (\mathit{Terms}) \\ V & ::= & x \mid (\lambda x.M) & \qquad (\mathit{Values}) \\ x & \in & \mathit{Vars} & \end{array}$$

We adopt Barendregt's [2, ch 2,3] notation and terminology for this syntax. In particular, we use the standard definitions of free and bound variables, substitution, contexts (terms with holes) and work with the quotient of Λ under α-equivalence.

The semantics of the language is a partial function, *eval*, from programs to answers. A *program* is a term with no free variables, and an *answer* is a member of the syntactic category of values. The operational specification of the function *eval* relies on the notion of *evaluation contexts* [7]. Informally, an evaluation context is the context surrounding the *current* instruction during the evaluation of the program, i.e., the evaluation context is a syntactic representation of the continuation. For a call-by-value semantics, the definition of evaluation contexts, E, is:

$$E \ ::= \quad [\,] \mid (V\ E) \mid (E\ M)$$

An evaluation step simply executes the current instruction in the program. Again, for call-by-value, the instruction is β_v:

$$E[((\lambda x.M)\,V)] \ \longmapsto\ E[M[x := V]]$$

[1] Even though this language does not contain constants, our results carry over to full Scheme- and ML-like languages. One may interpret free variables in the following proofs as constants, primitives, or side-effecting operations, without effect on their validity. The one exception is η_v, which must be removed from the set of valid reductions in Scheme-like languages.

The language $cps(\Lambda)$:

$$\begin{array}{rcll}
P & ::= & (K\ W) & (\mathit{Answers}) \\
W & ::= & x \mid (\lambda k.K) & (\mathit{Values}) \\
K & ::= & k \mid (\lambda x.P) \mid (W\ K) & (\mathit{Continuations}) \\
x & \in & \mathit{Vars} & \\
k & \in & K\text{-}\mathit{Vars} = \{k\} \text{ and } \mathit{Vars} \cap K\text{-}\mathit{Vars} = \emptyset &
\end{array}$$

The calculus $\boldsymbol{\lambda}\beta\eta$:

$$\begin{array}{rcll}
((\lambda x.M)\ N) & \longrightarrow & M[x := N] & & (\beta) \\
(\lambda x.Mx) & \longrightarrow & M & x \notin FV(M) & (\eta)
\end{array}$$

The CPS transformation and its inverse:

$$\begin{array}{rll|rcl}
\multicolumn{3}{c}{\Lambda \Rightarrow cps(\Lambda)} & \multicolumn{3}{c}{cps(\Lambda) \Rightarrow \Lambda} \\
\mathcal{C}_k[\![V]\!] & = (k\ \Phi[\![V]\!]) & & \mathcal{C}^{-1}[\![(K\ W)]\!] & = & \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]] \\
\mathcal{C}_k[\![E[(V_1\ V_2)]]\!] & = ((\Phi[\![V_1]\!]\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V_2]\!]) & & & & \\
\Phi[\![x]\!] & = x & & \Phi^{-1}[\![x]\!] & = & x \\
\Phi[\![\lambda x.M]\!] & = \lambda c.\lambda x.\mathcal{C}_c[\![M]\!] & & \Phi^{-1}[\![\lambda k.K]\!] & = & \lambda z.\mathcal{K}^{-1}[\![K]\!][z] \\
\mathcal{K}_k[\![[\ ]]\!] & = k & & \mathcal{K}^{-1}[\![k]\!] & = & [\ ] \\
\mathcal{K}_k[\![E[(V\ [\ ])]]\!] & = (\Phi[\![V]\!]\ \mathcal{K}_k[\![E]\!]) & & \mathcal{K}^{-1}[\![(\lambda x.P)]\!] & = & ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\ ]) \\
\mathcal{K}_k[\![E[([\ ]\ M)]]\!] & = (\lambda f.\mathcal{C}_k[\![E[(f\ M)]]\!]) & & \mathcal{K}^{-1}[\![(W\ K)]\!] & = & \mathcal{K}^{-1}[\![K]\!][(\Phi^{-1}[\![W]\!]\ [\ ])]
\end{array}$$

Figure 2: The CPS language.

An evaluation applies the single steps repeatedly until it reaches an answer, if any:

$$eval(M) = V \quad \text{iff} \quad M \longmapsto^{*} V$$

The semantics of the language determines the validity of program transformations and optimizations. The replacement of an expression M by an expression N in a program is sound if it does not change the *observable* behavior of the program. Formally, two expressions M and N are (call-by-value) observationally equivalent, $M \cong_v N$, if the following condition holds:

> For all contexts C such that $C[M]$ and $C[N]$ are programs, either $eval(C[M])$ and $eval(C[N])$ are both defined or are both undefined.

The CPS transformation maps the language Λ to a proper subset of itself. The evaluation of the resulting CPS terms is indifferent to the call-by-value or call-by-name strategies [17, 19]. Therefore, both the λ-calculus (see Figure 2) and the λ_v-calculus are suitable theories for reasoning about the equivalence of CPS procedures. In general, the λ-calculus is preferred because it is a larger theory.

The CPS transformation in Figure 2 produces compact outputs that are provably equal to the outputs of the traditional Fischer/Reynolds CPS transformation [20].[2] The set of CPS terms that results from this transformation is the language $cps(\Lambda)$ (see Figure 2). This set is closed under all possible $\beta\eta$ reductions and thus includes all the relevant CPS terms. Because the source language does not include any control operators, the CPS language includes exactly one special variable k that represents the current continuation.

The function $\mathcal{U}$ is a quasi-inverse to the CPS transformation; it is not a complete inverse because $\mathcal{C}$ maps different Λ terms to the same CPS term. Using this function, it is possible to identify a set of reductions that proves an equation between call-by-value terms if and only if the λ-calculus proves the corresponding equation between the CPS transforms. The resulting set of reductions is as follows:

$$
\begin{array}{rcllr}
((\lambda x.M)\, V) & \longrightarrow & M[x := V] & & (\beta_v) \\
(\lambda x.Vx) & \longrightarrow & V & x \notin FV(V) & (\eta_v) \\
E[((\lambda x.M)\, N)] & \longrightarrow & ((\lambda x.E[M])\, N) & x \notin FV(E) & (\beta_{lift}) \\
((\lambda x.E[x])\, M) & \longrightarrow & E[M] & x \notin FV(E) & (\beta_\Omega)
\end{array}
$$

The theory generated by the above axioms is an extension of the λ_v-calculus. We refer to this extension as $\boldsymbol{\lambda}\beta_v\eta_v A$ where $A = \{\beta_{lift}, \beta_\Omega\}$. The calculi $\boldsymbol{\lambda}\beta_v\eta_v A$ and $\boldsymbol{\lambda}\beta\eta$ satisfy a correspondence theorem similar to the correspondence between the λ-calculus and Combinatory Logic [2, ch 7].

Theorem 2.1 (Correspondence [20]) *Let $M, N \in \Lambda$ and $P, Q \in cps(\Lambda)$. Then,*

1. $\boldsymbol{\lambda}\beta_v\eta_v A \vdash M = \mathcal{C}^{-1}[\![\mathcal{C}_k[\![M]\!]]\!]$.
2. $\boldsymbol{\lambda}\beta\eta \vdash P = \mathcal{C}_k[\![\mathcal{C}^{-1}[\![P]\!]]\!]$.
3. $\boldsymbol{\lambda}\beta_v\eta_v A \vdash M = N$ *iff* $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$.
4. $\boldsymbol{\lambda}\beta\eta \vdash P = Q$ *iff* $\boldsymbol{\lambda}\beta_v\eta_v A \vdash \mathcal{C}^{-1}[\![P]\!] = \mathcal{C}^{-1}[\![Q]\!]$.

3 The Control Extensions

The analysis of mathematical CPS models [22] shows that these models can essentially express continuation capture (**callcc**), error exits ($\mathcal{A}$), and control delimiters (#). The first subsection deals with the first two of these operators, which are more widely used and less problematic than control delimiters. The second subsection is an analysis of two varieties of prompts. We are able to extend the correspondence theorem in all cases.

3.1 A complete set of axioms for callcc and $\mathcal{A}$

The syntax of the source language in Figure 3 reflects the addition of **callcc** and $\mathcal{A}$ as two functional constants. The extension of the CPS transformation gives the usual semantics to **callcc** and $\mathcal{A}$. The operator **callcc** calls its argument with a procedural representation

The source language: $\Lambda + \mathbf{callcc} + \mathcal{A}$

$$\begin{array}{rcl} M & ::= & V \mid (M\ M) \\ V & ::= & x \mid (\lambda x.M) \mid \\ & & \mathbf{callcc} \mid \mathcal{A} \end{array}$$

The CPS language: $cps(\Lambda + \mathbf{callcc} + \mathcal{A})$

$$\begin{array}{rcll} P & ::= & W \mid (K\ W) & (\mathit{Answers}) \\ W & ::= & x \mid (\lambda k.K) & (\mathit{Values}) \\ K & ::= & k \mid (\lambda x.P) \mid (W\ K) & (\mathit{Continuations}) \\ \\ x & \in & \mathit{Vars} & \\ k & \in & K\text{-}\mathit{Vars} = \{k_1, k_2, \cdots\} & \\ & & \text{and } K\text{-}\mathit{Vars} \cap \mathit{Vars} = \emptyset & \end{array}$$

The extensions of the CPS transformation and its inverse:

$\Lambda + \mathbf{callcc} + \mathcal{A} \Rightarrow Cps$

$$\begin{array}{rcl} \Phi[\![\mathbf{callcc}]\!] & = & \lambda k.\lambda u.((u\ k)\ \lambda d.k) \\ \Phi[\![\mathcal{A}]\!] & = & \lambda k.\lambda x.x \end{array}$$

$Cps \Rightarrow \Lambda + \mathbf{callcc} + \mathcal{A}$

$$\begin{array}{rcl} \mathcal{C}^{-1}[\![W]\!] & = & (\mathcal{A}\ \Phi^{-1}[\![W]\!]) \\ \Phi^{-1}[\![\lambda k.K]\!] & = & \lambda z.\mathbf{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K]\!][z] \\ \mathcal{K}^{-1}[\![k]\!] & = & (k\ [\ ]) \end{array}$$

Figure 3: Extending Λ and CPS with **callcc** and $\mathcal{A}$.

of the current continuation. The CPS transform of $\mathcal{A}$ is a value in the CPS language (a continuation transformer) that ignores the current continuation and returns the initial continuation $\lambda x.x$ instead.

The set of CPS terms in Figure 3 includes all the terms that are reachable by the extended CPS transformation and their $\beta\eta$ contractions. A comparison of this set with the language in Figure 2 explains how the addition of **callcc** and $\mathcal{A}$ affects the set of reachable CPS terms and thus, how it affects the semantics. Intuitively, **callcc** permits the programmer to "label" arbitrary points in the program. Thus, more than one continuation can potentially be lexically visible at any point during the execution of the program. The CPS language accomodates this fact by providing an infinite set of continuation variables instead of a singleton. Also, the addition of $\mathcal{A}$ permits the programmer to ignore the current continuation by returning a value as the answer of the entire program. This extension is reflected in the CPS language by extending the syntactic category of answers to include values directly.

The extension of the CPS language still has a natural pre-image as in Figure 3. The inverse of an answer W is an expression that aborts with the value that corresponds to W. To map the term $\lambda k.K$ to the source language, we extend the function $\mathcal{U}$ in Figure 2 using an idea of Danvy and Lawall [5].[3] Finally, as the last line in the inverse transformation shows, all continuations are explicitly invoked.

In order to adopt Correspondence Theorem 2.1, it is necessary to identify reductions

[2]Steele [23] and Danvy/Filinski [4] have similar compactifying CPS transformations but this is not an issue in this paper. We prefer the transformation $\mathcal{C}$ because its definition fits nicely with the syntactic categories of answers, values, and continuations, thus simplifying a lot of the technical work.

[3]Danvy and Lawall perform a counting analysis to determine whether to include a **callcc**. This analysis is unnecessary for our purposes. The outputs of our inverse transformation are reducible to their outputs, thus achieving the same effect without the counting analysis.

for **callcc** and $\mathcal{A}$ that capture the effect of $\beta\eta$ on the new CPS language. This set includes the reductions of the $\boldsymbol{\lambda}\beta_v\eta_v A$ calculus *mutatis mutandis* and the additional reductions (C) of Figure 4. We arrive at this set by decomposing $\beta\eta$ on $cps(\Lambda + \textbf{callcc} + \mathcal{A})$, mapping the reductions back to the source language, and analyzing the result. [The full paper will contain the details.]

$$
\begin{array}{rcll r}
(\textbf{callcc}\ \lambda k.kM) & \longrightarrow & (\textbf{callcc}\ \lambda k.M) & & (C_{current}) \\
(\textbf{callcc}\ \lambda d.M) & \longrightarrow & M & d \notin FV(M) & (C_{elim}) \\
E[(\textbf{callcc}\ M)] & \longrightarrow & \textbf{callcc}\ \lambda k.E[(M\ (\lambda f.(k\ E[f])))] & k, f \notin FV(E, M) & (C_{lift}) \\
\textbf{callcc}\ \lambda k.C[E[(k\ M)]] & \longrightarrow & \textbf{callcc}\ \lambda k.C[(k\ M)] & & (C_{abort}) \\
(\textbf{callcc}\ \lambda k.((\lambda x.M)\ N)) & \longrightarrow & ((\lambda x.(\textbf{callcc}\ \lambda k.M))\ N) & k \notin FV(N) & (C_{tail}) \\
E[(\mathcal{A}\ M)] & \longrightarrow & (\mathcal{A}\ M) & & (Abort)
\end{array}
$$

Figure 4: The reductions C.

The **callcc** specific reductions have the following intuitive explanation. The first reduction shows that the current continuation is always implicitly applied. The reduction C_{elim} is a garbage-collection rule: continuations captured but not used can be collected. The reduction C_{lift} characterizes the capture of continuations via **callcc** while the reduction C_{abort} shows that continuations abort their context upon invocation. The last **callcc** reduction implies that the continuation of an application is indistinguishable from the continuation of the body.[4] The explanation of $(Abort)$ is obvious.

The following theorem summarizes the results of this section.

Theorem 3.1 (Correspondence) *Let* $M, N \in \Lambda + \textbf{callcc} + \mathcal{A}$ *and* $P, Q \in cps(\Lambda + \textbf{callcc} + \mathcal{A})$. *Then,*

1. $\boldsymbol{\lambda}\beta_v\eta_v AC \vdash \mathcal{C}^{-1}[\![\mathcal{C}_k[\![M]\!]]\!] = (k\ M)$.
2. *Let* $k_1, k_2, \cdots, k_n$ *be the free continuation variables in* P, *then*

$$\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![\mathcal{C}^{-1}[\![P]\!]]\!][k_1 := (\lambda d.k_1), k_2 := (\lambda d.k_2), \cdots, k_n := (\lambda d.k_n)] = P.$$

3. $\boldsymbol{\lambda}\beta_v\eta_v AC \vdash M = N$ *iff* $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$.
4. $\boldsymbol{\lambda}\beta\eta \vdash P = Q$ *iff* $\boldsymbol{\lambda}\beta_v\eta_v AC \vdash \mathcal{C}^{-1}[\![P]\!] = \mathcal{C}^{-1}[\![Q]\!]$.

For entire programs, the first part of the theorem implies that $\mathcal{C}^{-1}[\![\mathcal{C}_k[\![M]\!]]\!][k := \lambda x.x]$ is equal to M. Moreover, since programs are closed, the substitution in the second part is unnecessary.

[4] ... and a good implementation would use the same continuation: we can also interpret this law as imposing tail-call optimization on faithful implementations.

3.2 A complete set of axioms for

The last control facility that fully abstract denotational CPS models can express is the ability to delimit control actions. We extend the source language with such a facility, the prompt expression [6]: $(\# \ M)$.

Intuitively, the prompt treats its subexpression as a complete program by evaluating it in the initial continuation, forwarding the result to the current continuation. To specify this behavior formally, we extend the CPS transformation accordingly:

$$\mathcal{C}_k[\![(\# \ M)]\!] = (k \ \mathcal{C}_c[\![M]\!])[c := \lambda x.x]$$

Unfortunately, this extension produces CPS terms that are not tail-recursive, and hence CPS programs are no longer indifferent to the order of evaluation. For example, the expression:

$$((\lambda x.y) \ (\# \ \Omega)), \qquad \text{where } \Omega = ((\lambda x.xx) \ (\lambda x.xx))$$

translates as follows:

$$((\lambda x.ky) \ ((\lambda x.((x \ \lambda x.x) \ x)) \ (\lambda k.\lambda x.((x \ k) \ x)))).$$

This expression would reduce to $(k \ y)$ in a call-by-name evaluation but would diverge in a call-by-value evaluation. In the source language, this choice corresponds to whether the expression $(\# \ \Omega)$ is a value or not. We refer to the prompt in the first case as the *lazy prompt* because its subexpression may not be evaluated, and refer to the other prompt as the *strict prompt*.

The classic semantics for control delimiters [4, 6] corresponds to the strict prompt. However, meta-theorems are, as always, easier to formulate and prove for the by-name version. We therefore address the lazy prompt first and defer the strict prompt until the second subsection.

3.2.1 Lazy prompt

For the lazy prompt, the CPS language has a call-by-name semantics and therefore the reductions for the CPS language are the canonical reductions β and η. The extensions to the source language and the CPS transformation are as in Figure 5.

Extended languages:

$\Lambda_c (= \Lambda + \mathbf{callcc} + \mathcal{A} + \#)$

$$V \ ::= \ \cdots \mid (\# \ M)$$

The CPS language

$$\begin{array}{rcll} P, W & ::= & x \mid (\lambda k.K) \mid (K \ W) & (\mathit{Answers, Values}) \\ K & ::= & k \mid (\lambda x.P) \mid (W \ K) & (\mathit{Continuations}) \end{array}$$

Extended transformations:

$\Lambda_c \Rightarrow \mathit{Cps}$

$$\Phi[\![(\# \ M)]\!] = \mathcal{C}_c[\![M]\!][c := \lambda x.x]$$

$\mathit{Cps} \Rightarrow \Lambda_c$

$$\Phi^{-1}[\![(K \ W)]\!] = (\# \ \mathcal{C}^{-1}[\![(K \ W)]\!])$$

Figure 5: Λ_c and the CPS transformation.

The set of CPS terms reflects that the prompt actually injects answers into the domain of values. Since $\mathcal{A}$ has the opposite effect, the two syntactic categories collapse into one.

The reductions for the extended source language include all the previously defined reductions *mutatis mutandis* and the additional reductions (P) of Figure 6. The new reductions

$$
\begin{array}{rcllr}
(\#\ V) & \longrightarrow & V & & (\#_v) \\
(\#\ (\mathcal{A}\ M)) & \longrightarrow & (\#\ M) & & (\#_{\mathcal{A}}) \\
(\#\ (\mathbf{callcc}\ M)) & \longrightarrow & (\#\ (M\ (\lambda x.(\mathcal{A}\ x)))) & x \notin FV(E) & (\#_{\mathbf{callcc}}) \\
(\mathcal{A}\ (\#\ M)) & \longrightarrow & (\mathcal{A}\ M) & & (\#_{elim})
\end{array}
$$

Figure 6: The reductions P.

reflect the essential operational properties of the prompt. The prompt disappears after its subexpression becomes a value. The prompt delimits all control actions, therefore $\mathcal{A}$ can only ignore the context up to the (dynamically) closest prompt and **callcc** can only capture the continuation up to the (dynamically) closest prompt. The last reduction shows that $\mathcal{A}$ forces the evaluation of its argument to occur next to a prompt, or equivalently, in an empty continuation.

The set of reductions is complete with respect to $\boldsymbol{\lambda}\beta\eta$ on CPS terms.

Theorem 3.2 (Correspondence) *The calculi $\boldsymbol{\lambda}\beta_v\eta_v ACP$ and $\boldsymbol{\lambda}\beta\eta$ correspond to each other in the sense of Theorem 2.1.*

3.2.2 Strict prompt

The CPS transformation and its inverse remain essentially the same. However, the call-by-value semantics of the CPS language invalidates some of the CPS reductions and hence some of the source reductions. The exact reductions that become invalid depend on which theory is used for reasoning about the CPS language, which is not obvious since there is no canonical theory for call-by-value languages.

If the calculus for the by-value CPS language is the λ_v-calculus, then this means that the expression $(\#\ M)$ is no longer a value. The reductions remain essentially the same except that evaluation contexts and values are modified to reflect the new definition. Also, the reduction $(\mathcal{A}\ (\#\ M)) \longrightarrow (\mathcal{A}\ M)$ is no longer provable.

With a stronger calculus for the by-value CPS language like the $\boldsymbol{\lambda}\beta_v\eta_v A$ calculus, we can re-validate the following reductions:

$$
\begin{array}{rcl}
(\#\ (\#\ M)) & \longrightarrow & (\#\ M) \\
(\mathcal{A}\ (\#\ M)) & \longrightarrow & (\mathcal{A}\ M)
\end{array}
$$

4 Axiomatizing Observational Equivalence

To address the question of semantic completeness, we restrict our attention to typed models. The CPS type structure in Figure 7 is the Meyer/Wand [13] structure (after reversing the order of arguments to procedures). The set of answers includes only values of ground type ι. The source language is the simply typed λ-calculus augmented with the control operators

The full type frame:

$$t \;::=\; \iota \mid (t \to t)$$

The CPS type frame:

$$\begin{array}{lcll} \nu & ::= & \iota \mid (\kappa \to \kappa) & (\mathit{Values}) \\ \kappa & ::= & \nu \to a & (\mathit{Continuations}) \\ a & ::= & \iota & (\mathit{Answers}) \end{array}$$

The typed source language: Λ_c^t

$$\begin{array}{lcl} M & ::= & V \mid (M^{s\to t}\ M^s)^t \\ V & ::= & x^t \mid (\lambda x^s.M^t)^{s\to t} \\ & & \mid \mathbf{callcc}^{((s\to t)\to s)\to s} \\ & & \mid \mathcal{A}^{\iota\to t} \mid (\#\ M^\iota)^\iota \end{array}$$

The typed CPS language: CPS^t

$$\begin{array}{lcl} P^a & ::= & x^a \mid (K^{\nu\to a}\ W^\nu)^a \mid (K^{a\to a}\ P^a)^a \\ W^\nu & ::= & x^\nu \mid (\lambda k^{\kappa_1}.K^{\kappa_2})^{\kappa_1\to\kappa_2} \\ K^\kappa & ::= & k^\kappa \mid (\lambda x^\nu.P^a)^{\nu\to a} \mid (W^{\kappa_1\to\kappa_2}\ K^{\kappa_1})^{\kappa_2} \end{array}$$

Figure 7: Simply typed languages

and their usual types [22]. To apply the correspondence theorem to these typed languages, it is necessary to prove that the CPS transformation and its inverse preserve the typability of the terms.

Lemma 4.1 *Let $M \in \Lambda_c^t$, and $P \in CPS^t$, then $\mathcal{C}_k[\![M]\!] \in CPS^t$ and $\mathcal{C}^{-1}[\![P]\!] \in \Lambda_c^t$.*

As a corollary, Correspondence Theorem 3.2 extends to these typed languages.[5]

Theorem 4.2 (Correspondence for Typed Languages) *The typed calculi $\boldsymbol{\lambda}\beta_v\eta_v ACP$ and $\boldsymbol{\lambda}\beta\eta$ correspond to each other in the sense of Theorem 3.2.*

Therefore, the calculus $\boldsymbol{\lambda}\beta_v\eta_v ACP$ proves all denotational equivalences in the CPS continuous type frame.

Theorem 4.3 *Let $M, N \in \Lambda_c^t$, then $\boldsymbol{\lambda}\beta_v\eta_v ACP \vdash M = N$ if and only if the CPS continuous type frame satisfies $M = N$.*

Proof Sketch. By the correspondence theorem, $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$. Since the latter terms are also simply typed terms, the *full* continuous type frame satisfies the equation $\mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$ [16]. The result follows because the two denotational models define the same semantics. ∎

Finally, the calculus $\boldsymbol{\lambda}\beta_v\eta_v ACP$ corresponds to a restriction of the call-by-value observational equivalence that accounts for control operations.

Theorem 4.4 *Let $M, N \in \Lambda_c^t$, then $\boldsymbol{\lambda}\beta_v\eta_v ACP \vdash M = N$ if and only if M and N are call-by-value observationally equivalent in PCF contexts[6] augmented with* **callcc**, *$\mathcal{A}$, the lazy prompt, and* **pif**.

[5] This theorem provides an alternative proof for Griffin's [10] Weak Normalization Theorem for the λ_v-C-calculus. The problem of Strong Normalization for λ_v-C-calculus (or for our new calculus) remains open since reductions in the source language do not necessarily translate to reductions on the CPS side.

[6] PCF terms include numerals, **add1**, **sub1**, conditionals, and a recursion operator [18].

Proof Sketch. By modifying the full abstraction proof of Sitaram and Felleisen [22], we prove that the CPS continuous frame model satisfies $M = N$ if only if M and N are observationally equivalent. The rest follows from Theorem 4.3. ∎

5 Related Work

Our work must be compared to the two prominent theories for reasoning about languages with control operators: the λ_v-C-calculus [6, 9, 8] and the theory IOCC [24, 25]. The λ_v-C-calculus is an equational theory that approximates our calculus. The theory includes enough axioms to reason about the evaluation of programs but is incomplete with respect to the CPS transformation and with respect to semantic models. For example, it is impossible to prove the equivalence of the programs in Figure 1 in this theory. The second theory (Impredicative logic of Operations, Classes, and Control) is a first-order theory of partial terms with a subtheory of program equality. Although Talcott is not motivated by the notion of completeness nor by denotational models, her equational subtheory turns out to be equivalent to our calculus and is therefore complete by Theorem 4.4.

The development of the "direct style" transformation [5] provides an alternative reasoning system in which programs are mapped to CPS form, analyzed, and then un-translated. Because both the CPS transformation and its inverse are global transformations, this approach is expensive and inefficient. Furthermore, it does not provide any intuition to the programmer.

6 New Intermediate Languages for Compilers

We have presented an extensive and complete analysis of the expressive power of the CPS formalism. The main result of this analysis is the development of a canonical theory for reasoning about the evaluation, optimization, and implementation of control abstractions in call-by-value languages.

In the meantime, we also received a paper by Muylaert-Filho and Burn [14], which claims that programs in continuation-passing style are more informative with respect to abstract interpretations. In this context, it is interesting to see that the reductions of our calculus *increase the information content of programs* with respect to abstract interpretation. In other words, if $M \longrightarrow N$, then an abstraction interpretation of N yields at least as much information as the interpretation of M. Given that our reductions capture the essence of the CPS transformation for languages with control operators, and given that our set of new reductions is strongly normalizing, we conjecture that the normal form of a program is as informative as its CPS transform. The consequences for direct-style compilers like Chez Scheme [12] are clear: They can use the conventional compilation techniques on their inputs and still collect and use all the information that a CPS-based compiler would be able to deduce.

References

[1] Appel, A. and T. Jim. Continuation-passing, closure-passing style. In *Proc. 16th*

ACM Symposium on Principles of Programming Languages (1982), pp. 242–254. 293–302.

[2] Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.

[3] Consel C. and O. Danvy. For a better support of static data flow. In *Proc. of Functional Languages and Computer Architecture* (Cambridge, MA, 1991), pp. 496–519.

[4] Danvy, O. and A. Filinski. Representing control: A study of the CPS transformation. Tech. Rep. CIS-91-2, Kansas State University, 1991.

[5] Danvy, O. and J. L. Lawall. Back to Direct Style II: First-class continuations. In *Proc. 1992 ACM Conference on Lisp and Functional Programming* (1992).

[6] Felleisen, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages* (1988), pp. 180–190.

[7] Felleisen, M. and D.P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III* (Amsterdam, 1986), M. Wirsing, Ed., Elsevier Science Publishers B.V. (North-Holland), pp. 193–217.

[8] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Tech. Rep. 100, Rice University, June 1989. *Theor. Comput. Sci.*, 1991, to appear.

[9] Felleisen, M., D.P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Comput. Sci. 52*, 3 (1987), 205–237. Preliminary version: Reasoning with Continuations, in *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

[10] Griffin, T. A formulae-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages* (1990), pp. 47–58.

[11] Haynes, C.T., D.P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press) *11*, 3/4 (1986), 143–153.

[12] Hieb R., R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (1990), pp. 66–77.

[13] Meyer, A.R. and M. Wand. Continuation semantics in typed lambda-calculi. *Proc. Workshop Logics of Programs* (1985), 219–224. Lecture Notes in Computer Science 193, Springer-Verlag, Heidelberg.

[14] Muylaert-Filho, A. J. and G. L. Burn. Continuation semantics and abstract interpretation. Unpublished manuscript, Imperial College of Science, London, 1992.

[15] Nielson, F. A denotational framework for data flow analysis. *Acta Inf. 18* (1982), 265–287.

[16] Plotkin, G.D. Completeness for the continuous type frame. Unpublished manuscript.

[17] Plotkin, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Comput. Sci. 1* (1975), 125–159.

[18] Plotkin, G.D. LCF considered as a programming language. *Theoretical Comput. Sci. 5* (1977), 223–255.

[19] Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference* (1972), pp. 717–740.

[20] Sabry, A. and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. ACM Conference on Lisp and Functional Programming* (1992), pp. 288–298.

[21] Shivers, O. *Control-flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnegie-Mellon University, 1991.

[22] Sitaram, D. and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming* (1990), pp. 161–175.

[23] Steele, G.L., Jr. RABBIT: A compiler for SCHEME. Memo 474, MIT AI Lab, 1978.

[24] Talcott, C. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* PhD thesis, Stanford University, 1985.

[25] Talcott, C. A theory for program and data specification. In *Proc. Design and implementation of Symbolic Computation Systems, DISCO'90* (Berlin, 1990), Lecture Notes in Computer Science 429, pp. 91–100. To appear in Theor. Comput. Sci. 1992.

A Coroutines from Continuations

The original program is:

```
(define make-coroutine
  (lambda (f)
    (call/cc (lambda (maker)
               (let ([LCS 'any])
                 (let ([resume (lambda (dest val) (call/cc (lambda (k) (set! LCS k) (dest val))))])
                   (f resume (resume maker (lambda (v) (LCS v))))
                   (error "coroutine fell off end")))))))
```

Using the standard macro definition for **let**, we use the reduction C_{tail} twice, and then perform two β_v reductions for *resume* and its arguments:

```
(define make-coroutine
  (lambda (f)
    (let ([LCS 'any])
      (let ([resume (lambda (dest val) (call/cc (lambda (k) (set! LCS k) (dest val))))])
        (call/cc (lambda (maker)
                   (f resume (call/cc (lambda (k) (set! LCS k) (maker (lambda (c) (LCS v))))))
                   (error "coroutine fell off end")))))))
```

Expanding the (implicit) **begin** in the body of the (**lambda** (*maker*) ⋯) abstraction, it becomes apparent that the context (**begin** (*f resume* [ ]) (*error* "coroutine fell off end")) is an evaluation context. Therefore, we can apply the reduction C_{lift}:

```
(define make-coroutine
  (lambda (f)
    (let ([LCS 'any])
      (let ([resume (lambda (dest val) (call/cc (lambda (k) (set! LCS k) (dest val))))])
        (call/cc (lambda (maker)
                   (call/cc (lambda (kk)
                              (f resume ((lambda (k) (set! LCS k) (maker (lambda (c) (LCS v))))
                                         (lambda (x) (kk (begin (f resume x)
                                                                (error "coroutine fell off end"))))))
                              (error "coroutine fell off end")))))))))
```

Expanding the **begin** expression shows that the application of *kk* can be pushed around the second clause by a β_{lift} reduction. Since *error* is just another name for $\mathcal{A}$, the application of *kk* is ignored (*Abort*). Therefore, it is unnecessary to capture the continuation *kk* (C_{elim}). The resulting program after one β_v reduction is:

```
(define make-coroutine
  (lambda (f)
    (let ([LCS 'any])
      (let ([resume (lambda (dest val) (call/cc (lambda (k) (set! LCS k) (dest val))))])
        (call/cc (lambda (maker)
                   (begin (f resume
                             (begin (set! LCS (lambda (x) (begin (f resume x)
                                                                 (error "coroutine fell off end"))))
                                    (maker (lambda (c) (LCS v)))))
                          (error "coroutine fell off end"))))))))
```

By β_{lift}, the application of *maker* can be lifted outside the **begin** expression. Since *maker* is bound to a continuation, the reduction C_{abort} implies that it will abort its context. Therefore, we get:

```
(define make-coroutine
  (lambda (f)
    (let ([LCS 'any])
      (let ([resume (lambda (dest val) (call/cc (lambda (k) (set! LCS k) (dest val))))])
        (call/cc (lambda (maker)
                   (maker (begin (set! LCS (lambda (x) (begin (f resume x)
                                                              (error "coroutine fell off end"))))
                                 (lambda (c) (LCS v))))))))))
```

Clearly, the reductions $C_{current}$ and C_{elim} imply that the continuation *maker* need not be captured. A final β_v reduction produces the program:

```
(define make-coroutine
  (lambda (f)
    (let ([LCS 'any])
      (set! LCS (lambda (x)
                  (f (lambda (dest val) (call/cc (lambda (k) (set! LCS k) (dest val))))
                     x)
                  (error "coroutine fell off end")))
      (lambda (c) (LCS v)))))
```

and the result follows by the standard macro definition for **letrec**.[7]

[7] The above transformations were performed with a macro package for Emacs (by Cormac Flanagan) that implements the $\lambda\beta_v ACP$ reductions. The Scheme code was typeset using Dorai Sitaram's SLaTeX.