Note On Axiomatizing The Semantics of Control Operators

Amr Sabry

CIS-TR-96-03
March 1996

Abstract

We present a novel proof of the soundness and completeness of axiomatizations of call-by-value control operators with respect to the continuation semantics. Our proof is much simpler than the two previously known proofs and our resulting axioms extend previous systems to allow reasoning about all monadic effects.

Department of Computer and Information Science
University of Oregon

FIRST EDITION: March 25, 1996

1 Control Operators

For the convenience of programmers, programming languages such as Scheme and SML extend a pure call-by-value language with a variety of control operators, *e.g.*, $\mathcal{A}$ (abort) and *callcc* [2, 12]. The semantics of these control operators can be specified in two main ways: using axioms that extend the λ_v-calculus [6, 25], or by translation to continuation-passing style (CPS). Connecting these two specifications is important for at least two reasons:

1. Compiling: A relationship between the two semantics reveals the role of the CPS transformation in compilers, and clarifies the connection between direct and CPS compilers [10, 21, 22].

2. Logic: Given the Curry-Howard isomorphism, the relationship between the two semantic specifications yields a relationship between classical and intuitionistic proofs (cf. [11, 17, 18]).

Two recent papers developed proof techniques to establish the soundness and completeness of the control axioms with respect to the continuation semantics. The first proof [21] relies on a rather complicated CPS translation and its inverse, and the second [13] on non-elementary categorical notions. In this note, we present a much simpler proof.

In addition to simplifying the proof, we extend the previous results by axiomatizing the semantics of a control delimiter or prompt # [3, 4, 5, 7, 23]. The importance of this extension is due to the fact that the combination of $\mathcal{A}$, *callcc*, and # can express *all* other computational effects, *e.g.*, state, exceptions, etc [8].

The remainder of this note is organized as follows. The next section introduces a simple call-by-value language with control operators and its semantics. Section 3 discusses the main insight that led to the new proof, which is presented in detail in the next section. Section 5 concludes.

2 A Simple Call-by-Value Language

We consider a prototypical call-by-value language Λ_c with control operators. The set of terms includes variables drawn from a set *Vars*, procedures, applications, the two control operators $\mathcal{A}$ and *callcc*, and the control delimiter #:

$$\begin{array}{rcll} M & ::= & V \mid (M\ M) \mid (\mathcal{A}\ M) \mid (\mathit{callcc}\ x.M) & (\Lambda_c\text{-Terms}) \\ V & ::= & x \mid (\lambda x.M) \mid (\#\ M) & (\text{Values}) \end{array}$$

Informally, $\mathcal{A}$ permits the programmer to ignore the rest of a computation and return the value of a subexpression as the result of the entire program. The construct captures, among other computational patterns, the common idiom of an "error exit." The operator *callcc* provides the programmer with a procedural abstraction of the rest of the computation. The prompt # delimits the control actions that occur during the evaluation of its subexpression. Unlike the standard variants our delimiter is "lazy" as its subexpression is not evaluated until needed.[1]

[1]The laziness of # is due to the use of full $\beta\eta$ in the CPS language.

The language has the following context-sensitive properties. In both the expressions $(\lambda x.M)$ and $(callcc\ x.M)$ the variable x is *bound* in M. We distinguish a subset $\mathcal{K}$ of *Vars*: variables in $\mathcal{K}$ (ranged over by the meta-variables $k, k', \ldots$) can only be bound by *callcc* and cannot be bound by λ. A variable that is not bound is *free*; the set of free variables in a term M is $FV(M)$. Like Barendregt [1:ch 2,3], we identify terms modulo bound variables and we assume that free and bound variables do not interfere in definitions or theorems. The term $M[x/N]$ is the result of the capture-free substitution of all free occurrences of x in M by N. A *context* C is a term with a "hole," [ ], in the place of one subterm. The operation of *filling* the context C with a term M yields the term $C[M]$, possibly capturing some free variables of M in the process. An *evaluation context* E is a special kind of context defined as follows:

$$E ::= [\,] \mid (E\ M) \mid (V\ E) \mid (\mathcal{A}\ E)$$

2.1 CPS Semantics

We specify the semantics of Λ_c in two phases. First we translate Λ_c-terms using the standard CPS transformation [9, 16, 24]. The resulting language $\mathcal{L}_p$ has no occurrences of $\mathcal{A}$, *callcc*, or # and its semantics is axiomatized by the rules of the call-by-name λ-calculus [19].

Definition 2.1 (Call-by-Value CPS Transformation) *Let m, n be variables in Vars that do not occur in the argument to $\mathcal{P}$:*

$$\begin{array}{rcl}
\mathcal{P} : \Lambda_c & \rightarrow & \mathcal{L}_p \\
\mathcal{P}[\![x]\!] & = & \lambda k.kx \\
\mathcal{P}[\![\lambda x.M]\!] & = & \lambda k.k(\lambda x.\mathcal{P}[\![M]\!]) \\
\mathcal{P}[\![\#\ M]\!] & = & \lambda k.k(I_w\ (\mathcal{P}[\![M]\!]\ (\lambda x.I_p x))) \\
\mathcal{P}[\![MN]\!] & = & \lambda k.\mathcal{P}[\![M]\!]\ (\lambda m.\mathcal{P}[\![N]\!]\ (\lambda n.((m\ n)\ k))) \\
\mathcal{P}[\![\mathcal{A}M]\!] & = & \lambda k.\mathcal{P}[\![M]\!]\ (\lambda x.I_p x) \\
\mathcal{P}[\![callcc\ x.M]\!] & = & \lambda k.((\lambda x.\mathcal{P}[\![M]\!]k)\ (\lambda y.\lambda k'.ky))
\end{array}$$

It is straightforward to check that the CPS transformation maps source terms to CPS terms in $\mathcal{L}_p$.

Definition 2.2 (CPS Terms $\mathcal{L}_p$, Axioms X_p) *Let KVars be a set of variables disjoint from the set Vars. The meta-variables $k, k', \ldots$ ranges over the new variables:*

$$\begin{array}{rcll}
T & ::= & (\lambda k.P) \mid (W\ W) & (\mathcal{L}_p\text{-Terms}) \\
P & ::= & (K\ W) \mid (T\ K) \mid (I_p\ W) & (\text{Answers}) \\
K & ::= & k \mid (\lambda x.P) & (\text{Continuations}) \\
W & ::= & x \mid (\lambda x.T) \mid (I_w\ P) & (\text{Values})
\end{array}$$

The semantics of the CPS language is axiomatized by the following instances of the call-by-name β and η axioms:

$$\begin{array}{rcll}
((\lambda x.T)\ W) & = & T[x/W] & & (\beta_1)\\
((\lambda x.P)\ W) & = & P[x/W] & & (\beta_2)\\
((\lambda k.P)\ K) & = & P[k/K] & & (\beta_3)\\
(\lambda k.Tk) & = & T & k \notin FV(T) & (\eta_1)\\
(\lambda x.Kx) & = & K & x \notin FV(K) & (\eta_2)\\
(\lambda x.Wx) & = & W & x \notin FV(W) & (\eta_3)\\
(I_p\ (I_w\ P)) & = & P & & (id_P)\\
(I_w\ (I_p\ W)) & = & W & & (id_W)
\end{array}$$

The definition reveals a few properties of CPS terms. First every CPS term T expects a continuation argument k. Continuations K map values to answers. A value W can be injected into the set of answers using the (identity) procedure I_p and an answer can be injected into the set of values using the (identity) procedure I_w.

Our goal is to find Λ_c-axioms that prove the same identities as the axioms X_p. More precisely, we want to find a set of axioms $\mathcal{V}$ such that for Λ_c-terms M and N:

$$\mathcal{V} \vdash M = N \quad \text{if and only if} \quad X_p \vdash \mathcal{P}[\![M]\!] = \mathcal{P}[\![N]\!].$$

2.2 Direct Semantics

For the sake of presentation, we immediately introduce the axioms $\mathcal{V}$, check their soundness, and then prove their completeness in the next two sections. In practice, the axioms are discovered during the proof of completeness.

Definition 2.3 (The Axioms $\mathcal{V}$)

$$\begin{array}{rcll}
((\lambda x.M)\ V) & = & M[x/V] & & (\beta_v)\\
(\lambda x.Vx) & = & V & x \notin FV(V) & (\eta_v)\\
((\lambda x.E[x])\ M) & = & E[M] & x \notin FV(E) & (\beta_\Omega)\\
 & & & & \\
E[\mathit{callcc}\ x.M] & = & \mathit{callcc}\ y.E[M[x/(\lambda a.(y\ E[a]))]] & y, a \notin FV(E, M) & (C_{lift})\\
\mathit{callcc}\ x.xM & = & \mathit{callcc}\ x.M & & (C_{curr})\\
\mathit{callcc}\ x.M & = & M & x \notin FV(M) & (C_{elim})\\
E[(k\ M)] & = & (k\ M) & k \in \mathcal{K} & (C_{\mathcal{A}})\\
 & & & & \\
E[(\mathcal{A}\ M)] & = & (\mathcal{A}\ M) & & (Abort)\\
 & & & & \\
(\#\ V) & = & V & & (\#_v)\\
(\#\ (\mathcal{A}\ M)) & = & (\#\ M) & & (\#_{\mathcal{A}})\\
(\mathcal{A}\ (\#\ M)) & = & (\mathcal{A}\ M) & & (\#_{elim})
\end{array}$$

The axioms have the following intuitive explanation. The first three axioms are equivalent to Moggi's computational λ-calculus [14]. The axiom C_{lift} characterizes the capture of continuations via *callcc* [6]. The axiom C_{curr} shows that the current continuation is always implicitly applied. The axiom C_{elim} is a garbage-collection rule: continuations captured but not used can be collected. The axioms $C_{\mathcal{A}}$ and *Abort* show that continuations and $\mathcal{A}$ abort their context upon invocation. The last three axioms show that a prompt disappears when its subexpression reduces to a value, and that control actions are indeed delimited by #.

The axioms for *callcc* and $\mathcal{A}$ are well-known [6, 25], and have been shown sound and complete with respect to the CPS semantics using two independent proofs [13, 21].[2] The axioms for # are less known; the axiom $\#_{elim}$ appears to be original.

Proposition 2.1 (Soundness) *Let* $M, N \in \Lambda_c$, $\mathcal{V} \vdash M = N$ implies $X_p \vdash \mathcal{P}[\![M]\!] = \mathcal{P}[\![N]\!]$.

3 Propagation of Continuations

The salient aspect of CPS terms is that they explicitly propagate the continuation from one computation to the next. In fact, traditional CPS terms are hardwired to propagate the continuation in *one* specific way via procedure calls. Thus, not only is the CPS term $T = (\lambda k.P)$ parameterized over a continuation k, but it also expects this continuation to be passed as an argument, *i.e.*, via a procedure call $(T\ K)$.

The main insight of this note is that we could replace the two terms $(\lambda k.P)$ and $(T\ K)$ that are responsible for the propagation of the continuation via procedure calls by the "black boxes" $(\textsf{get}\ k.P)$ and $(\textsf{send}\ K\ T)$. The details of the implementation of the black boxes are irrelevant as long as their external behavior satisfies the same equations that are satisfied by the CPS implementation:

$$\begin{array}{rcll} (\textsf{send}\ K\ (\textsf{get}\ k.P)) & = & P[k/K] & & (\beta_3) \\ (\textsf{get}\ k.(\textsf{send}\ k\ T)) & = & T & k \notin FV(T) & (\eta_1) \end{array}$$

To formalize this idea, we define opaque CPS terms that hide the details of *how* the continuation propagates and the specific injections between values and answers.

Definition 3.1 (Opaque CPS Transformation) *Let m, n be variables in Vars that do*

[2]In our previous analysis [20], we also included the following three axioms:

$$\begin{array}{rcll} E[((\lambda x.M)\ N)] & = & ((\lambda x.E[M])\ N) & x \notin FV(E) & (\beta_{lift}) \\ \textit{callcc}\ y.((\lambda x.M)\ N) & = & ((\lambda x.\textit{callcc}\ y.M)\ N) & y \notin FV(N) & (C_{tail}) \\ (\#\ (\textit{callcc}\ x.M)) & = & (\#\ M[x/(\lambda a.(\mathcal{A}\ a))]) & & (\#_{callcc}) \end{array}$$

In a private message, Filinski pointed out that the first axiom is provable from (β_v) and (β_Ω). Using the new proof technique in this paper, we have discovered that the second and third identities are also provable from the other axioms.

not occur in the argument to $\mathcal{C}$:

$$\begin{aligned}
\mathcal{C} : \Lambda_c &\rightarrow \mathcal{L}_\bullet \\
\mathcal{C}[\![x]\!] &= \textsf{get } k.kx \\
\mathcal{C}[\![\lambda x.M]\!] &= \textsf{get } k.k(\lambda x.\mathcal{C}[\![M]\!]) \\
\mathcal{C}[\![\# \ M]\!] &= \textsf{get } k.k(\textsf{injw } (\textsf{send } (\lambda x.\textsf{injp } x) \ \mathcal{C}[\![M]\!])) \\
\mathcal{C}[\![MN]\!] &= \textsf{get } k.(\textsf{send } (\lambda m.(\textsf{send } (\lambda n.(\textsf{send } k \ (m \ n))) \ \mathcal{C}[\![N]\!])) \ \mathcal{C}[\![M]\!]) \\
\mathcal{C}[\![\mathcal{A}M]\!] &= \textsf{get } k.\textsf{send } (\lambda x.\textsf{injp } x) \ \mathcal{C}[\![M]\!] \\
\mathcal{C}[\![\mathit{callcc}\ x.M]\!] &= \textsf{get } k.((\lambda x.\textsf{send } k \ \mathcal{C}[\![M]\!]) \ (\lambda y.\textsf{get } k'.ky))
\end{aligned}$$

The set of opaque terms and their semantics are exactly like for regular CPS terms except that we no longer know about (nor care) how the continuation is passed and how values and answers are injected into each other.

Definition 3.2 (Opaque CPS Terms $\mathcal{L}_\bullet$, Axioms $X_\bullet$) *We use four black boxes* get, send, injp, *and* injw. *Let KVars be a set of variables disjoint from the set Vars. The meta-variables $k, k', \ldots$ ranges over the new variables:*

$$\begin{aligned}
T &::= (\textsf{get } k.P) \mid (W \ W) && (\mathcal{L}_\bullet\text{-Terms}) \\
P &::= (K \ W) \mid (\textsf{send } K \ T) \mid (\textsf{injp } W) && (\text{Answers}) \\
K &::= k \mid (\lambda x.P) && (\text{Continuations}) \\
W &::= x \mid (\lambda x.T) \mid (\textsf{injw } P) && (\text{Values})
\end{aligned}$$

The axioms $X_\bullet$ consist of (β_1), (β_2), (η_2), (η_3) in Definition 2.2 and the following opaque versions of the remaining axioms:

$$\begin{aligned}
(\textsf{send } K \ (\textsf{get } k.P)) &= P[k/K] && && (\beta_3) \\
(\textsf{get } k.(\textsf{send } k \ T)) &= T && k \notin FV(T) && (\eta_1) \\
(\textsf{injp } (\textsf{injw } P)) &= P && && (id_P) \\
(\textsf{injw } (\textsf{injp } W)) &= W && && (id_W)
\end{aligned}$$

4 Continuation-Grabbing Style

As far as the external behavior of CPS terms is concerned, it is possible to implement the black boxes as follows:

$$\begin{aligned}
(\textsf{get } k.P) &\overset{\text{def}}{=} (\mathit{callcc}\ k.P) & (\textsf{send } K \ T) &\overset{\text{def}}{=} (K \ T) \\
(\textsf{injp } W) &\overset{\text{def}}{=} (\mathcal{A} \ W) & (\textsf{injw } P) &\overset{\text{def}}{=} (\# \ P)
\end{aligned}$$

Intuitively the idea is simply a change of perspective in the direction of the flow of the continuation. Instead of passively waiting for the continuation as an argument, each procedure instead *grabs* the continuation using *callcc*. This implementation of the black boxes yields a transformation similar to CPS that we call the continuation-grabbing style (CGS) transformation.

Definition 4.1 (Call-by-Value CGS Transformation) *Let m, n be variables in Vars that do not occur in the argument to $\mathcal{G}$:*

$$\begin{array}{rcl}
\mathcal{G} : \Lambda_c & \rightarrow & \mathcal{L}_g \\
\mathcal{G}[\![x]\!] & = & callcc\ k.kx \\
\mathcal{G}[\![\lambda x.M]\!] & = & callcc\ k.(k\ (\lambda x.\mathcal{G}[\![M]\!])) \\
\mathcal{G}[\![\#\ M]\!] & = & callcc\ k.k\ (\#\ ((\lambda x.(\mathcal{A}\ x))\ \mathcal{G}[\![M]\!])) \\
\mathcal{G}[\![M N]\!] & = & callcc\ k.((\lambda m.((\lambda n.(k\ (m\ n)))\ \mathcal{G}[\![N]\!]))\ \mathcal{G}[\![M]\!]) \\
\mathcal{G}[\![\mathcal{A}\ M]\!] & = & callcc\ k.((\lambda x.(\mathcal{A}\ x))\ \mathcal{G}[\![M]\!]) \\
\mathcal{G}[\![callcc\ x.M]\!] & = & callcc\ k.((\lambda x.(k\ \mathcal{G}[\![M]\!]))\ (\lambda y.callcc\ k'.ky))
\end{array}$$

The remarkable property of the CGS transformation is that although it is a specific implementation of the opaque CPS transformation, it is the identity function on source terms.

Proposition 4.1 *Let M be a source term in Λ_c, then $\mathcal{V} \vdash M = \mathcal{G}[\![M]\!]$.*

The grammar for CGS terms is naturally obtained from the grammar for opaque CPS terms.

Definition 4.2 (CGS Terms $\mathcal{L}_g$) *Let KVars be a set of variables disjoint from the set Vars. The meta-variables $k, k', \ldots$ ranges over the new variables:*

$$\begin{array}{rcll}
T & ::= & (callcc\ k.P) \mid (W\ W) & (\mathcal{L}_g\text{-Terms}) \\
P & ::= & (K\ W) \mid (K\ T) \mid (\mathcal{A}\ W) & (\text{Answers}) \\
K & ::= & k \mid (\lambda x.P) & (\text{Continuations}) \\
W & ::= & x \mid (\lambda x.T) \mid (\#\ P) & (\text{Values})
\end{array}$$

It remains to show that each of the CGS version of the axioms $X_\bullet$ (which we call X_g) is provable using the axioms $\mathcal{V}$.

Definition 4.3 (Equations X_g) *The equations X_g consist of (β_1), (β_2), (η_2), (η_3) in Definition 2.2 and:*

$$\begin{array}{rclll}
(callcc\ K\ (callcc\ k.P)) & = & P[k/K] & & (\beta_3) \\
(callcc\ k.(k\ T)) & = & T & k \notin FV(T) & (\eta_1) \\
(\mathcal{A}\ (\#\ P)) & = & P & & (id_P) \\
(\#\ (\mathcal{A}\ W)) & = & W & & (id_W)
\end{array}$$

Although the equations X_g do not hold between arbitrary source terms, they hold between CGS terms.

Lemma 4.2 *For every equation $P = Q$ in X_g, $\mathcal{V} \vdash P = Q$.*

Proof. The equations (β_1), (β_2), (η_2) and (η_3) are instances of β_v and η_v and hence are provable. The equation (η_1) is provable using C_{curr} and C_{elim}. The equation (id_W) is provable using $\#_{\mathcal{A}}$ and $\#_v$. For the remaining two equations (β_3) and (id_P), we calculate as follows:

$$\begin{array}{rcll}
(K\ (callcc\ k.P)) & = & callcc\ k'.(K\ P[k/(\lambda x.k'(K\ x))]) & \text{by } C_{lift} \\
& = & callcc\ k'.(K\ P[k/(\lambda x.(K\ x))]) & \text{by Lemma 4.3} \\
& = & callcc\ k'.(K\ P[k/K]) & \text{by } \eta_v \\
& = & callcc\ k'.P[k/K] & \text{by Lemma 4.3} \\
& = & P[k/K] & \text{by } C_{elim}
\end{array}$$

Similarly:

$$\begin{array}{rcll}
(\mathcal{A}\ (\#\ P)) & = & (\mathcal{A}\ P) & \text{by } \#_{elim} \\
& = & P & \text{by Lemma 4.3}
\end{array}$$

Lemma 4.3 *For any CGS term in the syntactic category P, $\mathcal{V} \vdash E[P] = P$.*

Proof. The proof is by induction on the structure of the term P and proceeds by cases:

- $P = (K\ W)$, or $P = (K\ T)$. Then, we have two cases:
  - $K = k$, then the result holds by the axiom $C_{\mathcal{A}}$.
  - $K = (\lambda x.P')$, then $P = ((\lambda x.P')\ X)$ where X is either W or T and:

$$\begin{array}{rcll}
E[P] & = & E[((\lambda x.P')\ X)] & \\
& = & ((\lambda x.E[P'])\ X) & \text{by } \beta_{lift} \\
& = & ((\lambda x.P')\ X) & \text{by the inductive hypothesis} \\
& = & P &
\end{array}$$

- $P = (\mathcal{A}\ W)$, then the result hold by the axiom *Abort*.

By pasting together the previous results we get our desired proof of completeness.

Theorem 4.4 *Let M and N be source terms in Λ_c:*

$$X_p \vdash \mathcal{P}[\![M]\!] = \mathcal{P}[\![N]\!] \quad \text{implies} \quad \mathcal{V} \vdash M = N$$

Proof. Assume $X_p \vdash \mathcal{P}[\![M]\!] = \mathcal{P}[\![N]\!]$, then the opaque CPS axioms prove the equation between opaque CPS terms, *i.e.*, $X_\bullet \vdash \mathcal{C}[\![M]\!] = \mathcal{C}[\![N]\!]$. It follows that $X_g \vdash \mathcal{G}[\![M]\!] = \mathcal{G}[\![N]\!]$. The result follows by Lemma 4.2 and Proposition 4.1.

5 Other Monadic Effects

With the axiomatization of the control operators *callcc*, $\mathcal{A}$, and $\#$, we can reason about any monadic effect whose definition can be expressed in a purely functional language, *e.g.*, exceptions, state, lists, parsers, continuations (!), etc.

Given a monad with unit operation η and extension operation *bind*, computational effects can be expressed in two ways:

1. write programs in "effect-passing style" [26], or
2. extend the source language with the operations *reflect* and *reify* to gain access to the monad [8, 15].

The disadvantage of the first approach is that it requires a global transformation which obscures the original program. Its advantage is that the semantics of the translated programs is axiomatized by the simple axioms $\beta\eta$ of the λ-calculus. The disadvantage of the second approach is that there is no known axiomatization of the operations *reflect* and *reify* for every monad.

Recently Filinski [8] shows how to represent the operations *reflect* and *reify* for every monad as follows:

$$\begin{aligned}(\mathit{reflect}\ M) &\overset{\text{def}}{=} \mathit{callcc}\ k.(\mathcal{A}\ (\mathit{bind}\ M\ (\lambda a.\#\ (k\ a))))\\ (\mathit{reify}\ M) &\overset{\text{def}}{=} \#\ (\eta\ M)\end{aligned}$$

In other words, given our axioms for *callcc*, $\mathcal{A}$, and #, we can reason about the pair of operations *reflect* and *reify* of *any* monad. Moreover, our reasoning system is as powerful as the one obtained by applying the global program transformation and using $\beta\eta$ on the translated terms.

References

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

[2] Clinger, W., and Rees, J. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers 4*, 3 (1991), 1–55.

[3] Danvy, O., and Filinski, A. Abstracting control. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 151–160.

[4] Danvy, O., and Filinski, A. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science 2*, 4 (1992), 361–391.

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

[6] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 102* (1992), 235–271. Technical Report 89-100, Rice University.

[7] Felleisen, M., Wand, M., Friedman, D., and Duba, B. Abstract continuations: A mathematical semantics for handling full functional jumps. In *ACM Conference on Lisp and Functional Programming* (1988), pp. 52–62.

[8] Filinski, A. Representing monads. In *ACM Symposium on Principles of Programming Languages* (1994), pp. 446–457.

[9] Fischer, M. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs* (1972), SIGPLAN Notices, 7, 1, pp. 104–109. Revised version in *Lisp and Symbolic Computation*, **6**, 3/4, (1993) 259-287.

[10] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1993), pp. 237–247.

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

[12] Harper, R., Duba, B. F., and MacQueen, D. Typing first-class continuations in ml. *Journal of Functional Programming 3 part 4* (1993), 465–484. Preliminary version in POPL991.

[13] Hofmann, M. Sound and complete axiomatisations of call-by-value control operators. *Mathematical Structures in Computer Science, Special Issue on the Proceedings of the 5th Conference on Category Theory and Computer Science (1993)* (To appear).

[14] Moggi, E. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science* (1989), pp. 14–23. Also appeared as: LFCS Report ECS-LFCS-88-86, University of Edinburgh, 1988.

[15] Moggi, E. Notions of computation and monads. *Information and Computation 93* (1991), 55–92.

[16] Morris, L. The next 700 formal language descriptions. *Lisp and Symbolic Computation 6*, 3/4 (1993), 249–256. Original manuscript dated 1970.

[17] Murthy, C. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell, 1990.

[18] Murthy, C. An evaluation semantics for classical proofs. In *IEEE Symposium on Logic in Computer Science* (1991).

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

[20] Sabry, A. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms.* PhD thesis, Rice University, 1994. Available as TR94-241.

[21] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation 6*, 3/4 (1993), 289–360. Also in Proceedings of the ACM Conference on Lisp and Functional Programming, 1992, and Technical Report 92-180, Rice University.

[22] Sabry, A., and Felleisen, M. Is continuation-passing useful for data flow analysis? In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 1–12.

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

[24] Strachey, C., and Wadsworth, C. Continuations: A mathematical semantics for handling full jumps. Technical monograph prg-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

[25] Talcott, C. L. A theory for program and data specification. *Theoretical Computer Science 104* (1992), 129–159. Preliminary version in Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems, (Lecture Notes in Computer Science, 429, 1990).

[26] Wadler, P. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming* (1990).