Explicit Polymorphism and CPS Conversion

Robert Harper* Mark Lillibridge†
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We study the typing properties of CPS conversion for an extension of F_ω with control operators. Two classes of evaluation strategies are considered, each with call-by-name and call-by-value variants. Under the "standard" strategies, constructor abstractions are values, and constructor applications can lead to non-trivial control effects. In contrast, the "ML-like" strategies evaluate beneath constructor abstractions, reflecting the usual interpretation of programs in languages based on implicit polymorphism. Three continuation passing style sub-languages are considered, one on which the standard strategies coincide, one on which the ML-like strategies coincide, and one on which all the strategies coincide. Compositional, type-preserving CPS transformation algorithms are given for the standard strategies, resulting in terms on which all evaluation strategies coincide. This has as a corollary the soundness and termination of well-typed programs under the standard evaluation strategies. A similar result is obtained for the ML-like call-by-name strategy. In contrast, such results are obtained for the call-by value ML-like strategy only for a restricted sub-language in which constructor abstractions are limited to values.

*This work was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628–91–C–0168. Electronic mail address: `rwh@cs.cmu.edu`.

†Supported by a National Science Foundation Graduate Fellowship. Electronic mail address: `mdl@cs.cmu.edu`.

1 Introduction

Among the many advances in the theory and practice of programming language design, the concepts of polymorphism [14, 28, 39] and continuation-passing [38, 41, 43] are of particular interest. The use of polymorphism in a practical programming language was first explored in ML [15, 28, 29]. This style of polymorphism, called *implicit* polymorphism, is based on the idea that programs are type-free, with types interpreted as predicates expressing properties of programs under evaluation. Numerous extensions of these ideas have been explored in the literature ([7, 25, 31, 45], to name just a few).

Although implicit polymorphism is appealingly simple and natural, it does not scale well to more sophisticated language features such as modularity and abstract types [20]. Recent languages, notably Quest [4] and LEAP [35], are based instead on the notion of *explicit* polymorphism introduced by Girard and Reynolds [39]. In these languages, types are an intrinsic part of the programming notation; in particular, polymorphic abstraction and application are expression-forming operations. Some of the convenience of implicit polymorphism may be restored by allowing the omission of certain forms of type information, provided that it can be unambiguously recovered by a *type reconstruction* algorithm [4, 26, 34].

A type discipline is primarily a means of enforcing levels of abstraction [40], and as such is primarily concerned with the static structure and properties of programs. Matters of control are elegantly addressed using the method of continuations. The semantics of control operations may be concisely expressed using continuations [9, 36, 38, 42, 43]. Important control constructs such as co-routines [21] and user-level threads [5, 37] can be defined using primitives for "reifying" continuations. Conversion into "continuation-passing style" (CPS) is a useful compilation technique for higher-order functional languages [3, 2, 23, 41]. Continuations are central to eliciting the computational content of proofs in classical logic [16, 17, 32] and provide a computational interpretation of classical linear logic [12].

The addition of continuation primitives to polymorphic languages has not, however, been an unalloyed success. In particular, a very natural typing discipline for first-class continuations in Standard ML has proved to be unsound [18, 19]. Since the semantics of first-class continuations may be expressed by conversion into continuation-passing style, it is natural to investigate their typing properties by considering the relation between the type of a term and the type of its CPS transform. Work in this area was initiated by Meyer and Wand for a call-by-value interpretation of the simply-typed λ-calculus [27], and extended to continuation-passing primitives by Griffin [16] and Duba, *et. al.* [8]. In earlier work, the authors extended these analyses to implicit polymorphism, and established some limitative results [19].

In this paper, we conduct a systematic investigation of the typing properties of CPS conversion for F_ω+$\mathsf{control}$, the higher-order polymorphic λ-calculus of Girard and Reynolds [14, 39] extended with the control primitives $callcc$ and $abort$. Extensions and variations of F_ω lie at the core of Quest [4] and LEAP [35], and it is the underlying programming language of the Calculus of Constructions [6, 33]. We extend F_ω with control primitives in order to illustrate the role of "impure" programming language features in the analysis of typing properties of realistic programming languages. (Similar issues and trade-offs arise with mutable data structures (see Tofte [44]) and exceptions [46]. See Leroy [24] for related discussion.)

We consider two classes of evaluation strategies for F_ω+$\mathsf{control}$, each with a call-by-value and a call-by-name variant. Under the "standard" strategies, type abstractions are values and type applications are significant evaluation steps. These strategies are compatible with extensions to the language involving primitive operations that are sensitive to type information — *e.g.*, storage allocation operations that determine the size of the allocation based on the type of the argument. The "ML-like" strategies are inspired by implicit polymorphism. Under these strategies, evaluation proceeds beneath type abstractions. This limits the ability of primitive operations to use types because types are no longer always ground types. (In particular, they may contain free type variables.) The full language enjoys the subject reduction property for complete programs evaluated under the standard strategies and the ML-like call-by-name strategy, but only a restricted language enjoys this property when interpreted under the ML-like call-by-value strategy.

The focus of our study is on the typing properties of CPS conversion of F_ω+$\mathsf{control}$, following the seminal work of Plotkin [36] (extended by Felleisen, *et. al.* [10, 9]) and Meyer and Wand [27] (extended by Griffin [16] and Duba, *et. al.* [8, 18]). First, we isolate several "continuation-passing style" sub-languages of F_ω. The "standard" CPS language is the largest sub-language of F_ω on which the by-value and by-name variants of the standard strategies coincide, and the "ML-like" CPS language is the largest sub-language on which the ML-like strategies coincide. The ML-like CPS form is a proper subset of the standard CPS form, and hence the two variants of the standard strategy and the two variants of the ML-like strategy coincide with each other on terms in ML-like CPS form. However, the standard call-by-value (call-by-name) and ML-like call-by-value (call-by-name) strategies do *not* coincide on terms in ML-like CPS form. We define a "strict" CPS form on which all four strategies coincide.

With this in mind, we define a CPS conversion algorithm for each of the standard strategies that preserves typing in a generalization of the Meyer-Wand sense, and which yields terms in strict CPS form. It turns out that we can use the standard call-by-name algorithm to handle the ML-like call-by-name strategy case as well. Such a result can be achieved for the ML-like call-by-value strategy only if we restrict attention to a restriction F_ω^-+$\mathsf{control}$ in which type abstractions are limited to values. On this fragment, the ML-like and standard strategies coincide, and hence the standard CPS algorithms may be used for the ML-like interpretations. However, the standard CPS conversion algorithms do not adequately reflect the "spirit" of the ML-like strategies, and we therefore consider variant transforms that do embody this "spirit" but which yield terms in a "relaxed" CPS form introduced solely for this purpose.

2 The Language F_ω+$\mathsf{control}$

The language F_ω+$\mathsf{control}$ is the extension of the "pure" F_ω language by two primitive control operators, $callcc_A(-)$ and $abort_A(-)$. By pure we mean that the language has no effect producing terms where an effect is something other than a simple value computation. Effects include side effects (i.e., assignment), non-termination, and non-local control changes. For the purposes of this paper, when we say something is pure, we mean it does not contain any control operators.

Definition 2.1 (Syntax)

$$
\begin{array}{llcl}
Kinds & K & ::= & \Omega \mid K_1 \Rightarrow K_2 \\
Constructors & A & ::= & \alpha \mid u \mid A_1 \to A_2 \mid \forall u{:}K.A \mid \\
 & & & \lambda u{:}K.A \mid A_1\, A_2 \\
Terms & M & ::= & x \mid \lambda x{:}A.M \mid M_1\, M_2 \mid \\
 & & & \Lambda u{:}K.M \mid M\{A\} \mid \\
 & & & callcc_A(M) \mid abort_A(M) \\
 & & & \\
Constr.\ Contexts & \Delta & ::= & \emptyset \mid \Delta, u{:}K \\
Term\ Contexts & \Gamma & ::= & \emptyset \mid \Gamma, x{:}A
\end{array}
$$

The meta-variable u ranges over *constructor variables*, and the meta-variable x ranges over *term variables*. The constructor α is a distinguished base type, representing the type of "answers". We make $abort$ and $callcc$ primitives taking one type and term argument each as a technical device to simplify the direct semantics. The more usual definitions of $abort$ and $callcc$ as special constants can be recovered by using the following definitions:

$$\begin{array}{lcl} abort & = & \Lambda u{:}\Omega.\lambda x{:}\alpha.\ abort_u(x) \\ callcc & = & \Lambda u{:}\Omega.\lambda x{:}((\forall v{:}\Omega.u \to v) \to u).\ callcc_u(x) \end{array}$$

Note that due to the greater expressiveness of the F_ω type system, it is not necessary to introduce a *throw* operator or a special type of continuations as it is in ML [8].
The typing rules for F_ω+control appear in the appendix.

3 Operational Semantics for F_ω+control

In this section, we introduce the two main evaluation strategies for F_ω+control, each with a call-by-value and a call-by-name variant.

The "standard" strategies treat constructor abstractions as values and constructor applications as significant computation steps. Standard strategies in this sense are used in Quest [4] and LEAP [35], and are directly compatible with extensions that make significant uses of types at run time (for example, "dynamic" types [1, 4]). Since polymorphic expressions are kept distinct from their instances, the anomalies that arise in implicitly polymorphic languages in the presence of references [45] and control operators [18] do not occur.

The "ML-like" strategies are inspired by the operational semantics of ML [29]. Evaluation proceeds beneath constructor abstractions, leading to a once-for-all-instances evaluation of polymorphic terms. Constructor application is retained as a computation step, but its force is significantly attenuated by the fact that type expressions may have free type variables in them, precluding primitives that inductively analyze their type arguments. The superficial efficiency improvement gained by evaluating beneath type abstractions comes at considerable cost since it is incompatible with extensions such as mutable data structures and control operators [45, 18, 19].

3.1 Notation

The definitions of these strategies make use of Plotkin's notion of a *syntactic value* [36] and Felleisen's notion of an *evaluation context* [11], chosen suitably for each situation. To specify a strategy using this method, we first give a grammar which defines three syntactic categories: V, a set of *values*, R, a set of *redices*, and E, a set of *evaluation contexts*. As an example, the grammar used to specify a call-by-value strategy for the simply-typed fragment of F_ω is as follows:

$$\begin{array}{llcl} Values & V & ::= & x \mid \lambda x{:}A.M \\ Redices & R & ::= & (\lambda x{:}A.M)\,V \\ Evaluation\ Contexts & E & ::= & [] \mid E\,M \mid V\,E \end{array}$$

The expression [] is called a "hole"; an evaluation context has exactly one occurrence of a hole. If E is an evaluation context, we write $E[M]$ for the result of "filling the hole" in E with M, possibly incurring capture of free variables in M.

A *program* is a closed term P of type α. Unless we say otherwise, programs and terms are drawn from the full F_ω+control language and typed using F_ω+control. Pure programs and terms can be considered to be drawn from and typed using F_ω.

We will arrange things so that a program P can only be represented in at most one way as $E[R]$ where E is an evaluation context and R is a redex. If P can be so represented, then E is said to be the *program context* of P, while R is said to be the *current redex* of P. If P can not be so represented, it is considered to be in normal form for the strategy. In order to complete the specification of a strategy, we must specify how to reduce (by one step) each possible kind of redex given its surrounding context. For the example strategy, the reduction rules are as follows:

$$E[(\lambda x{:}A.M)\,V] \quad \hookrightarrow \quad E[[V/x]M]$$

It should be noted that in all the strategies we consider, values are in normal form for that strategy. We say that a program P *evaluates* to a value V iff $P \hookrightarrow^* V$, *i.e.*, iff V is the terminus of a maximal one-step evaluation sequence starting at P.

3.2 Standard Strategies

We consider two "standard" evaluation strategies, *call-by-value* and *call-by-name*. In both cases constructor abstractions are values, and constructors applications are significant computation steps. The two variants differ from one another in the treatment of ordinary applications.

3.2.1 Call-By-Value (CBV) Strategy

The standard call-by-value strategy is defined as follows:

$$\begin{array}{lcl} V & ::= & x \mid \lambda x{:}A.M \mid \Lambda u{:}K.M \\ R & ::= & (\lambda x{:}A.M)\,V \mid (\Lambda u{:}K.M)\{A\} \mid \\ & & abort_A(M) \mid callcc_A(M) \\ E & ::= & [] \mid E\,M \mid V\,E \mid E\{A\} \end{array}$$

$$\begin{array}{rcl} E[(\lambda x{:}A.M)\,V] & \hookrightarrow_{cbv} & E[[V/x]M] \\ E[(\Lambda u{:}K.M)\{A\}] & \hookrightarrow_{cbv} & E[[A/u]M] \\ E[abort_A(M)] & \hookrightarrow_{cbv} & M \\ E[callcc_A(M)] & \hookrightarrow_{cbv} & \\ \multicolumn{3}{l}{E[M\ (\Lambda u{:}\Omega.\lambda x{:}A.\ abort_u(E[x]))]\ (u \notin FTV(A))} \end{array}$$

Theorem 3.1 (Decomposition) *If M is a closed, well-typed term of type A, then either M is a CBV value, or else there exists a unique CBV evaluation context E, a unique CBV redex R, and a type expression B such that*

1. $M = E[R]$;

2. F_ω+control $\vdash \emptyset; \emptyset \triangleright R : B$;

3. F_ω+control $\vdash \emptyset; x{:}B \triangleright E[x] : A$.

Theorem 3.2 (Subject Reduction) *If P is a program, and $P \hookrightarrow_{cbv} Q$, then Q is a program.*

Proof: If $P \hookrightarrow_{cbv} Q$, then by the decomposition theorem $P = E[R]$ for some CBV evaluation context E and CBV redex R such that F_ω+control $\vdash \emptyset; x{:}B \triangleright E[x] : \alpha$ and F_ω+control $\vdash \emptyset; \emptyset \triangleright R : B$ for some type B. Using this, it is straightforward to verify that each of the evaluation rules preserves typing. □

It follows from these two theorems that a terminating CBV evaluation sequence starting from a program terminates with a CBV value of type α — CBV evaluation does not "get stuck". The restriction of CBV evaluation to pure programs is a particular β-reduction strategy. It follows from the strong normalization property of F_ω [14, 13] that CBV evaluation of pure programs terminates. Termination of CBV evaluation for full F_ω+control will be established in Section 5. The following property of CBV evaluation will be important to that argument.

Lemma 3.3 *Any infinite CBV evaluation sequence starting from a program contains infinitely many β-reduction steps.*

Proof: If $E[\mathcal{C}_A(M)] \hookrightarrow_{cbv} E'[\mathcal{C}'_{A'}(M')]$ where $\mathcal{C}, \mathcal{C}' \in \{abort,\ callcc\}$ and $E[\mathcal{C}_A(M)]$ is a program then M' is a proper subterm of M. □

3.2.2 Call-By-Name (CBN) Strategy

The standard call-by-name strategy is defined as follows:

$$\begin{array}{rcl} V & ::= & \lambda x{:}A.M \mid \Lambda u{:}K.M \\ R & ::= & (\lambda x{:}A.M_1)\, M_2 \mid (\Lambda u{:}K.M)\{A\} \mid \\ & & abort_A(M) \mid callcc_A(M) \\ E & ::= & [] \mid E\, M \mid E\{A\} \end{array}$$

$$\begin{array}{rcl} E[(\lambda x{:}A.M_1)\, M_2] & \hookrightarrow_{cbn} & E[[M_2/x]M_1] \\ E[(\Lambda u{:}K.M)\{A\}] & \hookrightarrow_{cbn} & E[[A/u]M] \\ E[abort_A(M)] & \hookrightarrow_{cbn} & M \\ E[callcc_A(M)] & \hookrightarrow_{cbn} & \\ \multicolumn{3}{l}{E[M\ (\Lambda u{:}\Omega.\lambda x{:}A.\ abort_u(E[x]))]\ (u \notin FTV(A))} \end{array}$$

The decomposition and subject reduction theorems (stated above for the CBV strategy) can be proved in a similar way for the call-by-name strategy case. The analysis of termination is identical. Once again, an infinite CBN evaluation sequence must contain infinitely many β steps.

3.3 ML-like Strategies

An evaluation strategy is said to be *ML-like* if it evaluates under constructor abstractions. We shall consider two ML-like strategies, a call-by-value variant, designated *ML-CBV*, and a call-by-name variant, designated *ML-CBN*.

3.3.1 ML-CBV Strategy

The ML-like call-by-value strategy is defined as follows:

$$\begin{array}{rcl} V & ::= & x \mid \lambda x{:}A.M \mid \Lambda u{:}K.V \\ R & ::= & (\lambda x{:}A.M)\, V \mid (\Lambda u{:}K.V)\{A\} \mid \\ & & abort_A(M) \mid callcc_A(M) \\ E & ::= & [] \mid E\, M \mid V\, E \mid \Lambda u{:}K.E \mid E\{A\} \end{array}$$

$$\begin{array}{rcl} E[(\lambda x{:}A.M)\, V] & \hookrightarrow_{ml-cbv} & E[[V/x]M] \\ E[(\Lambda u{:}K.V)\{A\}] & \hookrightarrow_{ml-cbv} & E[[A/u]V] \\ E[abort_A(M)] & \hookrightarrow_{ml-cbv} & M \\ E[callcc_A(M)] & \hookrightarrow_{ml-cbv} & \\ \multicolumn{3}{l}{E[M\ (\Lambda u{:}\Omega.\lambda x{:}A.\ abort_u(E[x]))]\ (u \notin FTV(A))} \end{array}$$

Notice that a constructor abstraction is a ML-CBV value only if its body is a ML-CBV value and that ML-CBV evaluation contexts may extend within the scopes of constructor abstractions. The decomposition property for the ML-CBV strategy is somewhat more complex than that for the standard CBV strategy due to the possibility of evaluation under constructor abstractions.

Theorem 3.4 (Decomposition) *If M is a well-typed, closed term of type A, then either M is a ML-CBV value, or there exists a unique ML-CBV evaluation context E, a unique ML-CBV redex R, a constructor context Δ, and a type expression B such that*

1. $M = E[R]$;

2. F_ω+control $\vdash \Delta; \emptyset \triangleright R : B$;

3. F_ω+control $\vdash \emptyset; \emptyset \triangleright E[N] : A$ *for any term N such that* F_ω+control $\vdash \Delta; \emptyset \triangleright N : B$.

Notice that the typing condition on E is strictly weaker than the condition F_ω+control $\vdash \Delta; x{:}B \triangleright E[x] : A$.

Theorem 3.5 (Subject Reduction for F_ω) *If P is a pure program and $P \hookrightarrow_{ml-cbv} Q$, then Q is a pure program.*

Proof: Follows from the fact that the restriction of the ML-CBV strategy to terms of F_ω is a particular β-reduction strategy and from subject reduction for F_ω. □

Similarly, since F_ω is strongly normalizing, ML-CBV evaluation on pure terms must terminate; by the decomposition theorem, the terminus must be a ML-CBV value of type α.

The subject reduction property cannot be extended to full F_ω+control, for essentially the same reasons that type soundness fails for the extension of ML with `callcc` [8, 18]. To see where the problem arises in the present setting, let P be a program of the form $E[callcc_A(M)]$, where E is a ML-CBV evaluation context of the form $E'[\Lambda t{:}\Omega.[]]$, and consider the evaluation step $P \hookrightarrow_{ml-cbv} E[M\ (\Lambda u.\lambda x{:}A.\ abort_u(E[x]))]$. To prove that typing is preserved, it suffices to show that $\Delta, t{:}\Omega, u{:}\Omega; \emptyset \triangleright E[x] : A$ (for some constructor context Δ). But this is strictly stronger than the condition on E given by the decomposition theorem, as remarked above. This observation may be turned into a counterexample to subject reduction by a simple adaptation of the argument given elsewhere by the authors [19], taking advantage of the call-by-value strategy to simulate the "sequential" semantics of the ML `let` construct.

A simple way to avoid the counterexample is to rule out programs with non-trivial evaluation steps lying within the scope of a constructor abstraction. Let F_ω^-+control denote the restriction of F_ω+control in which terms of the form $\Lambda u{:}K.M$ where M is not a ML-CBV value are excluded. This suffices to recover subject reduction in the presence of the control operators.

Theorem 3.6 (Subject Reduction for F_ω^-+control) *If P is a F_ω^-+control program, and $P \hookrightarrow_{ml-cbv} Q$, then Q is a F_ω^-+control program.*

Careful inspection reveals that the CBV and ML-CBV strategies coincide on F_ω^-+control programs. Consequently, termination of ML-CBV evaluation on F_ω^-+control programs follows from termination of CBV evaluation on programs. Moreover, any CPS transform for CBV will suffice as a CPS transform for ML-CBV F_ω^-+control programs.

The fact that the two strategies coincide on F_ω^-+control is unfortunate: we have simply eliminated the parts of the language on which CBV and ML-CBV differ so as to ensure soundness. However, it does not seem possible to give a CPS transform for the pure language under ML-CBV [19]. This would seem to indicate that CPS transforms alone are not sufficient to characterize the difference between CBV and ML-CBV.

3.3.2 ML-CBN Strategy

The ML-like call-by-name strategy is defined as follows:

$$\begin{array}{lcl} V & ::= & \lambda x{:}A.M \mid \Lambda u{:}K.V \\ R & ::= & (\lambda x{:}A.M_1)\, M_2 \mid (\Lambda u{:}K.V)\{A\} \mid \\ & & abort_A(M) \mid callcc_A(M) \\ E & ::= & [] \mid E\, M \mid \Lambda u{:}K.E \mid E\{A\} \end{array}$$

$$\begin{array}{rcl} E[(\lambda x{:}A.M_1)\, M_2] & \hookrightarrow_{ml-cbn} & E[[M_2/x]M_1] \\ E[(\Lambda u{:}K.V)\{A\}] & \hookrightarrow_{ml-cbn} & E[[A/u]V] \\ E[abort_A(M)] & \hookrightarrow_{ml-cbn} & M \\ E[callcc_A(M)] & \hookrightarrow_{ml-cbn} & \\ \multicolumn{3}{l}{E[M\ (\Lambda u{:}\Omega.\lambda x{:}A.\ abort_u(E[x]))]\ (u \notin FTV(A))} \end{array}$$

As with ML-CBV, evaluation may proceed under constructor abstractions resulting in a similarly complex decomposition theorem. Although we can easily show subject reduction and termination for the pure language, this prevents us in much the same way as in the ML-CBV case from obtaining subject reduction in the presence of control operators. We could, as before, simply consider the restricted language F_ω^-+control but there is a better alternative in the call-by-name case.

Careful examination of ML-CBN evaluation contexts reveals that whenever we evaluate under a constructor abstraction in a well-typed, closed term of monomorphic type, that abstraction is ready to be instantiated. I.e., there is a sequence of beta-reduction steps, each of which instantiates one constructor abstraction, which will result in that abstraction being instantiated. Thus, if we alter our evaluation strategy so that we instantiate constructor abstractions whenever possible before evaluating inside them, we will never evaluate inside a constructor abstraction when dealing with a well-typed, closed term of monomorphic type. The new strategy, which we will call ML-CBN′ is defined as follows:

$$\begin{array}{lcl} V & ::= & \lambda x{:}A.M \mid \Lambda u{:}K.V \\ R & ::= & (\lambda x{:}A.M_1)\, M_2 \mid (\Lambda u{:}K.M)\{A\} \mid \\ & & abort_A(M) \mid callcc_A(M) \\ E & ::= & []\{A_1\}\ldots\{A_n\} \mid (E\, M)\{A_1\}\ldots\{A_n\} \mid \Lambda u{:}K.E \end{array}$$

$$\begin{array}{rcl} E[(\lambda x{:}A.M_1)\, M_2] & \hookrightarrow_{ml-cbn'} & E[[M_2/x]M_1] \\ E[(\Lambda u{:}K.M)\{A\}] & \hookrightarrow_{ml-cbn'} & E[[A/u]M] \\ E[abort_A(M)] & \hookrightarrow_{ml-cbn'} & M \\ E[callcc_A(M)] & \hookrightarrow_{ml-cbn'} & \\ \multicolumn{3}{l}{E[M\ (\Lambda u{:}\Omega.\lambda x{:}A.\ abort_u(E[x]))]\ (u \notin FTV(A))} \end{array}$$

Note that the only difference between ML-CBN and ML-CBN′ is that they do constructor abstraction instantiations at different times. Although this effects subject reduction, it does not really alter the meaning of programs. We will make this explicit in the next section where we show that the erasures of these two strategies are the same. Unlike for ML-CBN, subject reduction holds for ML-CBN′. (A decomposition theorem similar to that of the CBV case can be obtained by restricting attention to monomorphic terms.)

Surprisingly, ML-CBN′ and CBN coincide on monomorphically typed terms in the sense that both strategies make precisely the same reductions. (The case of polymorphic terms is not very important since programs must be restricted to be of monomorphic type in order to add in the control operators.[1]) Accordingly, we will not investigate the ML-CBN(′) strategy further, considering it to be the same as the CBN one. (In particular, any CPS transform for CBN will suffice as a CPS transform for ML-CBN programs.)

3.3.3 Relation of ML-like Strategies to ML

The ML-like strategies may be related to their untyped counterparts by way of the following notion of the *erasure* M° of

[1] If we restricted ourselves to the pure language, we could allow programs to have polymorphic type. However, giving a CPS transform for this case is problematic for much the same reasons as in the ML-CBV case.

a term M:

$$\begin{array}{rcl} x^\circ & = & x \\ (\lambda x{:}A.M)^\circ = \lambda x.M^\circ & & (M\,N)^\circ = M^\circ N^\circ \\ (\Lambda u{:}K.M)^\circ = M^\circ & & (M\{A\})^\circ = M^\circ \\ (abort_A(M))^\circ & = & abort(M^\circ) \\ (callcc_A(M))^\circ & = & callcc(M^\circ) \end{array}$$

Erasure is extended to evaluation contexts by defining $[]^\circ = []$.

Theorem 3.7 (Simulation) *Let M be a well-typed closed term.*

1. *if $M \hookrightarrow_{ml-cbv} N$, then $M^\circ \hookrightarrow^{0,1}_{ucbv} N^\circ$.*
2. *if $M \hookrightarrow_{ml-cbn} N$ ($M \hookrightarrow_{ml-cbn'} N$), then $M^\circ \hookrightarrow^{0,1}_{ucbn} N^\circ$.*
3. *if $M^\circ \hookrightarrow_{ucbv} N_1$, then $\exists N_2$ such that $M \hookrightarrow^*_{ml-cbv} N_2$ and $N_2^\circ = N_1$.*
4. *if $M^\circ \hookrightarrow_{ucbn} N_1$, then $\exists N_2$ such that $M \hookrightarrow^*_{ml-cbn} N_2$ ($M \hookrightarrow^*_{ml-cbn'} N_2$) and $N_2^\circ = N_1$.*

Theorem 3.8 (Equivalence) *Let P_1 and P_2 be programs such that $P_1^\circ = P_2^\circ$. Then if $P_1 \hookrightarrow^*_{ml-cbn} Q_1$, and $P_2 \hookrightarrow^*_{ml-cbn'} Q_2$ then $\exists R_1, R_2$ such that $Q_1 \hookrightarrow^*_{ml-cbn} R_1$, $Q_2 \hookrightarrow^*_{ml-cbn'} R_2$, and $R_1^\circ = R_2^\circ$.*

4 Transform Target Languages

In the untyped case, the target language of a CPS transform is a restricted subset of the original language without any control operators. Terms in this restricted subset are said to be in untyped CPS form. This subset has the property that the (untyped) call-by-value and call-by-name evaluation strategies coincide. That is, exactly the same β-reductions occur regardless of which strategy is used. This subset also has the property that it is closed under call-by-value and call-by-name reductions.

4.1 Standard CPS Form

An analogue of untyped CPS form, which we will call *standard CPS form*, exists for the standard strategies. The grammar for this restricted subset of F_ω is as follows:

$$\begin{array}{lcll} Standard\ CPS\ values & W & ::= & x \mid \lambda x{:}A.N \mid \Lambda u{:}K.N \\ Standard\ CPS\ terms & N & ::= & W \mid N\,W \mid N\{A\} \end{array}$$

Note that terms in standard CPS form may not contain $callcc$ or $abort$.

Lemma 4.1 *If N (W_2) is a standard CPS term (standard CPS value) then $[W_1/x]N$ ($[W_2/x]W_1$) is also a standard CPS term (standard CPS value).*

Theorem 4.2 (Standard CPS form properties)

1. *Standard CPS form is closed under CBV and CBN reductions.*
2. *If N_1 is a standard CPS term then $N_1 \hookrightarrow_{cbv} N_2$ iff $N_1 \hookrightarrow_{cbn} N_2$.*
3. *CBV or CBN evaluation of well-typed, closed standard CPS terms terminates in a standard CPS value.*

4.2 ML-CPS Form

ML-CBV and ML-CBN do not coincide on standard CPS terms. To see this, consider the following standard CPS term:

$$(\lambda x{:}(\forall u{:}K.A).\,x)\,(\Lambda u{:}K.(\lambda y{:}A.y)c)$$

ML-CBV will do the innermost redex first while ML-CBN will do the outermost one first. An analogue of untyped CPS form for the ML-like strategies, which we call *ML-CPS form* is defined as follows:

$$\begin{array}{lcll} \textit{ML-CPS values} & X & ::= & x \mid \lambda x{:}A.O \mid \Lambda u{:}K.X \\ \textit{ML-CPS terms} & O & ::= & X \mid O\,X \mid \Lambda u{:}K.O \mid O\{A\} \end{array}$$

As in the standard CPS form case, it can be shown that ML-CBV and ML-CBN coincide on ML-CPS terms and that ML-CPS is closed under ML-CBV and ML-CBN reductions. Note that if O is an ML-CPS term, then O° is an untyped CPS term, and if X is an ML-CPS value, then X° is an untyped CPS value.

It is easy to see that every ML-CPS term is a standard CPS term, and that every ML-CPS value is a standard CPS value. A little checking shows that ML-CPS form is closed under CBV and CBN reductions so we have that CBV and CBN coincide on ML-CPS terms as well.

Theorem 4.3 (ML-CPS form properties)

1. *ML-CPS form is closed under CBV, CBN, ML-CBV, and ML-CBN reductions.*
2. *If O_1 is a ML-CPS term then $O_1 \hookrightarrow_{ml-cbv} O_2$ iff $O_1 \hookrightarrow_{ml-cbn} O_2$.*
3. *If O_1 is a ML-CPS term then $O_1 \hookrightarrow_{cbv} O_2$ iff $O_1 \hookrightarrow_{cbn} O_2$.*
4. *CBV, CBN, ML-CBV, or ML-CBN evaluation of well-typed, closed ML-CPS terms terminates in a ML-CPS value.*

4.3 Strict CPS Form

Neither of the pairs CBV/ML-CBV nor CBN/ML-CBN coincide on terms in ML-CPS form. To see this, consider the ML-CPS term $\Lambda u{:}K.(\lambda x{:}A.x)c$. This term is irreducible under CBV and CBN (since constructor abstractions are values), but is reducible under both ML-CBV and ML-CBN (since evaluation proceeds under constructor abstraction). By further restricting ML-CPS (in particular, by banning all non-value constructor abstractions), we may obtain a subset of ML-CPS called *strict CPS form*, on which all four strategies coincide:

Strict CPS values	Y	$::=$	$x \mid \lambda x{:}A.Q \mid \Lambda u{:}K.Y$
Strict CPS terms	Q	$::=$	$Y \mid Q\,Y \mid Q\{A\}$

Theorem 4.4 (Strict CPS form properties)

1. *Strict CPS form is closed under CBV, CBN, ML-CBV, and ML-CBN reductions.*
2. *If N_1 is a strict CPS term then if $N_1 \hookrightarrow N_2$ under one of CBV, CBN, ML-CBV, or ML-CBN, then it does so under all of them.*
3. *CBV, CBN, ML-CBV, or ML-CBN evaluation of well-typed, closed strict CPS terms terminates in a strict CPS value.*

4.4 Relaxed ML-CPS Form

As we shall see in the next section, the CPS conversion algorithms for the standard strategies yield terms in strict CPS form, and consequently any of the four evaluation methods may be used on the converted terms. As was explained in section 3, these algorithms can be used as CPS conversion algorithms for the ML-like strategies on certain restricted subsets of F_ω+$\mathsf{control}$.

However, it is enlightening to consider alternate algorithms specifically tailored to the ML-like strategies. As we shall see below, these transforms yield terms of the form $k\,(x\,\{A\})$, where k and x are variables, which is not in ML-CPS form. In *relaxed ML-CPS form* such applications are allowed, reflecting the philosophy that constructor applications are insignificant at run time. As with ML-CPS, erased relaxed ML-CPS terms (values) are untyped CPS terms (values). Relaxed ML-CPS form is defined as follows:

Relaxed ML-CPS values	Z	$::=$	$x \mid \lambda x{:}A.S \mid \Lambda u{:}K.Z \mid Z\{A\}$
Relaxed ML-CPS terms	S	$::=$	$Z \mid S\,Z \mid \Lambda u{:}K.S \mid S\{A\}$

The set of terms in relaxed ML-CPS form is closed under ML-CBV and ML-CBN reduction. However, ML-CBV and ML-CBN do not coincide on this subset because of terms such as $(\lambda x{:}A.x)((\Lambda u{:}K.Z)A)$ in which there is a constructor application in the argument position that would be reduced under ML-CBV, but not under ML-CBN. Their erasures do coincide, however, in the same sense that ML-CBN and ML-CBN$'$ coincided.

Theorem 4.5 (Equivalence) *Let P_1 and P_2 be relaxed ML-CPS programs such that $P_1^\circ = P_2^\circ$. Then if $P_1 \hookrightarrow^*_{ml-cbv} Q_1$, and $P_2 \hookrightarrow^*_{ml-cbn} Q_2$ then $\exists R_1, R_2$ such that $Q_1 \hookrightarrow^*_{ml-cbv} R_1$, $Q_2 \hookrightarrow^*_{ml-cbn} R_2$, and $R_1^\circ = R_2^\circ$.*

If we assume that constructor β-reductions do no work and have no side effects then ML-CBV and ML-CBN produce the same results on programs in this subset. This is a reasonable assumption for ML-like strategies because the normal implementation for such strategies is to erase then apply the untyped strategy.

5 Conversion to Continuation-Passing Style

In this section, we consider the conversion of terms of F_ω+$\mathsf{control}$ into continuation-passing style for each of the evaluation strategies. We present CPS transforms for the two standard strategies for the full F_ω+$\mathsf{control}$ language. As discussed in section 3, the CBV CPS transform can also be used as a ML-CBV CPS transform on the restricted subset F_ω^-+$\mathsf{control}$ and the CBN CPS transform can be used as a ML-CBN$'$ transform on monomorphic terms. Producing ML-like transforms for larger subsets of F_ω+$\mathsf{control}$ than these is problematic.

Although the standard transforms can be used to transform terms under the ML-like strategies, they are somewhat unsatisfactory in that they do not fully capture the essence of the ML-like strategies, namely that constructor applications are not significant computation steps. We consider two more satisfactory alternative ML-like CPS transforms which do embody this fact at the cost of being limited to terms of the F_ω^-+$\mathsf{control}$ language.

5.1 Transformation of Constructors

There are four constructor transformations, corresponding to the four evaluation strategies introduced in section 3. The transformations differ only in the treatment of the function types (call-by-name and call-by-value variants) and in the treatment of quantified types (standard and ML-like variants).

Definition 5.1 (Constructor Transforms)

$$|A| = (A^*{\to}\alpha){\to}\alpha$$

$$\alpha^* = \alpha$$
$$u^* = u$$

Function types, call-by-value:
$(A_1{\to}A_2)^* = A_1^*{\to}|A_2|$
Function types, call-by-name:
$(A_1{\to}A_2)^* = |A_1|{\to}|A_2|$

Quantified types, standard interpretation:
$(\forall u{:}K.A)^* = \forall u{:}K.|A|$
Quantified types, ML-like interpretation:
$(\forall u{:}K.A)^* = \forall u{:}K.A^*$

$$(\lambda u{:}K.A)^* = \lambda u{:}K.A^*$$
$$(A_1\, A_2)^* = A_1^* A_2^*$$

The constructor transforms are extended to contexts Γ by defining $\Gamma^*(x) = A^*$ and $|\Gamma|(x) = |A|$ whenever $\Gamma(x) = A$. The following properties apply to all four variants of the constructor transformation.

Theorem 5.2 (Constructor Well-formedness Preservation)

1. *If* $\mathsf{F}_\omega \vdash \Delta \triangleright A : K$*, then* $\mathsf{F}_\omega \vdash \Delta \triangleright A^* : K$*.*
2. *If* $\mathsf{F}_\omega \vdash \Delta \triangleright A : \Omega$*, then* $\mathsf{F}_\omega \vdash \Delta \triangleright |A| : \Omega$*.*

Theorem 5.3 (Constructor Equality Preservation)

1. *If* $\mathsf{F}_\omega \vdash \Delta \triangleright A_1 = A_2 : K$*, then* $\mathsf{F}_\omega \vdash \Delta \triangleright A_1^* = A_2^* : K$*.*
2. *If* $\mathsf{F}_\omega \vdash \Delta \triangleright A_1 = A_2 : \Omega$*, then* $\mathsf{F}_\omega \vdash \Delta \triangleright |A_1| = |A_2| : \Omega$*.*

Theorem 5.4 (Compositionality)
$([A_1/u]A_2)^* = [A_1^*/u]A_2^*$.

5.2 Notation

In addition to the transforms for constructors given above, each CPS transform has a transform for values, $(-)^*$, and a transform for terms, $|-|$. Keep in mind that the set of values varies from strategy to strategy. As a notational convenience, we drop the identifying subscripts on transform operators when refering to the current transform being defined. As a proof tool, we will need to introduce an optimized version of the transform we are defining. We will denote the optimized value transform using $(-)^\star$ and the optimized term transform relative to continuation Y by $|-|_Y$ to prevent confusion with the non-optimized transform.

The CPS transforms are defined by induction on typing derivations in F_ω+$\mathsf{control}$, yielding terms in a suitable CPS form. The typing rules of F_ω+$\mathsf{control}$ are "almost" syntax-directed — any two typing derivations for a given term and context differ only in the use of the type equality rule. Consequently, since our transforms ignore the type equality rule, they are coherent in the sense that if F_ω+$\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$, and F_ω+$\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A'$, then the transforms determined by each of these typing derivations are equivalent up to constructor equality. Since the evaluation rules for F_ω+$\mathsf{control}$ are independent of constructors, we are justified in ignoring this distinction, and simply write $|M|$ for the CPS transform obtained by a canonical choice of typing derivation for M.

New variables introduced by the transform are assumed to be chosen so as to avoid capture. In cases where more than one clause of the transform applies (this only occurs in the optimized versions), the first one listed is to be chosen. Where clear, we have omitted subderivations and the details of how recusion is done on the type derivation of the term.

5.3 Standard CPS Transforms

5.3.1 Call-by-Value

The definition of the CBV CPS transform is given in Table 1.

Theorem 5.5 (CBV CPS Typing) *If* F_ω+$\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$*, then* $|M|$ *exists and is a strict CPS value such that* $\mathsf{F}_\omega \vdash \Delta; \Gamma^* \triangleright |M| : |A|$*. If* M *is a CBV value, then* M^* *exists and is a strict CPS value such that* $\mathsf{F}_\omega \vdash \Delta; \Gamma^* \triangleright M^* : A^*$*.*

In the following theorem we write $|P|_V$ for the call-by-value CPS transform of P applied to the continuation V with so-called "administrative redices" (in the sense of Plotkin [36]) eliminated.[2]

Theorem 5.6 (CBV Simulation) *If* P *is a program and* $P \hookrightarrow_{cbv} Q$*, then* $|P|_{\lambda x:\alpha.x} \hookrightarrow_\beta^* |Q|_{\lambda x:\alpha.x}$*. Moreover, each* β*-step induces at least one* β*-step on the converted form.*

Theorem 5.7 *For any program* P*,*

1. *There exists a unique CBV value* V *such that* $P \hookrightarrow_{cbv}^* V$*.*
2. *If* $P \hookrightarrow_{cbv}^* V$ *then* $|P|\,(\lambda x{:}\alpha.x) \hookrightarrow_\beta^* V'$ *where* V' *is such that* $V^* \hookrightarrow_\beta^* V'$*.*

[2] The proofs of this and subsequent simulation theorems are interesting (particularly in the presence of control operators), but rather involved. A more complete account will be given in the full paper.

$$
\begin{array}{rcl}
|\Delta;\Gamma \triangleright V : A| & = & \lambda k{:}A^*{\to}\alpha.\, k\, V^* \\[1ex]
|\Delta;\Gamma \triangleright M_1\, M_2 : A| & = & \lambda k{:}A^*{\to}\alpha.\, |M_1|\, (\lambda x_1{:}(A{\to}A_2)^*.|M_2|\, (\lambda x_2{:}A_2^*.x_1\, x_2\, k)) \\
& & \quad \mathit{where}\ \Delta;\Gamma \triangleright M_1 : A_2{\to}A\ \mathit{and}\ \Delta;\Gamma \triangleright M_2 : A_2 \\[1ex]
|\Delta;\Gamma \triangleright M\,\{A_1\} : [A_1/u]A_2| & = & \lambda k{:}([A_1/u]A_2)^*{\to}\alpha.\, |M|\, (\lambda x{:}(\forall u{:}K_1.A_2)^*.x\{A_1^*\}\, k) \\[1ex]
|\Delta;\Gamma \triangleright \mathit{abort}_A(M) : A| & = & \lambda k{:}\alpha^*{\to}\alpha.\, |M|\, (\lambda m{:}\alpha^*.m) \\[1ex]
|\Delta;\Gamma \triangleright \mathit{callcc}_A(M) : A| & = & \lambda k{:}A^*{\to}\alpha.\, |M|\, (\lambda m{:}((\forall u{:}\Omega.A{\to}u){\to}A)^*. \\
& & \quad m\, (\Lambda u{:}\Omega.\lambda l{:}(A{\to}u)^*{\to}\alpha.l(\lambda x{:}A^*.\lambda k'{:}u^*{\to}\alpha.k\, x))\, k) \\[1ex]
|\Delta;\Gamma \triangleright M : A'| & = & |M|,\ \mathit{where}\ \Delta;\Gamma \triangleright M : A\ \mathit{and}\ \Delta \triangleright A = A' : \Omega \\[1ex]
(\Delta;\Gamma \triangleright x : A)^* & = & x \\
(\Delta;\Gamma \triangleright \lambda x{:}A.M : A{\to}A')^* & = & \lambda x{:}A^*.|M| \\
(\Delta;\Gamma \triangleright \Lambda u{:}K.M : \forall u{:}K.A)^* & = & \Lambda u{:}K.|M| \\[1ex]
(\Delta;\Gamma \triangleright V : A')^* & = & V^*,\ \mathit{where}\ \Delta;\Gamma \triangleright V : A\ \mathit{and}\ \Delta \triangleright A = A' : \Omega
\end{array}
$$

Table 1: CBV CPS Transform for F_ω+$\mathsf{control}$

$$
\begin{array}{rcl}
|\Delta;\Gamma \triangleright x : A| & = & x \\[1ex]
|\Delta;\Gamma \triangleright M_1\, M_2 : A| & = & \lambda k{:}A^*{\to}\alpha.|M_1|\, (\lambda x_1{:}(A_1{\to}A_2)^*.x_1\, |M_2|\, k) \\
& & \mathit{where}\ \Delta;\Gamma \triangleright M_1 : A_2{\to}A\ \mathit{and}\ \Delta;\Gamma \triangleright M_2 : A_2 \\[1ex]
|\Delta;\Gamma \triangleright \mathit{callcc}_A(M) : A| & = & \lambda k{:}A^*{\to}\alpha.\, |M|\, (\lambda m{:}((\forall u{:}\Omega.A{\to}u){\to}A)^*.m\, Y\, k),\ \mathit{where} \\
Y & = & \lambda l{:}(\forall u{:}\Omega.A{\to}u)^*{\to}\alpha.l\, (\Lambda u{:}\Omega.\lambda l{:}(A{\to}u)^*{\to}\alpha.l\, (\lambda x{:}|A|.\lambda k'{:}u^*{\to}\alpha.x\, k)) \\[1ex]
(\Delta;\Gamma \triangleright \lambda x{:}A.M : A{\to}A')^* & = & \lambda x{:}|A|.|M|
\end{array}
$$

Table 2: CBN CPS Transform for F_ω+$\mathsf{control}$ (Selected Clauses)

5.3.2 Call-by-Name

The standard call-by-name semantics also admits a conversion into CPS sharing essentially the same properties as are enjoyed by the standard call-by-value transform. We have only to switch to the call-by-name variant of the constructor transform and modify the CBV transform by replacing the variable, application, and $callcc$ clauses by the clauses in Table 2. Note that in call-by-name variables are no longer considered values.

Theorem 5.8 (CBN CPS Typing) *If* $\mathsf{F}_\omega+\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$*, then* $|M|$ *exists and is a strict CPS value such that* $\mathsf{F}_\omega \vdash \Delta; |\Gamma| \triangleright |M| : |A|$*. If* M *is a CBN value, then* M^* *exists and is a strict CPS value such that* $\mathsf{F}_\omega \vdash \Delta; |\Gamma| \triangleright M^* : A^*$.

Theorem 5.9 *Let* P *be a program.*

1. *There exists a unique CBN value* V *such that* $P \hookrightarrow^*_{cbn} V$.
2. *If* $P \hookrightarrow^*_{cbn} V$ *then* $|P|\,(\lambda x{:}\alpha.x) \hookrightarrow^*_\beta V'$ *where* V' *is such that* $V^* \hookrightarrow^*_\beta V'$.

5.4 Alternative ML-like CPS Transforms

The constructor transforms for the standard strategies are based on the definition $(\forall u{:}K.A)^* = \forall u{:}K.|A|$, expressing the idea that constructor applications are "serious" computations (in the sense of Reynolds [38]), and hence require a continuation. However, for terms of $\mathsf{F}^-_\omega+\mathsf{control}$ there are no such non-trivial computations (the continuation is always immediately invoked with a value), and hence we may contemplate eliminating the continuation argument entirely. This would make constructor application a trivial computation step, resulting in a far more ML-like transform. This suggests the alternative definition $(\forall u{:}K.A)^* = \forall u{:}K.A^*$ in the constructor transformation, and a corresponding change to the constructor abstraction and construction application transforms for terms.

5.4.1 Call-by-Value

The definition of the alternative ML-CBV CPS transform is the same as for the standard CBV CPS transform, with the following differences. First, we employ the ML-like definition of the $(-)^*$ transform on constructors; in particular, $(\forall u{:}K.A)^* = \forall u{:}K.A^*$. Second, we take the clauses given in Table 3 for constructor abstraction and application and for $callcc$, leaving the remainder as for the standard CBV strategy.

Theorem 5.10 (ML-CBV CPS Typing) *If* $\mathsf{F}^-_\omega+\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$*, then* $|M|$ *exists and is a relaxed CPS value such that* $\mathsf{F}^-_\omega \vdash \Delta; \Gamma^* \triangleright |M| : |A|$.

A careful inspection of this transform reveals that it is essentially a typed version of the usual untyped call-by-value CPS transform. Its correctness follows from this plus the known correctness of the untyped CBV CPS transform.

Theorem 5.11 (ML-CBV Simulation) *If* $\mathsf{F}^-_\omega+\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$*, then* $|M|^\circ \hookrightarrow^*_\eta |M^\circ|_{ucbv}$.

5.4.2 Call-by-Name (ML-CBN′)

An alternative ML-CBN′ CPS transform can be obtained in a similar manner. As before, the transform is based on the standard strategy CPS transform (CBN) with some differences. The differences include using the ML-like version of the constructor transform and the two alternative constructor application and abstraction transform rules from the previous section. The only other difference is that we take the following term as the definition of $|\Delta; \Gamma \triangleright callcc_A(M) : A|$:

$$\lambda k{:}A^*{\to}\alpha.\ |M|\,(\lambda m{:}((\forall u{:}\Omega.A{\to}u){\to}A)^*.m\,Y\,k),$$

where

$$Y = \lambda l{:}(\forall u{:}\Omega.A{\to}u)^*{\to}\alpha.l\ (\Lambda u{:}\Omega.\lambda x{:}|A|.\lambda k'{:}u^*{\to}\alpha.x\ k)$$

Theorem 5.12 (ML-CBN′ CPS Typing) *If* $\mathsf{F}^-_\omega+\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$*, then* $|M|$ *exists and is a relaxed CPS value such that* $\mathsf{F}^-_\omega \vdash \Delta; |\Gamma| \triangleright |M| : |A|$.

Theorem 5.13 (ML-CBN′ Simulation) *If* $\mathsf{F}^-_\omega+\mathsf{control} \vdash \Delta; \Gamma \triangleright M : A$*, then* $|M|^\circ \hookrightarrow^*_\eta |M^\circ|_{ucbn}$.

6 Conclusion

We have presented a systematic study of the typing properties of CPS conversion for $\mathsf{F}_\omega+\mathsf{control}$ under four different semantic interpretations. The standard strategies — both call-by-value and call-by-name variants — validate subject reduction, are terminating, and admit faithful, type-preserving transformations into continuation-passing style. We conclude that the standard strategies are semantically unproblematic, at least when viewed from the point of view of compilation and typing. These strategies have the significant advantage of being extensible to a more sophisticated set of primitive operations, in particular, those that make non-trivial use of type information at run time.

On the other hand, the ML-like call-by-value strategy is problematic — $\mathsf{F}_\omega+\mathsf{control}$, when evaluated under this strategy, fails to be sound, and hence cannot admit a type-preserving, faithful transformation into CPS. Such a transformation is possible for the fragment $\mathsf{F}^-_\omega+\mathsf{control}$ in which constructor abstractions are limited to values, which is consistent with a similar restriction in the untyped case [19].

The ML-like call-by-name strategy (ML-CBN′) is unproblematic but uninteresting because it is (almost) identical to the standard call-by-name strategy. It only differs on

$$|\Delta;\Gamma \rhd M\{A\} : [A/u]B| \quad = \quad \lambda k{:}([A/u]B)^{*}{\rightarrow}\alpha.\,|M|\,(\lambda m{:}(\forall u{:}K.B)^{*}.k\,(m\,\{A^{*}\}))$$

$$|\Delta;\Gamma \rhd callcc_{A}(M) : A| \quad = \quad \lambda k{:}A^{*}{\rightarrow}\alpha.\,|M|\,(\lambda m{:}((\forall u{:}\Omega.A{\rightarrow}u){\rightarrow}A)^{*}.\; m\,(\Lambda u{:}\Omega.\lambda x{:}A^{*}.\lambda k'{:}u^{*}{\rightarrow}\alpha.k\,x)\,k)$$

$$(\Delta;\Gamma \rhd \Lambda u{:}K.V : \forall u{:}K.A)^{*} \quad = \quad \Lambda u{:}K.V^{*}$$

Table 3: ML-CBV CPS Transform for F_{ω}+$\mathsf{control}$ (Selected Clauses)

polymorphic terms. The difference is not very interesting because it conveys no extra power or expressiveness over the standard call-by-name strategy.

We are grateful to Olivier Danvy, Andrzej Filinski, and Timothy Griffin for their comments and suggestions.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*. ACM, January 1989.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, TX, January 1989.

[4] Luca Cardelli. Typeful programming. Technical Report 45, DEC SRC, 1989.

[5] Eric C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[6] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[7] Luis Manuel Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, Edinburgh University, 1985.

[8] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[9] Matthias Felleisen. *The Calculi of λ_v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, Bloomington, IN, 1987.

[10] Matthias Felleisen and Daniel Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III*. North-Holland, 1986.

[11] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *First Symposium on Logic in Computer Science*. IEEE, June 1986.

[12] Andrzej Filinski. Linear continuations. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 27–38, Albuquerque, NM, January 1992. ACM SIGPLAN/SIGACT.

[13] Jean Gallier. On girard's "candidats de reductibilité". In P. Odifreddi, editor, *Logic and Computation*, volume 31 of *The APIC Series*, pages 123–203. Academic Press, 1990.

[14] Jean-Yves Girard. *Interprètation Fonctionelle et Élimination des Coupures dans l'Arithmètique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[15] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[16] Timothy Griffin. A formulae-as-types notion of control. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990. ACM, ACM.

[17] Timothy G. Griffin. Logical interpretations and computational simulations. Tech. memo., AT&T Bell Laboratories, 1992. in preparation.

[18] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. Revised and expanded version of [8]. To appear, *Journal of Functional Programming*.

[19] Robert Harper and Mark Lillibridge. Polymorphic type assignment and cps conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 13–22, Stanford, CA 94305, June 1992. Department of Computer Science, Stanford University. Published as technical report STAN–CS–92–1426.

[20] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, ?(?):?–?, ? 1992? (To appear. See also [30].).

[21] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines from continuations. *Journal of Computer Languages*, 11:143–153, 1986.

[22] Gérard Huet, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[23] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J.Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *Proc. SIGPLAN Symposium on Compiler Construction*, pages 219–233. ACM SIGPLAN, 1986.

[24] Xavier leroy. Polymorphism by name. In *Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[25] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 291–302, Orlando, FL, January 1991. ACM SIGACT/SIGPLAN.

[26] Zhaolui Luo, Robert Pollack, and Paul Taylor. How to use lego: A preliminary user's manual. Technical Report LFCS-TN-27, Laboratory for the Foundations of Computer Science, Edinburgh University, October 1989.

[27] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda calculi (summary). In Rohit Parikh, editor, *Logics of Programs*, volume 224 of *Lecture Notes in Computer Science*, pages 219–224. Springer-Verlag, 1985.

[28] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[29] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[30] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[31] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988. (Reprinted in [22], pp. 153–194.).

[32] Chetan Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, Ithaca, NY, August 1990.

[33] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Sixteenth ACM Symposium on Principles of Programming Languages*, 1989.

[34] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 199? To appear. Preliminary version available as Technical Report CMU–CS–92–105, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1992.

[35] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.

[36] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[37] John Reppy. First-class synchronous operations in standard ML. Technical Report TR 89-1068, Computer Science Department, Cornell University, Ithaca, NY, December 1989.

[38] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972. ACM.

[39] John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[40] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier Science Publishers B. V., 1983.

[41] Guy L. Steele, Jr. RABBIT: A compiler for SCHEME. Technical Report Memo 474, MIT AI Laboratory, 1978.

[42] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[43] Christopher Strachey and Christopher Wadsworth. A mathematical semantics for handling full jumps. Technical Report Technical Monograph PRG–11, Oxford University Computing Laboratory, 1974.

[44] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988. Available as Edinburgh University Laboratory for Foundations of Computer Science Technical Report ECS–LFCS–88–54.

[45] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[46] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91–160, Department of Computer Science, Rice University, July 1991. To appear, *Information and Computation*.

A Rules for F_ω+$\mathsf{control}$

Definition A.1 (Constructor Context Formation Rules)

$$\triangleright \emptyset \qquad \text{(C-EMPTY)}$$

$$\frac{\triangleright \Delta \quad u \notin \mathrm{dom}(\Delta)}{\triangleright \Delta, u{:}K} \qquad \text{(C-EXTEND)}$$

Definition A.2 (Term Context Formation Rules)

$$\frac{\triangleright \Delta}{\Delta \triangleright \emptyset} \qquad \text{(T-EMPTY)}$$

$$\frac{\Delta \triangleright \Gamma \quad \Delta \triangleright A : \Omega \quad x \notin \mathrm{dom}(\Gamma)}{\Delta \triangleright \Gamma, x{:}A} \qquad \text{(T-EXTEND)}$$

Definition A.3 (Constructor Formation Rules)

$$\frac{\triangleright \Delta}{\Delta \triangleright \alpha : \Omega} \qquad \text{(C-ANS)}$$

$$\frac{\triangleright \Delta}{\Delta \triangleright u : \Delta(u)} \qquad \text{(C-VAR)}$$

$$\frac{\Delta \triangleright A_1 : \Omega \quad \Delta \triangleright A_2 : \Omega}{\Delta \triangleright A_1 \to A_2 : \Omega} \qquad \text{(C-ARR)}$$

$$\frac{\Delta, u{:}K \triangleright A : \Omega}{\Delta \triangleright \forall u{:}K.A : \Omega} \qquad \text{(C-ALL)}$$

$$\frac{\Delta, u{:}K_1 \triangleright A : K_2}{\Delta \triangleright \lambda u{:}K_1.A : K_1 \Rightarrow K_2} \qquad \text{(C-ABS)}$$

$$\frac{\Delta \triangleright A_1 : K_2 \Rightarrow K \quad \Delta \triangleright A_2 : K_2}{\Delta \triangleright A_1\, A_2 : K} \qquad \text{(C-APP)}$$

Definition A.4 (Constructor Equality Rules)

$$\frac{\Delta \triangleright A : K}{\Delta \triangleright A = A : K} \quad \text{(REFL)}$$

$$\frac{\Delta \triangleright A_1 = A_2 : K}{\Delta \triangleright A_2 = A_1 : K} \quad \text{(SYMM)}$$

$$\frac{\Delta \triangleright A_1 = A_2 : K \quad \Delta \triangleright A_2 = A_3 : K}{\Delta \triangleright A_1 = A_3 : K} \quad \text{(TRANS)}$$

$$\frac{\Delta \triangleright A_1 = A_1' : \Omega \quad \Delta \triangleright A_2 = A_2' : \Omega}{\Delta \triangleright A_1 \to A_2 = A_1' \to A_2' : \Omega} \quad \text{(C-ARR-EQ)}$$

$$\frac{\Delta, u{:}K \triangleright A = A' : \Omega}{\Delta \triangleright \forall u{:}K.A = \forall u{:}K.A' : \Omega} \quad \text{(C-ALL-EQ)}$$

$$\frac{\Delta, u{:}K_1 \triangleright A = A' : K_2}{\Delta \triangleright \lambda u{:}K_1.A = \lambda u{:}K_1.A' : K_1 \Rightarrow K_2} \quad \text{(C-ABS-EQ)}$$

$$\frac{\Delta \triangleright A_1 = A_1' : K_2 \Rightarrow K \quad \Delta \triangleright A_2 = A_2' : K_2}{\Delta \triangleright A_1\, A_2 = A_1'\, A_2' : K} \quad \text{(C-APP-EQ)}$$

$$\frac{\Delta, u{:}K_1 \triangleright A_2 : K_2 \quad \Delta \triangleright A_1 : K_1}{\Delta \triangleright (\lambda u{:}K_1.A_2)\, A_1 = [A_1/u]A_2 : K_2} \quad \text{(C-BETA)}$$

$$\frac{\Delta \triangleright A : K_1 \Rightarrow K_2 \quad u \notin \mathrm{dom}(\Delta)}{\Delta \triangleright \lambda u{:}K_1.A\, u = A : K_1 \Rightarrow K_2} \quad \text{(C-ETA)}$$

Definition A.5 (Term Formation Rules)

$$\frac{\Delta \triangleright \Gamma}{\Delta; \Gamma \triangleright x : \Gamma(x)} \quad \text{(T-VAR)}$$

$$\frac{\Delta; \Gamma, x{:}A_1 \triangleright M : A_2}{\Delta; \Gamma \triangleright \lambda x{:}A.M : A_1 \to A_2} \quad \text{(T-ABS)}$$

$$\frac{\Delta; \Gamma \triangleright M_1 : A_2 \to A \quad \Delta; \Gamma \triangleright M_2 : A_2}{\Delta; \Gamma \triangleright M_1\, M_2 : A} \quad \text{(T-APP)}$$

$$\frac{\Delta, u{:}K; \Gamma \triangleright M : A \quad \Delta \triangleright \Gamma}{\Delta; \Gamma \triangleright \Lambda u{:}K.M : \forall u{:}K.A} \quad \text{(T-CABS)}$$

$$\frac{\Delta; \Gamma \triangleright M : \forall u{:}K.A' \quad \Delta \triangleright A : K}{\Delta; \Gamma \triangleright M\{A\} : [A/u]A'} \quad \text{(T-CAPP)}$$

$$\frac{\Delta \triangleright A : \Omega \qquad \Delta; \Gamma \triangleright M : \alpha}{\Delta; \Gamma \triangleright abort_A(M) : A} \quad \text{(T-ABORT)}$$

$$\frac{\Delta; \Gamma \triangleright M : (\forall u{:}\Omega.A \to u) \to A \qquad u \notin FTV(A)}{\Delta; \Gamma \triangleright callcc_A(M) : A} \quad \text{(T-CALLCC)}$$

$$\frac{\Delta; \Gamma \triangleright M : A \quad \Delta \triangleright A = A' : \Omega}{\Delta; \Gamma \triangleright M : A'} \quad \text{(T-EQ)}$$

Lemma A.6 (Properties of F_ω+control typing)

1. *if* F_ω+control $\vdash \Delta \triangleright \Gamma$ *then* F_ω+control $\vdash \triangleright \Delta$
2. *if* F_ω+control $\vdash \Delta \triangleright A : K$ *then* F_ω+control $\vdash \triangleright \Delta$
3. *if* F_ω+control $\vdash \Delta \triangleright A_1 = A_2 : K$ *then* F_ω+control $\vdash \Delta \triangleright A_1 : K$ *and* F_ω+control $\vdash \Delta \triangleright A_2 : K$
4. *if* F_ω+control $\vdash \Delta; \Gamma \triangleright M : A$ *then* F_ω+control $\vdash \Delta \triangleright \Gamma$ *and* F_ω+control $\vdash \Delta \triangleright A : \Omega$