Typing First-Class Continuations in ML*

Robert Harper† Bruce F. Duba‡ David MacQueen§

November 4, 1993

Abstract

An extension of ML with continuation primitives similar to those found in Scheme is considered. A number of alternative type systems are discussed, and several programming examples are given. A continuation-based operational semantics is defined for a small, purely functional, language, and the soundness of the Damas-Milner polymorphic type assignment system with respect to this semantics is proved. The full Damas-Milner type system is shown to be unsound in the presence of first-class continuations. Restrictions on polymorphism similar to those introduced in connection with reference types are shown to suffice for soundness.

1 Introduction

First-class continuations are a simple and natural way to provide access to the flow of evaluation in functional languages. The ability to seize the "current continuation" (control state of the evaluator) provides a simple and natural basis for defining numerous higher-level constructs such as coroutines [22], exceptions [41], and logic variables [8, 19], for supporting multiple threads of control [6, 20, 28, 40], for providing asynchronous signal handlers [29], and for implementing non-blind backtracking [15] and dynamic barriers such as unwind-protect [21]. Tractable logics for reasoning about program equivalence in the presence of first-class continuations in an untyped setting have been developed [11, 12, 37]. Recent studies of continuations have addressed the question of their typing in a restricted setting [13, 14, 16] and their impact on full abstraction results [32].

The subject of this paper is the extension of Standard ML with primitives for first-class continuations similar to those found in Scheme. The two new primitives are `callcc`, for *call with current continuation*, which takes a function as argument and calls it with the current continuation, and `throw`, which takes a continuation and a value and passes the value to that continuation. In Section 2 we give an informal presentation of the extension of ML with continuation primitives, and illustrate their use in programming examples. We also discuss the role of continuations in the implementation of Standard ML of New Jersey, and some problems that they raised. In Section 3 we consider the semantics of type assignment for the functional core of ML, extended with continuation-passing primitives. A simple and natural-seeming signature for the continuation primitives turns out to be unsound. We consider two restrictions of the type system for which a soundness theorem may be obtained.

We are grateful to Andrew Appel, Stephen Brookes, Matthias Felleisen, Andrzej Filinski, Timothy Griffin, John Reppy, Didier Rémy, Olin Shivers, Mads Tofte, and Andrew Wright for their comments and suggestions.

*This is a pre-print of an article to appear in the *Journal of Functional Programming*, and should be cited as such.

†School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213. This research was sponsored in part by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404, and in part by 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.

‡Department of Computer Science, Rice University, Houston, TX 77251. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under ARPA order 6253.

§AT&T Bell Laboratories, Murray Hill, NJ 07974

2 Adding First-Class Continuations to ML

First-class continuations are an abstraction that evolved from various nonstandard control structures such as Landin's J-operator [23], Reynold's `escape` [31], label variables in Gedanken [30] and PAL [7], and from the semantic analyses of general control structures, including jumps [34]. Scheme [36] originally introduced a binding construct `(catch k body)` that captured its own expression continuation and bound it to the variable `k`, with the expression `body` as the scope of the binding. The continuation represents the "rest of the computation," and behaves as a function that takes the value of the expression as its argument and yields the final result of the evaluation of the remainder of the program. In a typical implementation the final result is passed to the interactive top-level, which prints the result, and continues by evaluating the next expression.

Around 1982 the special-form `catch` was replaced by `call-with-current-continuation` or `call/cc` for short [2]. The act of capturing the current continuation did not require a special variable binding form, but could be performed by a primitive operator whose argument was a function that would be applied to the captured continuation. Therefore, `(catch x body)` becomes `(call/cc (lambda (x) body))` in Scheme. This is an example of the well-known technique of replacing a special variable binding form with an operation acting on a function, so that variable binding is handled solely by lambda abstraction.

In an untyped language there is not much to choose between the functional and binding forms of continuation-capturing construct. However, in the context of an ML-like type system, the two differ substantially. To understand the distinction, it is helpful to consider the interaction between typing and the invocation of a captured continuation. There are two main points. First, continuations arise in a program only by capturing the evaluation context of some expression: there are no expression forms denoting continuations. Therefore a continuation expects values of the type of the expression whose evaluation context the continuation represents. Second, the invocation of a captured continuation discards the current evaluation context, passing a value to the captured, instead of the current, continuation. Although the passed value must be consistent with the argument type of the continuation, the result type is unconstrained since invocations of continuations do not return to the evaluation context. (For similar reasons the exception-raising construct of Standard ML has arbitrary result type.)

For example, if `k` is bound to a continuation expecting an integer value, we may invoke `k` in several incompatible type contexts, as in the following expression[1]

```
1 + callcc(fn k => hd(if b then [ (k 3) + 1 ] else 5 :: (k 4)))
```

Here `k` is invoked in two contexts, one expecting an integer, the other expecting an integer list. Since continuation invocations never return, it makes sense to regard this as a well-typed expression (of type `int`).

The incorporation of continuation primitives in ML involves making two related decisions, namely the continuation-capturing construct and the continuation-invoking construct. Since ML is a typed language, continuations should be values of some type, say τ `cont`, the type of continuations expecting values of type τ. The continuation-capturing constructs may then be given typing rules as follows. The functional form, written `callcc` in keeping with the ML lexical conventions, may be assigned any type of the form $(\tau\ \texttt{cont} \rightarrow \tau) \rightarrow \tau$ since the body may either invoke the passed continuation, or else return normally. Written polymorphically, we have the typing

$$\texttt{callcc} : \forall\alpha.(\alpha\ \texttt{cont} \rightarrow \alpha) \rightarrow \alpha.$$

The variable-binding form, written `letcc` k `in` e, has the following typing rule:

$$\frac{A, k{:}\tau\ \texttt{cont} \vdash e : \tau}{A \vdash \texttt{letcc}\ k\ \texttt{in}\ e : \tau}$$

where A is a type assignment giving types to the set of free variables of e.

The choice of functional or binding form of continuation-capturing construct depends on the definition of the type τ `cont`. We consider two possibilities: regard a continuation as a function that is invoked by application, or regard a continuation as a new form of value that is invoked by a special primitive. In the

[1] We (temporarily) use ordinary function application notation to indicate invocation of a continuation.

first case the type τ `cont` is rendered as a functional type, whereas in the second it is introduced as a new primitive type. We consider each in turn.

If continuations are to be regarded as functions, some provision must be made for ensuring that the result type is allowed to vary according to context. This suggests the following polymorphic typing:

$$\texttt{callcc} : \forall\alpha.\forall\beta.((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha$$

But since `k` is lambda-bound in the expression `callcc(fn k => ...)` this does not give us the freedom to instantiate the polymorphic type variable β independently at each applied occurrence of `k` within the body of the abstraction. Instead we are forced to choose a *single* type for β suitable for all applications of `k`, ruling out examples such as the one considered above.

There are two ways to proceed. One involves moving the quantifier over β inward (which could be formally justified by the observation that β occurs in a positive position in the type expression), yielding the typing

$$\texttt{callcc} : \forall\alpha.((\alpha \rightarrow \forall\beta.\beta) \rightarrow \alpha) \rightarrow \alpha,$$

then replacing the type $\forall\beta.\beta$, which is not the type of any defined value, by a new primitive type `void`, resulting in the typing

$$\texttt{callcc} : \forall\alpha.((\alpha \rightarrow \texttt{void}) \rightarrow \alpha) \rightarrow \alpha.$$

The type τ `cont` is then defined to be the type $\tau \rightarrow$ `void`. To match the type of a continuation invocation (*i.e.*, `void`) with its context we could either view `void` as a subtype of all types and use a subsumption rule, which introduces many of the complexities of subtyping into the type system, or we can simply introduce a polymorphic coercion function

$$\texttt{ignore} : \forall\alpha.\texttt{void} \rightarrow \alpha$$

and surround applications of continuations with a call to `ignore`, as in

```
if b then [ ignore(k 3) + 1 ] else 5 :: ignore(k 4)
```

(where `k` is an `int cont`).

As an alternative to `ignore` we can exploit the polymorphic type system of ML, using `letcc` instead of `callcc`. The idea is to take type τ `cont` to stand for the polymorphic type $\forall\alpha.\tau \rightarrow \alpha$, leading to the following typing rule for `letcc`:

$$\frac{A, k : \forall\alpha.\tau \rightarrow \alpha \vdash e : \tau}{A \vdash \texttt{letcc}\ k\ \texttt{in}\ e : \tau}$$

Since k is assigned a polymorphic type, the result type, α, may be chosen freely on a case-by-case basis for each applied occurrence of k. This rule is consistent with the ML type system in that `let`-like constructs admit assignment of polymorphic types to the bound identifier. This method cannot be adapted to `callcc`. The required type has the form $(\forall\alpha.\tau \rightarrow \alpha) \rightarrow \tau$ which lies outside of the scope of the ML type system. It is here that the two constructs differ in an ML-like setting.

Another way of typing continuations, and the one currently adopted in Standard ML of New Jersey [1], is to abandon the view that continuations are functions in the ordinary sense and to consider τ `cont` as a primitive type with an operation `throw` for invoking a continuation. The type of `throw` is given by

$$\texttt{throw} : \forall\alpha.\forall\beta.(\alpha\ \texttt{cont}) \rightarrow (\alpha \rightarrow \beta),$$

and hence `throw` is essentially a coercion that turns a continuation into a function, introducing a separate instance of the type parameter β at each invocation of the continuation. Our example becomes

```
if b then [ (throw k 3) + 1 ] else 5 :: (throw k 4)
```

where the first and second occurrences of throw receive the types `int cont` $\rightarrow$ `int` $\rightarrow$ `int` and `int cont` $\rightarrow$ `int` $\rightarrow$ `int list`, respectively.

It is easy to define the `cont` and `throw` primitives in terms of `void` and `ignore`:

```
type α cont = α -> void
fun throw k x = ignore(k x).
```

Defining `void` and `ignore` in terms of `cont` and `throw` is a bit trickier, but it can be done:

```
abstype void = VOID
with
  fun ignore (x:void) : 'a =
      let fun loop() = loop() in loop() end
  val callcc =
      fn f => callcc(fn k => f((throw k) : 'a -> void))
end
```

So, in principle, there is not much to distinguish the two approaches. In practice, it is useful to be able to easily distinguish the invocation of a continuation from the application of a function. This is why `cont` and `throw` are the chosen primitives in Standard ML of New Jersey.

It would seem, then, that there are essentially two alternatives for representing continuations in ML: as polymorphic functions, using `letcc` as the capturing construct, and values of a new primitive type, using `throw` to invoke them. Although the two are equivalent for the purely functional fragment of ML, the approach based on a primitive type of continuations is better-behaved in the context of the full Standard ML language than is the polymorphic approach. The problem is that current schemes for introducing references (assignable cells) in ML preclude the possibility of storing objects of polymorphic type. For instance, if the identity function is stored into a cell, then a single instance of its polymorphic type must be chosen for all subsequent retrievals: its polymorphic character is lost. (See Tofte's thesis [38] for further discussion of this point.) Thus if a continuation was represented as a function of polymorphic result type, then the result type, which is irrelevant since no result is actually returned, would have to be fixed when the continuation is *stored*, rather than when it is *invoked*. This would significantly limit the utility of stored continuations because all invocations would have to be in the same type context. The approach based on a primitive type of continuations does not suffer from this limitation, and is therefore to be preferred for Standard ML.

We are thus led to adopt the following simple signature for first-class continuations in Standard ML:

$$\begin{array}{l}\texttt{type}\ \alpha\ \texttt{cont}\\ \texttt{val callcc : }(\alpha\ \texttt{cont} \rightarrow \alpha) \rightarrow \alpha\\ \texttt{val throw : }\alpha\ \texttt{cont} \rightarrow \alpha \rightarrow \beta\end{array}$$

As will be demonstrated in Section 3 this signature accurately reflects the typing properties of `callcc` and `throw`. However, certain combinations of the polymorphic `let` construct and first-class continuations lead to run-time type errors, as will be explained in Section 3. Restrictions on the type system similar to those considered by Tofte in connection with mutable cells [38, 39] suffice to recover soundness without sacrificing too many useful programs. These issues will be discussed in detail in Section 3, and a suitable soundness theorem can be obtained. For the remainder of this section we gloss over these issues, focusing instead on the use of first-class continuations. We stress that these programs are well-typed in the restricted type system, and hence do not "go wrong".

Some examples will suggest how first-class continuations are used in practice. The simplest and earliest use of continuations was to provide an escape, as in the following function that returns the product of a list of integers. If a zero is found the answer is returned via a continuation and no multiplications are performed.

```
fun prod l =
    callcc(fn exit =>
      let fun loop [] = 1
            | loop(0::t) = throw exit 0
            | loop(h::t) = h * loop t
      in loop l end)
```

Another common application is to implement coroutines. Here an interesting typing issue arises. A common technique is to resume a coroutine by passing the continuation of the current coroutine as the argument to the continuation representing the resumed coroutine. If `state` is the type representing the state of a coroutine, this leads naively to the circular identification

$$\texttt{state} = \texttt{state cont}$$

We cannot solve this identity directly, but we can use a datatype declaration to define the type `state` recursively. This is illustrated by the following example of a pair of coroutines, one producing and the other consuming a sequence of integers.

```
datatype state = S of state cont
fun resume(S k: state) : state =
    callcc(fn k': state cont => throw k (S k'))
val buffer = ref 0
fun produce(n: int, cons: state) =
    (buffer := n; produce(n+1, resume(cons)))
fun consume(prod: state) =
    (print(!buffer); consume(resume prod))
fun pinit (n: int) : state =
    callcc(fn k : state cont => produce(n,S k))
fun prun () = consume(pinit(0))
```

Coroutines can be generalized to lightweight processes or threads. Continuations have been used as the basis for the implementations of several process facilities for Standard ML of New Jersey, some of which use preemptive scheduling [28, 3, 27, 35].

The following example uses stored continuations to implement a simple backtracking scheme.

```
let
 val stack : unit cont list ref = ref []
 fun pushstate(k : unit cont) = stack := k :: !stack
 fun popstate() = stack := tl(!stack)
 fun backtrack() : 'a =
     case !stack
       of [] => raise Error
        | k :: r => (stack := r; throw k ())
 fun alt(a:unit -> unit,b:unit -> unit) =
     callcc(fn exit =>
         (callcc(fn k => (pushstate k;
                          a();
                          popstate();
                          throw exit ()));
         b()))
in ... backtracking application ...
end
```

Calls of `alt` can be nested (i.e. inside of the actions `a` and `b`), and `backtrack` can be called in any type context.

Another use of continuations is to provide a clean, typed interface for asynchronous signal handling [29]. In Standard ML of New Jersey the type of a signal handler is `(int * unit cont) -> unit cont`, where the argument is a pair consisting of a count of pending signals of the kind being handled and a continuation representing the state of the interrupted process. The continuation returned by the signal handler is typically used to resume the interrupted process after signals have been unmasked, but it can also provide an alternative continuation, for instance one that aborts the computation and returns to top-level. The signal handling module provides functions to set handlers for each signal and to mask all signals.

There is a subtle issue concerning the behavior of continuations in an interactive system. Unless the context of a continuation is carefully defined and controlled one can subvert the type system. The following sequence of top-level declarations illustrates the problem.

```
val c = ref NONE : int cont option ref;
val n:int = callcc(fn k => (c := SOME k; 2));
val b:bool = let val SOME k' = !c in throw k' 3 end;
```

We are dealing with expression continuations that merely deliver a value; the binding of that value in the top-level environment and the printing of a report for the user are the responsibility of the interactive top-level.[2] So the evaluation context represented by the continuation `k` stored in `c` is limited to the right hand side of the declaration of `n`, of type `int`. When this continuation is fetched and invoked in the right hand side of the declaration of `b`, that expression returns the value 3, which the top-level would erroneously interpret as a boolean value. To prevent this anomaly we must enforce a strict association between a continuation and its top-level context that determines the type of the answer returned. The continuation should "expire" when this context changes and if it is invoked after it has expired this should be detected and should generate an error message. In Standard ML of New Jersey, expiration of continuations is enforced by timestamping continuations. Each time a top-level evaluation is begun a new stamp v is generated and pushed onto a stack. The initial continuation used for that evaluation will finish by popping the stack and comparing the top value with v, and if they differ it will signal an error. A stack of stamps is used because "top-level" evaluations may be nested when files are loaded with the *use* function.

Another interesting issue is the relation between continuations and exception handling. (See also Griffin's recent work on this subject [17].) In the dynamic semantics of Standard ML, an expression can produce either a normal value or an exception packet indicating that an exception has been raised but not handled during the evaluation of the expression. Therefore the dynamic context of an expression, *i.e.* its continuation, must be able to deal with either sort of result. In effect, one could think of the dynamic context of an expression in ML as a pair of continuations, one for the "normal" value return, the other giving the exception handling context. This suggests possible new primitives that would provide access to the exception handling part of a continuation, either by invoking a continuation with an exception packet instead of a normal value, or by extracting the "exception handler" part of a continuation as a separate object. One reason why such primitives should not be provided is that in conjunction with asynchronous signal handling they introduce the possibility of asynchronous exceptions. The existence of asynchronous exceptions makes it impossible to statically verify that a particular expression cannot raise a particular exception, precluding some compiler optimizations.

3 Semantics of Polymorphic Type Assignment

In this section we make precise the informal ideas presented in Section 2. We begin by recalling the polymorphic type assignment system introduced by Damas and Milner [4]. This type system will form the basis of our investigation of the typing properties of the continuation-passing primitives introduced in Section 2. We then give a continuation-passing structured operational semantics for this language, and prove the soundness of type assignment with respect to this semantics. We then show that the extension of the Damas-Milner type system with the continuation-passing primitives of Section 2 is unsound. We consider two approaches to recovering soundness, one based on the restriction of polymorphism to values, and the other based on Tofte's notion of imperative type variable [39].

3.1 The Damas-Milner Language

We begin by recalling the polymorphic type assignment system introduced by Damas and Milner [4].

The syntax of *ordinary expressions* is given by the following grammar:

$$\textit{expressions} \quad e \quad ::= \quad x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2$$

The metavariable x ranges over a countably infinite set of variables, and the metavariable c ranges over a countable set of constants. The set $\mathrm{FV}(e)$ of variables occuring freely in e is defined as usual, as is the operation of capture-avoiding substitution of an expression e for free occurrences of a variable x in another expression e', written $[e/x]e'$. Expressions differing only in the names of the bound variables are identified; we are therefore free to assume that bound variable names may always be chosen so as to avoid conflicts.

[2] The interface between the interactive system and the object level evaluation is similar to a *prompt* [9].

The syntax of *type expressions* is given by the following grammar:

$$\begin{array}{llll} \textit{monotypes} & \tau & ::= & b \mid t \mid \tau_1 \to \tau_2 \\ \textit{polytypes} & \sigma & ::= & \tau \mid \forall t.\sigma \end{array}$$

The metavariable t ranges over a countably infinite set of *type variables*, and the metavariable b ranges over a countable set of *base types*. The set $\mathrm{FTV}(\sigma)$ of type variables occurring freely in a polytype σ is defined as usual, as is the operation of capture-avoiding substitution of a monotype τ for free occurrences of a type variable t in a poltype σ, written $[\tau/t]\sigma$.

A *context* is a finite sequence of variable declarations of the form $x{:}\sigma$ assigning the polytype σ to the variable x, subject to the condition that no variable may be assigned more than one polytype. The variable Γ ranges over contexts. The "no redeclaration" condition implies that a context may be regarded as a partial function sending a variable x to the unique (if it exists) σ such that $x{:}\sigma$ is in Γ. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ regarded as a partial function, and $\Gamma, x{:}\sigma$, where $x \notin \mathrm{dom}(\Gamma)$, for the extension of Γ with the given declaration. The set of type variables occuring freely in a context Γ, written $\mathrm{FTV}(\Gamma)$, is defined to be $\bigcup_{x \in \mathrm{dom}(\Gamma)} \mathrm{FTV}(\Gamma(x))$. A *signature* is a finite sequence of constant declarations of the form $c{:}\sigma$, subject to the condition that no constant is declared more than once. Notational conventions similar to those for contexts apply to signatures.

We shall work with a syntax-directed formulation of the Damas-Milner polymorphic type assignment system inspired by the static semantics of Standard ML [25]. The rules given in Table 1 define a formal system for deriving judgements of the form $\Gamma \vdash e : \tau$, expressing that the expression e may be assigned the monotype τ in context Γ. The rules are parametric in a signature Σ, which we leave implicit. We often write $\Gamma \vdash e : \tau$, or just $e : \tau$ when Γ is empty, to mean that this typing judgement is derivable in accordance with the rules of Table 1. An expression e is said to be *well-typed* in a context Γ iff there exists a τ such that $\Gamma \vdash e : \tau$.

Some of the rules given in Table 1 make use of auxiliary notions that merit further explanation. Rule VAR makes use of the *polymorphic instance* relation $\sigma \geq \tau$ which is defined to hold iff σ is a polytype of the form $\forall t_1. \ldots .\forall t_n.\tau'$ and τ is a monotype of the form $[\tau_1, \ldots, \tau_n/t_1, \ldots, t_n]\tau'$ for some monotypes $\tau_1, \ldots, \tau_n$. This relation is extended to polytypes by defining $\sigma \geq \sigma'$ iff $\sigma \geq \tau$ whenever $\sigma' \geq \tau$. Rule LET makes use of *polymorphic generalization* of a monotype τ in a context Γ, $\mathrm{Close}_\Gamma(\tau)$, defined to be the polytype $\forall t_1. \ldots .\forall t_n.\tau$, where $\mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\Gamma) = \{ t_1, \ldots, t_n \}$. We sometimes abbreviate $\mathrm{Close}_\Gamma(\tau)$ to just $\mathrm{Close}(\tau)$ when Γ is the empty context.

The formal system of Table 1 is clearly a subsystem of the system given by Damas and Milner [4] in the sense that if $\Gamma \vdash e : \tau$ is derivable in the system of Table 1, then it is derivable in Damas and Milner's system. Conversely, if $\Gamma \vdash e : \sigma$ is derivable in Damas and Milner's system, then $\Gamma \vdash e : \tau$ is derivable in the system of Table 1 whenever $\sigma \geq \tau$. Thus all and only the monotypes derivable for a given term in Damas and Milner's system are derivable in the system considered here. The advantage of the formulation given here is that the rules are syntax-directed — there is precisely one rule for each form of ordinary expression.

The following lemma summarizes some important properties of the type system that will be of some use in the proof of Theorem 3.2.

Lemma 3.1 *1. If $\Gamma \vdash e : \tau$ and $x \in \mathrm{FV}(e)$, then $x \in \mathrm{dom}(\Gamma)$.*

2. If $\Gamma \vdash e : \tau$, $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma, x{:}\sigma \vdash e : \tau$.

3. If $\Gamma \vdash e : \tau$ and $\Gamma, x{:}\sigma \vdash e' : \tau'$ with $\mathrm{Close}_\Gamma(\tau) \geq \sigma$, then $\Gamma \vdash [e/x]e' : \tau'$.

3.2 Operational Semantics

The operational semantics of untyped terms is a three-place relation $k \vdash e \Rightarrow a$ defined by a set of inference rules. Here e is the expression being evaluated, k is a "continuation" representing the current evaluation context, and a is the final "answer" obtained by evaluating e in context k.

Table 1: Polymorphic Type Assignment

$$\frac{\Gamma(x) \geq \tau}{\Gamma \vdash x : \tau} \qquad (\textsc{var})$$

$$\frac{\Sigma(c) \geq \tau}{\Gamma \vdash c : \tau} \qquad (\textsc{const})$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\textsc{abs})$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau} \qquad (\textsc{app})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\,\mathrm{Close}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\textsc{let})$$

The semantics is defined in terms of three additional syntactic categories, defined by the following grammar:

$$\begin{array}{llll}
\textit{values} & v & ::= & c \mid \lambda x.e \\
\textit{continuations} & k & ::= & [] \mid k\,e \mid v\,k \mid \mathsf{let}\ x\ \mathsf{be}\ k\ \mathsf{in}\ e \\
\textit{answers} & a & ::= & v \mid \mathsf{wrong}
\end{array}$$

Values, continuations, and answers are all assumed to have no free variables. A continuation k has precisely one "hole", designated by the symbol "[]". If k is a continuation, we write $k[e]$ for the expression obtained by "filling" the hole in k with the expression e. Similarly, if k' is another continuation, $k[k']$ is the continuation that results from filling the hole in k with k', and may be thought of as the "composition" of k with k'.

The operational semantics of the language of untyped terms given above is defined by the rules of Table 2. We sometimes write $e \Rightarrow a$ to mean $[] \vdash e \Rightarrow a$.

The idea underlying the operational semantics is that in a judgement $k \vdash e \Rightarrow a$ the continuation k represents an "evaluation context" (in the sense of Felleisen [10]), and e represents the expression being evaluated in that context. If e is a value, then evaluation continues by replacing the "hole" in k with v, and re-starting the evaluation process. If e is not a value, then either it is reducible by rule BETA or SUB, in which case the reduction is performed, or else evaluation proceeds to a sub-expression of e, extending the continuation k accordingly.

3.3 Soundness of Type Assignment

The soundness of the polymorphic type system with respect to the operational semantics is summarized by the slogan "well-typed programs cannot go wrong". In other words, if a program has a type according to the type assignment system, then evaluation of that program, starting in the empty context, cannot result in the answer wrong. We prove a slightly stronger result admitting general evaluation contexts, and phrase the theorem in terms of preservation of typing under evaluation. (See Felleisen and Wright [42] for a similar perspective.)

Theorem 3.2 *Let α be an arbitrary monotype. If $k \vdash e \Rightarrow a$ with $e : \tau$ and $x{:}\,\mathrm{Close}(\tau) \vdash k[x] : \alpha$, then a is a value and $a : \alpha$.*

Proof The proof proceeds by induction on the structure of the derivation of $k \vdash e \Rightarrow a$.

VAL0: In this case $e = v = a$ and $k = []$. It is immediate that a is a value. We assume $v : \tau$ and $x{:}\,\mathrm{Close}(\tau) \vdash x : \alpha$ since $k[x] = x$, and it follows from Lemma 3.1, part 3, that $v{:}\alpha$, and hence $a{:}\alpha$.

Table 2: Operational Semantics

$$[] \vdash v \Rightarrow v \tag{VAL0}$$

$$\frac{[] \vdash k[v] \Rightarrow a}{k \vdash v \Rightarrow a} \quad (k \neq []) \tag{VAL1}$$

$$\frac{k[[]\, e_2] \vdash e_1 \Rightarrow a}{k \vdash e_1\, e_2 \Rightarrow a} \quad (e_1 \textit{ not a value}) \tag{FN}$$

$$\frac{k[v_1\, []] \vdash e_2 \Rightarrow a}{k \vdash v_1\, e_2 \Rightarrow a} \quad (e_2 \textit{ not a value}) \tag{ARG}$$

$$\frac{k \vdash [v_2/x]e_1 \Rightarrow a}{k \vdash (\lambda x.e_1)\, v_2 \Rightarrow a} \tag{BETA}$$

$$k \vdash v_1\, v_2 \Rightarrow \mathsf{wrong} \quad (v_1 \neq \lambda x.e) \tag{WRONG}$$

$$\frac{k[\mathsf{let}\, x\, \mathsf{be}\, []\, \mathsf{in}\, e_2] \vdash e_1 \Rightarrow a}{k \vdash \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 \Rightarrow a} \quad (e_1 \textit{ not a value}) \tag{BIND}$$

$$\frac{k \vdash [v_1/x]e_2 \Rightarrow a}{k \vdash \mathsf{let}\, x\, \mathsf{be}\, v_1\, \mathsf{in}\, e_2 \Rightarrow a} \tag{SUB}$$

VAL1: Assuming $k \neq []$ we have by rule VAL1 that $[] \vdash k[v] \Rightarrow a$. From the assumptions that $v : \tau$ and $x{:}\,\mathrm{Close}(\tau) \vdash k[x] : \alpha$ and Lemma 3.1, part 3, it follows that $k[v] : \alpha$. Clearly $x{:}\,\mathrm{Close}(\alpha) \vdash x{:}\alpha$, and hence $a : \alpha$ by induction.

FN: By rule FN we have $k[[]\, e_2] \vdash e_1 \Rightarrow a$. By type rule APP we have $e_1 : \tau_2 {\rightarrow} \tau$ and $e_2 : \tau_2$ for some type τ_2. By Lemma 3.1, part 2, and the type assumption on k we have $y{:}\,\mathrm{Close}(\tau_2 {\rightarrow} \tau), x{:}\,\mathrm{Close}(\tau) \vdash k[x] : \alpha$, and clearly $y{:}\,\mathrm{Close}(\tau_2 {\rightarrow} \tau) \vdash y\, e_2 : \tau$. Hence by Lemma 3.1, part 3, we have $y{:}\,\mathrm{Close}(\tau_2 {\rightarrow} \tau) \vdash k[[y]e_2] : \alpha$. It follows by induction that $a : \alpha$.

ARG: Since $v_1\, e_2 : \tau$ there is a type τ_2 such that $v_1 : \tau_2 {\rightarrow} \tau$ and $e_2 : \tau_2$. It suffices to show that $y{:}\,\mathrm{Close}(\tau_2) \vdash k[v_1\, y] : \alpha$, which follows from $y{:}\,\mathrm{Close}(\tau_2) \vdash v_1\, y : \tau$ by an argument similar to that of the FN case.

BETA: We have $k \vdash [v_2/x]e_1 \Rightarrow a$ by the BETA rule, and $x{:}\,\mathrm{Close}(\tau) \vdash k[x] : \alpha$ by assumption. It suffices to show that $[v_2/x]e_1 : \tau$. This follows from the assumption that $(\lambda x.e_1)\, v_2 : \tau$ by the typing rules and an application of Lemma 3.1, part 3.

WRONG: By assumption $v_1\, v_2 : \tau$, and hence $v_1 : \tau_2 \rightarrow \tau$ for some monotype τ_2. Since the value v_1 is assumed not to be a λ-abstraction, it must be a constant. But there are no constants of functional type, yielding a contradiction. Therefore this rule will not apply to a well-typed expression.

BIND: By the LET typing rule, there exists a type τ_1 such that $e_1 : \tau_1$ and $x{:}\,\mathrm{Close}(\tau_1) \vdash e_2 : \tau$. It suffices to show that $x_1{:}\,\mathrm{Close}(\tau_1) \vdash k[\mathsf{let}\, x\, \mathsf{be}\, x_1\, \mathsf{in}\, e_2] : \alpha$ (where x_1 is a fresh variable), for then we obtain $a : \alpha$ by an application of the induction hypothesis. Let Γ be $x_1{:}\,\mathrm{Close}(\tau_1)$. Then since Γ contains no free type variables, $\mathrm{Close}_\Gamma(\tau') = \mathrm{Close}(\tau')$ for any type τ'. Since $\mathrm{Close}(\tau_1) \geq \tau_1$ we have $\Gamma \vdash x_1 : \tau_1$. From the typing assumption on e_2 we have by Lemma 3.1, part 2, and that fact that $\mathrm{Close}(\tau_1) = \mathrm{Close}_\Gamma(\tau_1)$, $\Gamma, x{:}\,\mathrm{Close}_\Gamma(\tau_1) \vdash e_2 : \tau$. Hence by the LET rule we have $\Gamma \vdash \mathsf{let}\, x\, \mathsf{be}\, x_1\, \mathsf{in}\, e_2 : \tau$. Then, from the typing hypothesis on k, by Lemma 3.1, part 3, and the equality $\mathrm{Close}(\tau) = \mathrm{Close}_\Gamma(\tau)$, we have $\Gamma \vdash k[\mathsf{let}\, x\, \mathsf{be}\, x_1\, \mathsf{in}\, e_2] : \alpha$.

Table 3: Semantics of Continuation-Passing Primitives

$$\frac{k \vdash v\, k \Rightarrow a}{k \vdash \mathsf{callcc}\; v \Rightarrow a} \qquad \text{(SEIZE)}$$

$$\frac{k' \vdash v \Rightarrow a}{k \vdash \mathsf{throw}\; k'\; v \Rightarrow a} \qquad \text{(JUMP)}$$

$$k \vdash v_1\, v_2 \Rightarrow \mathsf{wrong} \quad (v_1 \neq \lambda x.e,\ v_1 \notin \{\,\mathsf{callcc}, \mathsf{throw}\,\}) \qquad \text{(WRONG)}$$

SUB: This case is similar to the BETA case. It suffices to show that $[v_1/x]e_2 : \tau$, which follows from the inductive assumptions by an application of Lemma 3.1, part 3.

□

The soundness of the polymorphic type system with respect to the continuation semantics follows directly:

Corollary 3.3 *If e is a well-typed program of type τ, and $e \Rightarrow a$, then $a \neq \mathsf{wrong}$.*

Proof Take $\alpha = \tau$ and $k = []$ in Theorem 3.2. □

It is worth mentioning that Theorem 3.2 does not directly entail any positive conditions on typing — the theorem assures us that a well-typed program cannot "go wrong", but does not provide any information about the actual result of evaluation. This is in contrast to soundness theorems based on a "direct" semantics which often establish, for example, that a program of base type yields a value of that base type, if it yields a value at all. To obtain such a result in this setting seems to require the use of an appropriate form of logical relation [26, 33]. An alternative is to consider the typing properties of the call-by-value cps transform, from which an observational soundness theorem may be extracted [5, 18, 24].

3.4 First-Class Continuations

To account for the continuation-passing primitives introduced in Section 2, we extend the language of monotypes as follows:

$$\textit{monotypes} \quad \tau \quad ::= \quad \ldots \mid \tau\, \mathsf{cont}$$

The signature Σ_{cont} is given by the following declarations:

$$\begin{array}{rcl} \mathsf{callcc} & : & \forall t.(t\, \mathsf{cont} \rightarrow t) \rightarrow t \\ \mathsf{throw} & : & \forall s.\forall t.s\, \mathsf{cont} \rightarrow s \rightarrow t \end{array}$$

The operational semantics of callcc and throw is defined in such a way that continuations must be "reified" as values. Accordingly we extend the syntax of values:

$$\textit{values} \quad v \quad ::= \quad \ldots \mid \mathsf{throw}\; v \mid k$$

(Since throw is represented by a two-place, curried function, it is necessary for technical reasons to regard the partial application of throw to a single argument to be a value.) The typing relation $v : \tau$ is extended to this additional case by defining $k : \tau\, \mathsf{cont}$ to hold iff $x{:}\tau \vdash k[x] : \alpha$, where α is a fixed type of answers.

The extension of the operational semantics to cover callcc and throw is given in Table 3.

We turn now to the question of soundness of type assignment for the extension of the semantics with first-class continuations. Despite the superficial plausibility of the typing rules for callcc and throw, the full polymorphic type assignment system for the extended language is unsound.[3] Specifically, assume that we

[3] This result was obtained jointly by Mark Lillibridge and the first author [18].

have base types int and bool, and constants $\mathsf{true} : \mathsf{bool}$ and $1 : \mathsf{int}$. The following program has type *bool* in the empty context, but yields answer 1 when evaluated in the empty context:

$$\mathsf{let}\ f\ \mathsf{be}\ \mathsf{callcc}\,(\lambda k.\lambda x.\mathsf{throw}\ k\ (\lambda y.x))\ \mathsf{in}\ f\ 1\ ;\ f\ \mathsf{true}$$

In the presence of a primitive operation such as logical negation that "goes wrong" on an integer argument this program may be readily adapted to give a counterexample to soundness. To see what is wrong, it is helpful to consider a naïve attempt to extend the proof of Theorem 3.2 to the present setting. Difficulties arise only in the case of SEIZE. Specifically, we have by assumption that $k \vdash \mathsf{callcc}\ v \Rightarrow a$ with $\mathsf{callcc}\ v : \tau$ and $x{:}\,\mathrm{Close}(\tau) \vdash k[x] : \alpha$; we are to show that $a : \alpha$. For the induction it suffices to show that $v\,k : \tau$, for which it is sufficient to show that $v : \tau\,\mathsf{cont} \rightarrow \tau$ and that $k : \tau\,\mathsf{cont}$, which is to say $x{:}\tau \vdash k[x] : \alpha$. But all we have is the weaker condition $x{:}\,\mathrm{Close}(\tau) \vdash k[x] : \alpha$, which is not sufficient. The counterexample shows that the soundness theorem fails for the enriched language.

The essential difficulty is that there are continuations which use their arguments polymorphically, but this cannot be expressed in the Damas-Milner type discipline due to the limitation to "prenex" quantification. The only such continuations are those of the form $\mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ e_2$ which arise during evaluation of let expressions whose let-bound expression is not a value, reflecting the sequential evaluation semantics given to such expressions in ML. Soundness in the presence of continuation-passing primitives may be recovered by considering variants of the language in which such continuations do not arise. We consider two such variants, one in which let expressions are given a "by name" interpretation, and one in which let-bound expressions are limited to values. In both of these variants continuations of the form $k[\mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ e]$ can be discarded, and all remaining continuations type check monomorphically, which is sufficient for soundness in the presence of callcc and throw.

The "by name" semantics for let expressions is defined by replacing rules BIND and SUB by the following rule:

$$\frac{k \vdash [e_1/x]e_2 \Rightarrow a}{k \vdash \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \Rightarrow a} \qquad (\text{SUB-NAME})$$

The "values only" semantics is defined by restricting let expressions so that the let-bound expression is a value, and dropping rule BIND from the operational semantics. In either case we omit continuations of the form $k[\mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ e_2]$.

Theorem 3.4 *In the language with* callcc *and* throw *and with either the "by-name" or "values-only" interpretation of* let *if* $k \vdash e \Rightarrow a$ *with* $e : \tau$ *and* $x{:}\tau \vdash k[x] : \alpha$, *then* $a : \alpha$.

Proof Similar to the proof of Theorem 3.2, amended as follows:

SEIZE: By assumption $\mathsf{callcc}\,v : \tau$, and hence $v : \tau\,\mathsf{cont} \rightarrow \tau$. It suffices to show that $v\,k : \tau$, which follows from $k : \tau\,\mathsf{cont}$, which follows from $x{:}\tau \vdash k[x] : \alpha$, which holds by assumption.

JUMP: By the inductive assumptions $\mathsf{throw}\,v\,k : \tau$, which implies that $v : \tau$ and $k : \tau\,\mathsf{cont}$, and consequently that $x{:}\tau \vdash k[x] : \alpha$. The result then follows by an application of the inductive hypothesis.

WRONG: If v_1 is neither a λ-abstraction nor callcc nor throw, then it cannot be well-typed.

The SUB case is handled as before; the BIND case no longer arises. □

The "by name" and "values only" variants of the language recover soundness in the presence of continuation-passing primitives at the expense of disturbing the semantics of let expressions even in those programs that make no use of callcc and throw. The pure functional core language has the appealing property that a polymorphic expression is evaluated once but used many times at many different types. Neither the "by name" nor the "values only" variants enjoy this property. Under the "by name" interepretation of polymorphism, the bound expression may be evaluated multiply (if at all). Under the "values only" restriction the single evaluation property is preserved only in a trivial sense since no non-value expression can be used polymorphically. It is natural to inquire whether it is possible to smoothly integrate continuation-passing primitives into the language in such a way as to retain soundness but nevertheless retaining the semantics of the pure functional core language. This can be achieved to a limited extent, at the expense of introducing a somewhat more complex type system adapted from Tofte's treatment of polymorphic references [39].

The set of type variables is divided into two disjoint infinite subsets, the *applicative* and the *imperative* type variables. Imperative type variables are written here with an underscore; applicative type variables are left unadorned. A type is said to be imperative iff all type variables occurring within it are imperative. The universal quantifier may bind either sort of type variables and the polymorphic instantiation relation is restricted so that an imperative type variable may only be instantiated by an imperative type; ordinary type variables may be freely instantiated by any type expression without restriction. The polymorphic closure operation is defined as before, retaining the applicative/imperative distinction when type variables are bound by a quantifier. The *applicative closure* operation is defined similarly, except that only applicative type variables may be quantified, according to whether or not they occur in the given typing context. We write $\mathrm{AppClose}_\Gamma(\tau)$ for the applicative closure of the type τ relative to the type assignment Γ.

Imperative type variables are used in the signature $\Sigma^{imp}_{\mathsf{cont}}$ given by the following declarations:

$$\begin{array}{rcl} \mathsf{callcc} & : & \forall \underline{t}.(\underline{t}\,\mathsf{cont}\to\underline{t})\to\underline{t} \\ \mathsf{throw} & : & \forall s.\forall t.s\,\mathsf{cont}\to s\to t \end{array}$$

The type of callcc differs from that in the signature Σ_{cont} by requiring that instances of the quantified type be imperative.

In the simplest version of the imperative type discipline the let typing rule is restricted so that only applicative type variables may be polymorphic.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\,\mathrm{AppClose}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 : \tau_2} \qquad (\text{LET-APPL})$$

In the absence of imperative type variables $\mathrm{AppClose}_\Gamma(\tau) = \mathrm{Close}_\Gamma(\tau)$, and hence programs that make no use of callcc will type check exactly as in the functional core language.

The soundness of this type system relative to the dynamic semantics given in Table 2 is stated as follows.

Theorem 3.5 *If $e : \tau$ and $k \vdash e \Rightarrow a$, where $x{:}\,\mathrm{AppClose}(\tau) \vdash k[x] : \alpha$, then a is a value such that $a : \alpha$.*

Proof The proof is very similar to the proof of Theorem 3.2. We give here only the two most interesting cases.

SEIZE If $\mathsf{callcc}\, v : \tau$, then $v : \tau\,\mathsf{cont}\to\tau$, and τ is imperative. Therefore $\mathrm{AppClose}(\tau) = \tau$, and hence the assumption on k ensures that $k : \tau\,\mathsf{cont}$. Therefore $v\,k : \tau$, and the result follows by an application of the induction hypothesis.

BIND If $e = \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2$ where e_1 is a not a value, then it follows from the assumptions that $e_1 : \tau_1$ and $x : \mathrm{AppClose}(\tau_1) \vdash e_2 : \tau$ for some type τ_1 and that $k[\mathsf{let}\, x\, \mathsf{be}\, [\,]\, \mathsf{in}\, e_2] \vdash e_1 \Rightarrow a$. It suffices to show that $x_1 : \mathrm{AppClose}(\tau_1) \vdash k[\mathsf{let}\, x\, \mathsf{be}\, x_1\, \mathsf{in}\, e_2] : \alpha$ (where x_1 is a fresh variable). Let Γ be $x_1 : \mathrm{AppClose}(\tau_1)$, and note that $\mathrm{AppClose}_\Gamma(\tau') = \mathrm{AppClose}(\tau')$ for any type τ' since Γ contains no free applicative type variables. Since $\mathrm{AppClose}(\tau_1) \geq \tau_1$, we have $\Gamma \vdash x_1 : \tau_1$. By the typing assumption on e_2 we have by Lemma 3.1, part 2, and the fact that $\mathrm{AppClose}(\tau_1) = \mathrm{AppClose}_\Gamma(\tau_1)$, we have $\Gamma, x{:}\,\mathrm{AppClose}_\Gamma(\tau_1) \vdash e_2 : \tau$. Consequently, by the LET rule, we have $\Gamma \vdash \mathsf{let}\, x\, \mathsf{be}\, x_1\, \mathsf{in}\, e_2 : \tau$ and therefore $\Gamma \vdash k[\mathsf{let}\, x\, \mathsf{be}\, x_1\, \mathsf{in}\, e_2] : \alpha$ by an application of Lemma 3.1, part 3, bearing in mind that $\mathrm{AppClose}_\Gamma(\tau) = \mathrm{AppClose}(\tau)$.

SUB If $\mathsf{let}\, x\, \mathsf{be}\, v_1\, \mathsf{in}\, e_2 : \tau_2$, then $v_1 : \tau_1$ and $\mathrm{AppClose}(\tau_1) \vdash e_2 : \tau_2$, and hence $[v_1/x]e_2 : \tau_2$ by Lemma 3.1, part 3, since $\mathrm{Close}(\tau_1) \geq \mathrm{AppClose}(\tau_1)$. The result follows by an application of the induction hypothesis.

□

The imperative type discipline allows us to conservatively track the occurrence of type variables in the argument type of a continuation so that we may suppress polymorphism whenever a continuation might be seized by an instance of callcc. However, the analysis is excessively conservative since it tracks those type variables that occur in the type of some instance of callcc without regard to whether or not that instance can actually lead to the capture of a polymorphic continuation. In particular, the simple imperative type discipline will preclude polymorphic generalization in the expression let c be callcc in e, even though it is safe to do so. This situation, and many like it, can be handled by admitting full polymorphism in the case that the let-bound expression is either a variable or a value. The typing rule LET-APPL is replaced by the following two rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\,\mathrm{Close}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2} \quad (e_1 \textit{ is a variable or a value}) \qquad (\text{LET-VAL})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\,\mathrm{AppClose}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2} \quad (e_1 \textit{ is not a variable nor a value}) \qquad (\text{LET-COMP})$$

The soundness of the strengthened imperative type system is stated as follows.

Theorem 3.6 *Let e be an expression such that $e : \tau$ and let k be a continuation such that $x : \sigma \vdash k[x] : \alpha$, where $\sigma = \mathrm{Close}(\tau)$ if e is a value and $\sigma = \mathrm{AppClose}(\tau)$ otherwise. If $k \vdash e \Rightarrow a$, then a is a value such that $a : \alpha$.*

The proof is a straightforward adaptation of the arguments given earlier for the simple imperative type discipline and for the "values only" restriction. The condition on k is phrased so that a continuation may use its argument polymorphically only if the next expression to be evaluated is a value. The two let rules ensure that this invariant is preserved.

4 Conclusions

First-class continuations are a powerful tool for implementing sophisticated control constructs like coroutines, processes, backtracking, and asynchronous signals. Until now they have been studied and employed in the context of dynamically typed languages like Scheme. We have been pleasantly surprised to discover that first-class continuations can also be accommodated in a polymorphically typed language like ML simply by adding a new primitive type with a couple of associated operations. In fact, the added discipline of the ML type system seems to simplify programming with first-class continuations. We have made the first steps toward integrating first-class continuations into the semantics of Standard ML and verifying the metaproperties of soundness and observational soundness, but it is clear that extensive work is required to integrate continuations fully into the definition of Standard ML.

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[2] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for higher-level semantic algebra. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, Cambridge, 1985.

[3] 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.

[4] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

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

[6] R. Kent Dybvig and Bob Hieb. Engines from continuations. *Journal of Computer Languages*, 14(2):109–124, 1989.

[7] Arthur Evans. PAL – a language designed for teaching programming linguistics. In *Proc. ACM 23rd National Conference*, pages 395–403, Princeton, 1968. ACM, Brandin Systems Press.

[8] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University Computer Science Department, 1985.

[9] Matthias Felleisen. The theory and practice of first-class prompts. In *Fifteenth ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, CA, 1988. ACM.

[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] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[13] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In *Summer Conference on Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, Manchester, UK, 1989. Springer-Verlag.

[14] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, Copenhagen, Denmark, August 1989. (DIKU Report 89/11).

[15] Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformations and Programming Environments*, pages 263–274. Springer-Verlag, 1985.

[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 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.

[19] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[20] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages*, 12:109–121, 1987.

[21] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9:582–598, 1987.

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

[23] Peter J. Landin. A correspondence between ALGOL–60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 1965.

[24] 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.

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

[26] Gordon Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.

[27] Norman Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Computer Science Department, Princeton University, 1990.

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

[29] John Reppy. Asynchronous signals in Standard ML. (Unpublished manuscript.), August 1990.

[30] John C. Reynolds. GEDANKEN — a simple typeless languages based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, May 1970.

[31] 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.

[32] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proc. 1990 Conference on LISP and Functional Programming*, pages 161–175, June 1990.

[33] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

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

[35] Bernard Sufrin. CSP-style processes in ML. (Private communication), 1989.

[36] Gerald Jay Sussman and Jr. Guy Lewis Steele. Scheme: An interpreter for extended lambda calculus. Technical Report Memo No. 349, MIT AI Laboratory, December 1975.

[37] Carolyn Talcott. Rum: An intensional theory of function and control abstractions. In *Proc. 1987 Workshop of Foundations of Logic and Functional Programming*, volume 306 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[38] 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.

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

[40] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, 1980.

[41] Andrew Wright and Matthias Felleisen. The nature of exceptions in polymorphic lannguages. Unpublished manuscript, Rice University, 1990.

[42] 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*.