Typing First-Class Continuations in ML

Bruce F. Duba[1] Robert Harper
David MacQueen[2]

October, 1990

CMU-CS-90-184

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a preprint of a POPL '91 article; all citations should refer to the conference proceedings.

Abstract

An extension of Standard 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. The semantics of type assignment for a small, purely functional fragment of the language is presented, for which both a Milner-style soundness theorem and an observational soundness theorem may be established.

[1]Rice University, Houston, TX

[2]AT&T Bell Laboratories, Murray Hill, NJ

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404.

Keywords: lambda calculus.type theory.continuations.functional programming

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 [16], exceptions [42], and logic variables [8, 19], for supporting multiple threads of control [41, 18, 29, 6], for providing asynchronous signal handlers [30], and for implementing non-blind backtracking [14] and dynamic barriers such as unwind-protect [17]. Tractable logics for reasoning about program equivalence in the presence of first-class continuations in an untyped setting have been developed [9, 10, 39]. Recent studies of continuations have addressed the question of their typing in a restricted setting [13, 12, 15] and their impact on full abstraction results [34].

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.

This paper is organized as follows. 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 present a formal system of type assignment for a small functional fragment of ML. A denotational semantics for this fragment is given in Section 3, and the semantics of type assignment is considered. The main results are a Milner-style soundness theorem ("well-typed programs cannot go wrong") and an observational soundness theorem ("convergent programs of type `int` yield integers"). Finally, in Section 4 we give an operational semantics for the language in the "natural semantics" style of Plotkin and Kahn [27, 3]. The operational presentation illustrates the extent to which the definition of Standard ML [24] would have to be changed to accommodate the proposed extension.

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

2 Adding Continuations to ML

First-class continuations are an abstraction that evolved from various nonstandard control structures such as Landin's J-operator [21], Reynold's operator `escape` [32], label variables in Gedanken [31] and PAL [7], and from the semantic analyses of general control structures, including jumps [36]. Scheme [38] 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 [4]. 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 continuations expect 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 list`).

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

$$(\tau\ \mathtt{cont} \to \tau) \to \tau$$

since the body may either invoke the passed continuation, or else return normally. Written polymorphically, the type of `callcc` is then

$$\forall\alpha.(\alpha\ \mathtt{cont} \to \alpha) \to \alpha.$$

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

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 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:

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

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

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

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

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

The type τ `cont` is then defined to be the type $\tau \to$ `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

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

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`).

A-1

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 [2], 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, reintroducing 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 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 [40] 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 the following simple signature for supporting first-class continuations in Standard ML:

```
type α cont
val callcc : (α cont → α) → α
val throw : α cont → α → β
```

(We could just as well have taken letcc as primitive, but since there is no advantage in doing so, it is simpler to introduce callcc as a new constant of polymorphic type.)

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 such that 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 buf = ref 0
fun produce(n: int, cons: state) =
    (buf := n; produce(n+1, resume(cons)))
fun consume(prod: state) =
    (print(!buf); 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 several implementations of process facilities for Standard ML of New Jersey, some of which use preemptive scheduling [29, 5, 28, 37].

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 [30]. 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 the continuation representing 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 like aborting the computation and returning 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 a strict association between a continuation and its top-level context (which determines the type of the answer returned) must be enforced. 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. In the dynamic semantics of Standard ML, an expression can either produce a normal value, or an exception value 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 either invoke a continuation with an exception value rather than a normal value, or return the "exception handler" part of a continuation. One pragmatic reason why such primitives are not provided is that in conjunction with asynchronous signal

[2]The interface between the interactive system and the object level evaluation is similar to a *prompt* [11].

handling they would introduce the possibility of asynchronous exceptions. These would make it impossible to statically verify that a particular expression could not raise a particular exception, precluding some compiler optimizations.

3 A Denotational Semantics of Typing

In this section we study the soundness of type assignment for a small, purely functional, monomorphic fragment of ML extended with primitives for first-class continuations. Explicit treatment of polymorphism and let is omitted since the main issues do not involve polymorphism.

3.1 Type Assignment

Consider the following language:

$$M ::= x \mid \lambda x.M \mid M\,M' \mid \texttt{letcc}\,x\,\texttt{in}\,M \mid \texttt{throw}\,M\,M'$$

The variables x, y, and z range over a set of variables, and M, N, P, Q, and R range over the set of terms. The expression $\texttt{letcc}\,x\,\texttt{in}\,M$ binds x in M. We use letcc in place of callcc to facilitate comparison between the various approaches to type assignment.

Let b range over some set B of base types. Type expressions are defined by the following grammar:

$$\tau \quad ::= \quad b \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \mathit{cont}$$

A *typing context* is a partial function Γ mapping some finite set of variables $\mathrm{dom}(\Gamma)$ to type expressions. A *typing assertion* is a triple $\Gamma \vdash M : \tau$, where τ is a type expression.

We take as given a map *type* assigning a type to each constant. (In a polymorphic system this map would assign a type scheme to each constant.) The type assignment rules for the above language are as follows:

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(VAR)}$$

$$\frac{\Gamma[x : \tau_1] \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \rightarrow \tau_2} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash M : \tau \rightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'} \qquad \text{(APP)}$$

$$\frac{\Gamma[x : \tau\ \mathit{cont}] \vdash M : \tau}{\Gamma \vdash \texttt{letcc}\,x\,\texttt{in}\,M : \tau} \qquad \text{(LETCC)}$$

$$\frac{\Gamma \vdash M : \tau\ \mathit{cont} \quad \Gamma \vdash N : \tau}{\Gamma \vdash \texttt{throw}\,M\,N : \tau'} \qquad \text{(THROW)}$$

3.2 Denotational Semantics

We give here a denotation semantics for the untyped language. in preparation for establishing a Milner-type soundness theorem. The value space is given by the following domain equations:

$$\begin{aligned} Val &= Bas(BaseVal) + \\ &\quad Clsr(Val \to (Cont \to Ans)) + \\ &\quad Cnt(Cont) \\ Cont &= Val \to Ans \\ Ans &= Value(Val) + Wrong() \end{aligned}$$

An answer is either a value or a special token *wrong* representing a "run-time" type error.

Let $Env = Var \to Val$. Let the variables u, v, w, m, and n range over Val. the variable κ range over $Cont$. and the variable ρ range over Env.

The definition of the meaning function $[\![\cdot]\!] : Env \to Cont \to Ans$ is given by induction on the structure of expressions as follows:

$$\begin{aligned} [\![x]\!]\,\rho\kappa &= \kappa(\rho x) \\ [\![\lambda x.M]\!]\,\rho\kappa &= \kappa(clsr(\lambda v\kappa'.\,[\![M]\!]\,\rho[x{:=}v]\kappa')) \\ [\![MN]\!]\,\rho\kappa &= [\![M]\!]\,\rho(\lambda m.\,[\![N]\!]\,\rho(\lambda n. \\ &\qquad let\ clsr(f){=}m \\ &\qquad in\ fn\kappa\ else\ wrong)) \\ [\![\mathtt{letcc}\,x\,\mathtt{in}\,M]\!]\,\rho\kappa &= [\![M]\!]\,\rho[x{:=}\,cnt(\kappa)]\kappa \\ [\![\mathtt{throw}\,M\,N]\!]\,\rho\kappa &= [\![M]\!]\,\rho(\lambda m.\,[\![N]\!]\,\rho(\lambda n. \\ &\qquad let\ cnt(\kappa'){=}m \\ &\qquad in\ \kappa' n\ else\ wrong)) \end{aligned}$$

3.3 Soundness

To state the semantic soundness theorem. we require two definitions. one for a value to be of a given type. written $v : \tau$. and one for a continuation to accept values of a given type. written $\kappa :: \tau$. These are defined simultaneously by induction on the structure of τ as follows:

1. $v : \tau$ holds iff $v = \perp$ or
   (a) $\tau = b$ and $v = bas(w)$ with w a value of base type b.
   (b) $\tau = \tau_1 \to \tau_2$ and $v = clsr(f)$ and for all values v_1 and continuations κ_2. if $v_1 : \tau_1$ and $\kappa_2 :: \tau_2$, then $f(v_1)(\kappa_2) \neq wrong$.
   (c) $\tau = \tau_1\ cont$ and $v = cnt(\kappa)$ and $\kappa :: \tau_1$.
2. $\kappa :: \tau$ holds iff $\kappa(v) \neq wrong$ for all v such that $v : \tau$.

To check that this in fact is a proper definition. simply "expand" the definition of $\kappa :: \tau$ in the definition of $v : \tau$, and check that the membership relation is used only on subsidiary types of the given type. The relation $v : \tau$ is extended to environments pointwise: $\rho : \Gamma$ iff for all $x \in \mathrm{dom}(\Gamma)$, $\rho(x) : \Gamma(x)$.

Theorem 3.1 (Soundness) *If* $\Gamma \vdash M : \tau$ *and* $\rho : \Gamma$ *and* $\kappa :: \tau$, *then* $[\![M]\!]\rho\kappa \neq$ *wrong.*

The proof is by a straightforward induction on the structure of the typing derivation.

As usual in continuation semantics, we need to provide an initial continuation. In order to preserve the soundness of typing the initial continuation must be chosen in such a way that it yields a non-*wrong* result for an argument of the type of the expression. An obvious choice is the continuation κ_0 defined by $\kappa_0(v) = value(v)$, which is essentially the identity function. In an implementation the initial continuation might print the result value of the computation, and hence must be chosen on a case-by-case basis, according to the type of the expression being evaluated.

3.4 Observational Soundness

In contrast to languages with a "direct" semantics, the soundness theorem does not yield positive results about typing. In particular, we may not conclude that the value of an arbitrary expression of a base type (*e.g.*, *int*) yields a result of the expected form (*e.g.*, a numeral). A natural attempt to obtain such results from the soundness theorem proceeds by choosing the continuation argument to be the function which yields *wrong* except on values of the desired type. However, this overlooks the fact that in a language with continuation-passing primitives, the result of evaluating an open expression need not be given in terms of its continuation argument. This suggests that the best we can hope for is a "observational soundness" theorem that yields positive results about closed expressions of base type.

It seems clear, on the basis of operational intuitions, that evaluation of a complete program either goes wrong, or else passes a value to the initial continuation. Unfortunately these operational intuitions do not seem to transfer readily to the setting of the untyped denotational semantics given above. We give here a brief sketch of the argument. The main idea is to prove that complete programs are "non-escaping" in the sense that their denotation is either *wrong*, or is determined as a function of the initial continuation. Since the semantics of certain closed expression involves the semantics of open sub-expressions, we must in fact prove a stronger result that takes account of environments and intermediate continuations. This entails extending the "non-escaping" property to arbitrary values, and to do so appears to require an inclusive predicate argument similar to that considered by Reynolds [33]. Given that such a predicate exists, we may choose the initial continuation as discussed above, and conclude, by the non-escaping property, that well-typed closed terms of base type evaluate to values of that type. Although it seems plausible that the required predicate exists, we have not proved this, and turn instead to a more straightforward argument.

The need for an inclusive predicate argument can be traced to the rich structure of the semantic domain needed to interpret untyped programs. By exploiting the type structure of the language, the complexity of reflexive domains can be avoided (while still admitting extensions such as *fix*). The fundamental idea is to adapt the methods of Meyer and Wand [22], and make use of standard results of the typed λ-calculus to obtain the desired result. The main idea is to define the meaning of a term M in our illustrative language as the meaning of its *cps transform*, $\overline{M}$, defined as follows:

$$\begin{array}{rcl}
\overline{x} & = & \lambda\kappa.\kappa\, x \\
\overline{\lambda x.M} & = & \lambda\kappa.\kappa\,(\lambda x.\overline{M}) \\
\overline{MN} & = & \lambda\kappa.\overline{M}\,(\lambda m.\overline{N}\,(\lambda n.mn\kappa)) \\
\overline{\texttt{letcc}\, x\, \texttt{in}\, M} & = & \lambda\kappa.(\lambda x.\overline{M})\,\kappa\,\kappa \\
\overline{\texttt{throw}\, M\, N} & = & \lambda\kappa.\overline{M}\,\overline{N}
\end{array}$$

(In fact, the Standard ML of New Jersey compiler is based on a similar transformation [1].)

To relate the type of a term M to the type of its cps transform, we associate to each type τ of the illustrative language, a simple type τ^* given by

$$\begin{array}{rcl}
b^* & = & b \\
(\sigma\rightarrow\tau)^* & = & \sigma^*\rightarrow((\tau^*\rightarrow\alpha)\rightarrow\alpha) \\
(\tau\ cont)^* & = & \tau^*\rightarrow\alpha
\end{array}$$

where α is a fixed, but arbitrary, base type. The function $(\ )^*$ is extended to typing contexts pointwise: if Γ is a typing context, then $\Gamma^*(x)$ is the typing context that assigns to each variable $x \in \mathrm{dom}(\Gamma)$ the simple type $\Gamma(x)^*$.

The following lemma relates the type of a term in the illustrative language to the type of its cps transform:

Lemma 3.2 *If* $\Gamma \vdash M : \tau$, *then* $\Gamma^* \vdash^{\lambda^{\rightarrow}} \overline{M} : (\tau^*\rightarrow\alpha)\rightarrow\alpha$.

(Here $\vdash^{\lambda^{\rightarrow}}$ denotes derivability in the simply-typed λ-calculus.) The proof is a simple induction on the structure of M, taking account of the definition of $\overline{M}$.

Let $\mathcal{A}$ be any model of the simply-typed λ-calculus with carrier set A^τ for each type τ and application operation $\cdot : A^{\sigma\rightarrow\tau} \rightarrow A^\sigma \rightarrow A^\tau$. We say that an environment ρ *matches* a typing context Γ iff $\rho(x) \in A^{\Gamma(x)}$ for each $x \in \mathrm{dom}(\Gamma)$. It is a standard result that if $\Gamma \vdash^{\lambda^{\rightarrow}} M : \tau$ and ρ matches Γ, then $\mathcal{A}[\![M]\!]\rho \in A^\tau$ [25]. Hence if we define $[\![M]\!]$ to be $\mathcal{A}[\![\overline{M}]\!]$, then $[\![M]\!]\rho$ is in the set $A^{(\tau^*\rightarrow\alpha)\rightarrow\alpha}$ whenever ρ matches Γ^*. Now if M is a closed term of base type b, we may take ρ to be the empty environment, and $\alpha = b$, to obtain $[\![M]\!]\rho \in A^{(b\rightarrow b)\rightarrow b}$. Now since the identity function on type b is denotable by a λ term, and since $\mathcal{A}$ is a model of the typed λ calculus, we may apply this to id_b, the initial continuation, to obtain $[\![M]\!]\rho \cdot id_b \in A^b$. In other words, the value of M, when applied to the identity as initial continuation, is a value of base type b. In the case of cpo-based models (which are needed to interpret arbitrary recursion) with base types interpreted as flat cpo's, the result is either $\perp$, or a "true" value of the base type. Note that this argument does not extend to higher types τ since the definition of τ^* for higher types involves α, and hence we may not simply choose α to coincide with τ.

3.5 Continuations as Functions

Two alternative type systems given in the introduction rely on the representation of continuations as functions. We consider here the semantics of typing for the system based on empty types, and for the system based on regarding continuations as functions of polymorphic result type.

To express the idea that a continuation is a function that never returns, we introduce a type, `void`, with no values, and regard a continuation accepting values of type τ as a function of type $\tau \rightarrow \mathtt{void}$. This leads to the following typing rule for `letcc`:

$$\frac{\Gamma[x : \tau \rightarrow \mathtt{void}] \vdash M : \tau}{\Gamma \vdash \mathtt{letcc}\, x\, \mathtt{in}\, M : \tau} \qquad \text{(CALLCC-EMPTY)}$$

Since the result of application of a continuation is now `void`, we must introduce, in compensation, a map `ignore` witnessing the inclusion of `void` into every type τ:

$$\frac{\Gamma \vdash M : \mathtt{void}}{\Gamma \vdash \mathtt{ignore}\, M : \tau} \qquad \text{(GIVE-UP)}$$

(These two rules have the form of Pierce's Law and false elimination, respectively; see [15] for further discussion.) The expression $\mathtt{throw}\, M\, N$ is now defined as $\mathtt{ignore}(M N)$. For the polymorphic variant, the typing rule for `letcc` appears in the introduction.

The denotational semantics must be changed to reflect the representation of continuations as functions. The domain equation for values is simplified to

$$\mathit{Val} = \mathit{Bas}(\mathit{BaseVal}) + \mathit{Clsr}(\mathit{Val} \rightarrow \mathit{Cont} \rightarrow \mathit{Ans}),$$

and the semantic equations become:

$$\begin{array}{rcl}
[\![x]\!]\,\rho\kappa & = & \kappa(\rho x) \\
[\![\lambda x.M]\!]\,\rho\kappa & = & \kappa(\mathit{clsr}(\lambda v\kappa'.\, [\![M]\!]\,\rho[x{:=}v]\kappa')) \\
[\![MN]\!]\,\rho\kappa & = & [\![M]\!]\,\rho(\lambda m.\, [\![N]\!]\,\rho(\lambda n. \\
 & & \mathit{let}\ \mathit{clsr}(f){=}m \\
 & & \mathit{in}\ fn\kappa\ \mathit{else}\ \mathit{wrong})) \\
[\![\mathtt{letcc}\, x\, \mathtt{in}\, M]\!]\,\rho\kappa & = & [\![M]\!]\,\rho[x{:=}\, \mathit{clsr}(\lambda v\kappa'.\kappa v)]\kappa \\
[\![\mathtt{ignore}\, M]\!]\,\rho\kappa & = & [\![M]\!]\,\rho\kappa
\end{array}$$

A continuation is represented by a closure that ignores its continuation argument [32]. The definition of the relation $v : \tau$ remains essentially the same, ignoring the clause for continuation types. Since `void` is a base type with no proper elements, if $v : \mathtt{void}$, then $v = \perp$: there are no terminating values of type `void`.

The proof of the soundness theorem for the system with an empty type relies on two facts. First, we must show that a continuation κ regarded as a closure has type $\tau \rightarrow \mathtt{void}$ whenever $\kappa :: \tau$. Suppose that $v : \tau$ and that $\kappa' :: \mathtt{void}$. Then $(\lambda v.\lambda\kappa'.\kappa(v))\, v\, \kappa' = \kappa(v)$. Now since $\kappa :: \tau$ and $v : \tau$, it follows that $\kappa(v) \neq \mathit{wrong}$, as required. Second, we must show that $\mathtt{ignore}\, M$ may be assigned an arbitrary type whenever M is of type `void`. But if $\kappa :: \tau$, then $\kappa :: \mathtt{void}$ since $\perp : \tau$ for any τ, and hence the result follows by induction. For the case of polymorphic continuations, we need only remark that $\forall t.\tau(t)$ is defined by intersection over all monotypes, and note that the above argument shows that if $\kappa :: \tau$, then $\lambda v.\lambda\kappa'.\kappa(v) : \tau \rightarrow \tau'$ for any type τ' (not just `void`).

4 Extending the Definition of Standard ML

An operational semantics for our language presented in the relational semantics style of the formal definition of Standard ML [24] may be obtained from the denotational semantics by a process of "defunctionalization" [32] whereby closures and continuations are represented by finitary objects that are then "interpreted" on an argument, rather than simply applied to it. The resulting semantics differs substantially from the dynamic semantics of Standard ML. This may be taken as evidence that the addition of callcc to Standard ML would be a substantial change, rather than an incremental modification, to the language.

The "finitary objects" of the operational semantics are defined by the following grammar:

$$\begin{array}{rcl} V & ::= & \mathrm{BAS}(B) \mid \mathrm{CLSR}(x, M, E) \mid \mathrm{CNT}(K) \\ K & ::= & \mathrm{TOP} \mid \mathrm{FARG}(N, E, K) \mid \mathrm{APP}(V, K) \mid \\ & & \mathrm{TARG}(N, E) \mid \mathrm{THR}(V) \\ A & ::= & \mathrm{VALU}(V) \mid \mathrm{WRONG} \end{array}$$

where B is a basic value (of unspecified structure), and E is a finite function mapping variables to finitary values.

The dynamic operational semantics is defined in terms of two judgement forms. $E; K \vdash M \Rightarrow A$ and $V \vdash K \Rightarrow A$. The derivation rules for these judgements are as follows:

$$\frac{E(x) \vdash K \Rightarrow A}{E; K \vdash x \Rightarrow A} \qquad \text{(O-VAR)}$$

$$\frac{\mathrm{CLSR}(x, M, E) \vdash K \Rightarrow A}{E; K \vdash \lambda x.M \Rightarrow A} \qquad \text{(O-ABS)}$$

$$\frac{E; \mathrm{FARG}(N, E, K) \vdash M \Rightarrow A}{E; K \vdash M\,N \Rightarrow A} \qquad \text{(O-APPLY)}$$

$$\frac{E[x{=}\mathrm{CNT}(K)]; K \vdash M \Rightarrow A}{E; K \vdash \mathtt{letcc}\, x\, \mathtt{in}\, M \Rightarrow A} \qquad \text{(O-LETCC)}$$

$$\frac{E; \mathrm{TARG}(N, E) \vdash M \Rightarrow A}{E; K \vdash \mathtt{throw}\, M\, N \Rightarrow A} \qquad \text{(O-THROW)}$$

$$V \vdash \mathrm{TOP} \Rightarrow \mathrm{VALU}(V) \qquad \text{(O-TOP)}$$

$$\frac{E; \mathrm{APP}(V, K) \vdash N \Rightarrow A}{V \vdash \mathrm{FARG}(N, E, K) \Rightarrow A} \qquad \text{(O-FARG)}$$

$$\frac{E[x{=}V]; K \vdash M \Rightarrow A}{V \vdash \mathrm{APP}(\mathrm{CLSR}(x, M, E), K) \Rightarrow A} \qquad \text{(O-APP)}$$

$$\frac{E; \mathrm{THR}(V) \vdash N \Rightarrow A}{V \vdash \mathrm{TARG}(N, E) \Rightarrow A} \qquad \text{(O-TARG)}$$

$$\frac{V \vdash K \Rightarrow A}{V \vdash \text{THR}(\text{CNT}(K)) \Rightarrow A} \qquad (\text{O-THR})$$

Here the initial continuation, designated TOP, is axiomatized as the trivial insertion of values into answers. Other choices are possible.

There is an intriguing parallel between this operational semantics and a call-by-value variant of graph reduction. The idea is that the argument K in $E{:}K \vdash M \Rightarrow A$ may be thought of as a "marked" spine stack, with the marks indicating whether or not the argument position has been evaluated. Rule O-APPLY "pushes" a node onto the spine stack, marking it as having an unevaluated argument. Rules O-VAR, and O-ABS are "turning points" at which traversal of the expression reverses direction by sending explicit values back up toward the root. This traversal is carried out by the rules for interpreting a continuation. Rules O-FARG and O-TARG cover the case where the argument position is as yet unevaluated, and proceed to evaluate it, marking the node appropriately. Upon return to such a node, the actual application or throw is carried out, either by evaluating the body of the closure in the appropriate environment, or by switching contexts entirely.

The soundness theorem for typing may be proved for the operational semantics by proceeding along much the same lines as for the denotational case. First, we must augment the evaluation relation to include error checking rules that make explicit the notion of "going wrong." These are:

$$\frac{V' \neq \text{CLSR}(x.M.E)}{V \vdash \text{APP}(V'.K) \Rightarrow \text{WRONG}} \qquad (\text{O-APP-X})$$

$$\frac{V' \neq \text{CNT}(K')}{V \vdash \text{THR}(V'.K) \Rightarrow \text{WRONG}} \qquad (\text{O-THR-X})$$

We then define the relations $V : \tau$ and $K :: \tau$ more or less as before, except that instead of relying on the existence of suitable functions in the value space, we appeal directly to the operational semantics.

1. $V : \tau$ holds iff
   (a) $\tau = b$ and $V = \text{VALU}(W)$ with W a value of type b;
   (b) $\tau = \tau_1 \rightarrow \tau_2$ and $V = \text{CLSR}(x, M, E)$ and for every V_1 and K_2, if $V_1 : \tau_1$, $K_2 :: \tau_2$, and $E[x = V_1]; K_2 \vdash M \Rightarrow A$, then $A \neq \text{WRONG}$;
   (c) $\tau = \tau_1$ *cont* and $V = \text{CNT}(K)$ and $K :: \tau_1$.
2. $K :: \tau$ holds iff for every V such that $V : \tau$, if $V \vdash K \Rightarrow A$, then $A \neq \text{WRONG}$.

The soundness theorem for the operational semantics is as follows:

Theorem 4.1 *If* $\Gamma \vdash M : \tau$ *and* $E : \Gamma$ *and* $K :: \tau$ *and* $E; K \vdash M \Rightarrow A$, *then* $A \neq \text{WRONG}$.

In contrast to the domain-theoretic semantics, the proof of the soundness theorem is significantly complicated by the introduction of fixed-point operators for defining recursive functions; see [23] for a careful discussion of a closely-related problem in the setting of natural semantics.

5 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 Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, TX, January 1989.

[2] Andrew W. Appel and David MacQueen. Standard ML of New Jersey Reference Manual. (in preparation), 1990.

[3] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, Sophia–Antipolis, France, June 1985.

[4] 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.

[5] Eric C. Cooper. An overview of the FOX project. Talk at Bell Laboratories, July 1990.

[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 No. 182, Indiana University Computer Science Department, 1985.

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

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

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

[12] 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.

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

[14] Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. Programming with continuations. *Program Transformations and Programming Environments*, edited by P. Pepper. Springer-Verlag, Heidelberg, 1985. 263–274.

[15] Timothy Griffin. A formulas-as-types notion of control. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990, pages 47–58.

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

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

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

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

[20] J. Roger Hindley. The completeness theorem for typing λ terms. *Theoretical Computer Science*, 22:127–134, 1983.

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

[22] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda calculi (summary). In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer-Verlag, 1985.

[23] Robin Milner and Mads Tofte. Co-induction in relational semantics. Technical Report ECS-LFCS-88-65, Laboratory for the Foundations of Computer Science, Edinburgh University, Edinburgh, October 1988.

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

[25] John C. Mitchell. Type systems for programming languages. Technical Report STAN-CS-89-1277, Stanford University, 1989.

[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] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, 1981.

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

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

[30] John Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Computer Science Department, Cornell University, Ithaca, NY, August 1990.

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

[32] 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.

[33] John C. Reynolds. On the relation between direct and continuation semantics. In *Proc. ICALP '74*, volume 14 of *Lecture Notes in Computer Science*, pages 141–156. Springer-Verlag, 1974.

[34] 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.

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

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

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

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

[39] 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.

[40] 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.

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

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