Computer Science 15-312
First-Class Continuations

Robert Harper

Spring, 2000

1 Introduction

The treatment of exceptions in terms of a handler and control stack relies on the *reification* of control stacks as values that can be pushed on the handler stack. At first glance this appears to be a rather heavyweight operation that would involve copying the entire control stack when establishing a handler, and restoring it when raising an exception. However, we observed that the machine satisfies a crucial invariant, namely that the saved control stack is always an initial segment of the current control stack. This allows us to reify a control stack as a "finger" in the control stack, and to install it by popping the stack back to the finger. This is a formal justification of an implementation based on the `setjmp` and and `longjmp` constructs of the C language. Unlike `setjmp` and `longjmp`, the exception mechanism is completely safe — it is impossible to return past the "finger" yet later attempt to "pop" the control stack to that point. In C the fingers are kept as addresses (pointers) in memory, and there is no discipline for ensuring that the set point makes any sense when invoked later in a computation.

The idea of reification of control stacks can be taken a step further, by allowing them to be passed as values within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *first-class continuations*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. In contrast to set points in C first-class continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

How is this achieved? The key to implementing first-class continuations is to arrange that control stacks are *persistent* data structures, just like any other data structure in ML that does not involve mutable references. By a persistent data structure we mean one for which operations on it yield a "new" version of the data structure without disturbing the old version. For example, lists in ML are persistent in the sense that if we cons an element to the front of a list we do not thereby destroy the original list, but rather yield a new list with an additional element at the front, retaining the possibility of using the old list for other purposes. In this sense persistent data structures allow time travel — we can easily switch between several versions of a data structure without regard to the temporal order in which they were created. This is in sharp contrast to more familiar *ephemeral* data structures for which operations such as insertion of an element irrevocably mutate the data structure, preventing any form of time travel.

Returning to the case in point, the standard implementation of a control stack is as an ephemeral data structure, a pointer to a region of mutable storage that is overwritten whenever we push a frame. This makes it impossible to maintain an "old" and a "new" copy of the control stack at the same time, making time travel impossible. If, however, we represent the control stack as a persistent data structure, then we can easily reify a control stack by simply binding it to a variable, and continue working. If we wish we can easily return to that control stack by referring to the variable that is bound to it. This is achieved in practice by representing the control stack as a list of frames in the heap so that the persistence of lists can be extended to control stacks. While we will not be specific about implementation strategies in this note, it should be born in mind when considering the semantics outlined below.

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor. Later in the course we will show how to build a threads package for concurrent programming using continuations.

2 Informal Overview of Continuations

We will extend MinML with the type τ `cont` of continuations accepting values of type τ. A continuation will, in fact, be a control stack of type τ `stack`, but rather than expose this representation to the programmer, we will regard τ `cont` as an abstract type supporting two operations, `letcc` x `in` e and `throw` e_1 `to` e_2.[1]

Informally, evaluation of `letcc` x `in` e binds the *current continuation*[2] to x and evaluates e. The current continuation as, as we've discussed, a reification of the current control stack, which represents the current point in the evaluation of the program. The type of x is τ `cont`, where τ is the type of e. The intuition is that the current continuation is the point to which e returns when it completes evaluation. Consequently, the control stack expects a value of type τ, which then determines how execution proceeds. Thus x is bound to a stack expecting a value of type τ, that is, a value of type τ `cont`. Note that this is the *only* way to obtain a value of type τ `cont`; there are no expressions that evaluate to continuations. (This is similar to our treatment of references — values of type τ `ref` are locations, but locations can only be obtained by evaluating a `ref` expression.)

We may "jump" to a saved control point by *throwing* a value to a continuation, written `throw` e_1 `to` e_2. The expression e_2 must evaluate to a τ_1 `cont`, and e_1 must evaluate to a value of type τ_1. The current control stack is abandoned in favor of the reified control stack resulting from the evaluation of e_2; the value of e_1 is then passed to that stack.

Here is a simple example, written in Standard ML notation. The idea is to multiply the elements of a list, short-circuiting the computation in case `0` is encountered. Here's the code:

```
fun mult_list (l:int list):int =
    callcc
      (fn ret =>
        let fun mult (nil) = 1
              | mult (0::_) = throw ret 0
              | mult (n::l) = n * mult (l)
         in mult (l) end)
```

Ignoring the `callcc` for the moment, the body of `mult_list` is a `let` expression that defines a recursive procedure `mult`, and applies it to the argument

[1] Close relatives of these primitives are available in SML/NJ in the following forms: for `letcc` x `in` e, write `callcc (fn` x `=>` e`)`, and for `throw` e_1 `to` e_2, write `throw (`e_2`,`e_1`)`.

[2] Hence the name "`letcc`".

of mult_list. The job of mult is to return the product of the elements of the list times the value of the accumulator; by calling mult with l and 1, we obtain the product of the elements of l. Ignoring the second line of mult, it should be clear why and how this code works.

Now let's consider the second line of mult, and the outer application of callcc. Intuitively, the purpose of the second line of mult is to short circuit the multiplication, returning 0 immediately in the case that a 0 occurs in the list. This is achieved by throwing the value 0 (the final answer) to the continuation bound to the variable ret. This variable is bound by the call to callcc surrounding the body of mult_list. What continuation is it? It's the continuation that runs upon completion of the body of mult_list. This continuation would be executed in the case that no 0 is encountered and evaluation proceeds normally. In the unusual case of encountering a 0 in the list, we branch directly to the return point, passing the value 0, effecting an early return from the procedure with result value 0.

Here's a slicker formulation of the same function:

```
fun mult_list l =
    let fun mult nil ret = 1
          | mult (0::_) ret = throw ret 0
          | mult (n::l) ret = n * mult l ret
     in callcc (mult l) end
```

Here the inner loop is parameterized by the return continuation for early exit. The multiplication loop is obtained by calling mult with the current continuation at the exit point of mult_list so that throws to ret effect an early return from mult_list, as desired.

Exercise 1
Study this example carefully to be sure you understand why it works!

Let's look at another example: given a continuation k of type τ cont and a function f of type $\tau'\to\tau$, return a continuation k' of type τ' cont with the following behavior: throwing a value v' of type τ' to k' throws the value $f(v')$ to k. This is called *composition of a function with a continuation.* We wish to fill in the following template:

```
fun compose (f:τ'->τ,k:τ cont):τ' cont = ...
```

The function compose will have type

```
((τ' -> τ) * τ cont) -> τ' cont
```

Exercise 2
This is a very difficult programming problem! But please take a few moments to try to solve it before reading on. The solution is very instructive, but is, for most people, rather hard to think up.

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` f`(...) to` k. This is the continuation that, when given a value v', applies f to it, and throws the result to k. We can seize this continuation using `letcc`, writing

`throw` f`(letcc` x:τ' `cont in ...) to` k

At the point of the ellipsis the variable x is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code, in SML/NJ notation:

```
fun compose (f, k) =
    callcc
      (fn return =>   (* return point *)
          throw k (f (callcc (fn k' => throw return k'))))
```

The type of `return` is τ' `cont cont`, a continuation expecting a continuation expecting a value of type τ'!

We can do without first-class continuations by "rolling our own". The idea is that we can perform (by hand or automatically) a systematic program transformation in which a "copy" of the control stack is maintained as a function, called a continuation. Every function takes as an argument the control stack to which it is to pass its result by applying given stack (represented as a function) to the result value. Functions never return in the usual sense; they pass their result to the given continuation. Programs written in this form are said to be in *continuation-passing style*, or *CPS* for short.

Here's the code to multiply the elements of a list (without short-circuiting) in continuation-passing style:

```
fun cps_mult nil k = k 1
  | cps_mult (n::l) k = cps_mult l (fn r => k (n * r))
fun mult l = cps_mult l (fn r => r)
```

It's easy to implement the short-circuit form by passing an additional continuation, the one to invoke for short-circuiting the result:

```
fun cps_mult_list l k =
    let fun cps_mult nil k0 k = k 1
      | fun cps_mult (0::_) k0 k = k0 0
      | fun cps_mult (n::l) k0 k = cps_mult k0 l (fn p => k (n*p))
    in cps_mult l k k end
```

The continuation `k0` never changes; it is always the return continuation for `cps_mult_list`. The argument continuation to `cps_mult_list` is duplicated on the call to `cps_mult`.

Observe that the type of the first version of `cps_mult` becomes

$$\texttt{int}\,\texttt{list} \to (\texttt{int} \to \alpha) \to \alpha,$$

and that the type of the second version becomes

$$\texttt{int}\,\texttt{list} \to (\texttt{int} \to \alpha) \to (\texttt{int} \to \alpha) \to \alpha,$$

These transformations are representative of the general case.

3 Semantics of Continuations

The informal description of evaluation is quite complex, as you no doubt have observed. Here's an example where a formal semantics is *much* clearer, and can serve as a useful guide for understanding how all of this works. The semantics is suprisingly simple and intuitive.

First, the abstract syntax. We extend the language of MinML types with continuation types of the form $\tau\,\texttt{cont}$. We extend the language of MinML expressions with these additional forms:

$$e \quad ::= \quad \ldots \mid \texttt{letcc}\,x\,\texttt{in}\,e \mid \texttt{throw}\,e_1\,\texttt{to}\,e_2 \mid K$$

In the expression $\texttt{letcc}\,x\,\texttt{in}\,e$ the variable x is bound in e. As usual we rename bound variables implicitly as convenient. We include control stacks K as expressions for the sake for the sake of the dynamic semantics, much as we included locations as expressions when considering reference types. We define continuations thought of as expressions to be values:

$$\frac{K\ \mathsf{stack}}{K\ \mathsf{value}}$$

Stacks are as defined for the C machine, extended with these additional frames:

$$\frac{e_2 \ \mathsf{expr}}{\mathtt{throw}\,\Box\,\mathtt{to}\,e_2 \ \mathsf{frame}} \qquad \frac{v_1 \ \mathsf{value}}{\mathtt{throw}\,v_1\,\mathtt{to}\,\Box \ \mathsf{frame}}$$

Second, the static semantics. The typing rules governing the continuation primitives are these:

$$\frac{\Gamma[x{:}\tau\,\mathtt{cont}] \vdash e : \tau}{\Gamma \vdash \mathtt{letcc}\,x\,\mathtt{in}\,e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\,\mathtt{cont}}{\Gamma \vdash \mathtt{throw}\,e_1\,\mathtt{to}\,e_2 : \tau'}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call. The typing rule for continuation values is as follows:

$$\frac{\vdash K : \tau\,\mathtt{stack}}{\Gamma \vdash K : \tau\,\mathtt{cont}}$$

That is, a continuation value K has type $\tau\,\mathtt{cont}$ exactly if it is a stack accepting values of type τ. This relation is as defined in our treatment of exceptions, extended to include the additional frames mentioned above.

Finally, the dynamic semantics. We use the C machine as a basis. We extend the language of expressions to include control stacks K as values. Like locations, these arise only during execution; there is no explicit notation for continuations in the language. The key transitions are as follows:

$$\frac{}{(K, \mathtt{letcc}\,x\,\mathtt{in}\,e) \mapsto (K, [K/x]e)} \qquad \frac{}{(\mathtt{throw}\,v\,\mathtt{to}\,\Box \rhd K, K') \mapsto (K', v)}$$

In addition we specify the order of evaluation of arguments to `throw`:

$$\frac{}{(K, \mathtt{throw}\,e_1\,\mathtt{to}\,e_2) \mapsto (\mathtt{throw}\,\Box\,\mathtt{to}\,e_2 \rhd K, e_1)}$$

$$\frac{}{(\mathtt{throw}\,\Box\,\mathtt{to}\,e_2 \rhd K, v_1) \mapsto (\mathtt{throw}\,v_1\,\mathtt{to}\,\Box \rhd K, e_2)}$$

Notice that evaluation of `letcc` *duplicates* the control stack, and that evaluation of `throw` *eliminates* the current control stack.

Exercise 3

Simulate the evaluation of `compose (f, k)` *on the empty stack. Observe that the control stack substituted for x is*

$$\mathtt{apply}(f, \Box) \rhd \mathtt{throw}\,\Box\,\mathtt{to}\,k \rhd \bullet$$

This stack is returned from `compose`*. Next, simulate the behavior of throwing a value v' to this continuation. Observe that the above stack is reinstated and that v' is passed to it.*

The safety of this extension of MinML may be established using familiar techniques. First we must define well-formedness for machine states. As before, we define (K, e) ok iff $\vdash K : \tau\ \texttt{stack}$ and $\vdash e : \tau$.

The preservation theorem is stated as follows:

Theorem 4 (Preservation)
If (K, e) ok and $(K, e) \mapsto (K', e')$, then (K', e') ok.

Proof: The proof is by induction on evaluation. The verification is left as an exercise. ∎

To establish progress we need the following extension to the canonical forms lemma:

Lemma 5 (Canonical Forms)
If $\vdash v : \tau\ \texttt{cont}$, then $v = K$ for some control stack K such that $\vdash K : \tau\ \texttt{stack}$.

Finally, progress is stated as follows:

Theorem 6 (Progress)
If (K, e) ok then either $K = \bullet$ and e value, or there exists K' and e' such that $(K, e) \mapsto (K', e')$.

Proof: By induction on typing. The verification is left as an exercise. ∎

4 Coroutines

Some problems are naturally implemented using *coroutines*, two (or more) routines that interleave their execution by an explicit hand-off of control from one to the other. In contrast to conventional sub-routines neither routine is "in charge", with one calling the other to execute to completion. Instead, the control relationship is symmetric, with each yielding control to the other during excecution.

A classic example of coroutining is provided by the producer-consumer model of interaction. The idea is that there is a common, hidden resource that is supplied by the producer and utilized by the consumer. Production of the resource is interleaved with its consumption by an explicit hand-off from producer to consumer. Here is an outline of a simple producer-consumer relationship:

```
val buf : int ref = ref 0
fun produce (n:int, cons:state) =
    (buf := n; produce (n+1, resume cons))
fun consume (prod:state) =
    (print (!buf); consume (resume prod))
```

There the producer and consumer share an integer buffer. The producer fills it with successive integers; the consumer retrieves these values and prints them. The producer yields control to the consumer after filling the buffer; the consumer yields control to the producer after printing its contents. Since the handoff is explicit, the producer and consumer run in strict synchrony, alternating between production and consumption.

The key to completing this sketch is to detail the handoff protocol. The overall idea is to represent the state of a coroutine by a continuation, the point at which it should continue executing when it is resumed by another coroutine. The function `resume` captures the current continuation and throws it to the argument continuation, transferring control to the other coroutine and, simultaneously, informing it how to resume the caller. This means that the state of a coroutine is a continuation accepting the state of (another) coroutine, which leads to a recursive type. This leads to the following partial solution:

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

All that remains is to initialize the coroutines. It is natural to start by executing the producer, but arranging to pass it a coroutine state corresponding to the consumer. This can be achieved as follows:

```
fun run () =
    consume (callcc (fn k : state cont => produce (0, S k)))
```

Because of the call-by-value semantics of function application, we first seize the continuation corresponding to passing an argument to `consume`, then invoke `produce` with initial value `0` and this continuation. When `produce` yields control, it throws its state to the continuation that invokes `consume` with that state, at which point the coroutines have been initialized — further hand-off's work as described earlier.

This is, admittedly, a rather simple-minded example. However, it illustrates an important idea, namely the symmetric hand-off of control between routines. The difficulty with this style of programming is that the hand-off protocol is "hard wired" into the code. The producer yields control to the consumer, and *vice versa*, in strict alternating order. But what if there are multiple producers? Or multiple consumers? How would we handle priorities among them? What about asynchronous events such as arrival of a network packet or completion of a disk I/O request?

An elegant solution to these problems is to generalize the notion of a coroutine to the notion of a *user-level thread*. As with coroutines, threads enjoy a symmetric relationship among one another, but, unlike coroutines, they do not explicitly hand off control amongst themselves. Instead threads run as coroutines of a *scheduler* that mediates interaction among the threads, deciding which to run next based on considerations such as priority relationships or availability of data. Threads yield control to the scheduler, which determines which other thread should run next, rather than explicitly handing control to another thread.

Here is a simple interface for a user-level threads package:

```
signature THREADS = sig
  exception NoMoreThreads
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

The function `fork` is called to create a new thread executing the body of the given function. The function `yield` is called to cede control to another thread, selected by the thread scheduler. The function `exit` is called to terminate a thread.

User-level threads are naturally implemented as continuations. A thread is a value of type `unit cont`. The scheduler maintains a queue of threads that are ready to execute. To dispatch the scheduler dequeues a thread from the ready queue and invokes it by throwing `()` to it. Forking is implemented by creating a new thread. Yielding is achieved by enqueueing the current thread and dispatching; exiting is a simple dispatch, abandoning the current thread entirely. This implementation is suggestive of a slogan suggested by Olin Shivers: "A thread is a trajectory through continuation space". During its lifetime a thread of control is represented by a succession of continuations that are enqueued onto and dequeued from the ready queue.

Here is a simple implementation of threads:

```
structure Threads :> THREADS = struct
    open SMLofNJ.Cont
    exception NoRunnableThreads
    type thread = unit cont
    val readyQueue : thread Queue.queue = Queue.mkQueue()
    fun dispatch () =
        let
            val t = Queue.dequeue readyQueue
                    handle Queue.Dequeue => raise NoRunnableThreads
        in
            throw t ()
        end
    fun exit () = dispatch()
    fun enqueue t = Queue.enqueue (readyQueue, t)
    fun fork f =
        callcc (fn parent => (enqueue parent; f (); exit()))
    fun yield () =
        callcc (fn parent => (enqueue parent; dispatch()))
end
```

Using the above thread interface we may implement the simple producer-consumer example as follows:

```
structure Client = struct
    open Threads
    val buffer : int ref = ref (~1)
    fun producer (n) =
        (buffer := n ; yield () ; producer (n+1))
    fun consumer () =
        (print (Int.toString (!buffer)); yield (); consumer())
    fun run () =
        (fork (consumer); producer 0)
end
```

This example is excessively naïve, however, in that it relies on the strict FIFO ordering of threads by the scheduler, allowing careful control over the order of execution. If, for example, the producer were to run several times in a row before the consumer could run, several numbers would be omitted from the output.

Here is a better solution that avoids this problem (but does so by "busy waiting"):

```
structure Client = struct
    open Threads
    val buffer : int option ref = ref NONE
    fun producer (n) =
        (case !buffer
           of NONE => (buffer := SOME n ; yield() ; producer (n+1))
            | SOME _ => (yield (); producer (n)))
    fun consumer () =
        (case !buffer
           of NONE => (yield (); consumer())
            | SOME n =>
              (print (Int.toString n); buffer := NONE; yield(); consumer()))
    fun run () =
        (fork (consumer); producer 0)
end
```

There is much more to be said about threads! We will return to this later in the course. For now, the main idea is to give a flavor of how first-class continuations can be used to implement a user-level threads package with very little difficulty. A more complete implementation is, of course, somewhat more complex, but not much more. We can easily provide all that is necessary for sophisticated thread programming in a few hundred lines of ML code.