15-312 Foundations of Programming Languages
Continuations

Daniel Spoonhower (spoons+@cs)
Edited by Jason Reed (jcreed+@cs)

October 6, 2004

1 Continuations

We began discussion of continuations last week in lecture; we will continue today with a pair of more detailed examples, both borrowed from Harper's notes.

1.1 Review

Recall the static semantics of our constructs for manipulating continuations.

$$\frac{\Gamma, x{:}\tau\ \mathsf{cont} \vdash e : \tau}{\Gamma \vdash \mathtt{callcc}\, x \,\mathtt{=>}\, e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\ \mathsf{cont}}{\Gamma \vdash \mathtt{throw}[\tau]\, e_1\, \mathtt{to}\, e_2 : \tau}$$

Remember that `callcc` binds the current continuation to a variable and then continues evaluation, while `throw` evaluations an expression and then continues at the point of evaluation marked by the second expression.

1.2 Short Circuiting Evaluation

Consider the following function that computes the product of the elements in an integer list.

```
fun mul(l : int list) : int =>
  case l
    of nil => 1
     | x::l => x * mul l
```

Note that if the list contains the element 0, this function might perform a significant amount of extra work, particularly if the 0 appears near the beginning of the list. We might consider a *short-circuited* version of multiplication, one where we stop inspecting the remainder of the list once we encounter a 0.

```
fun mul(l : int list) : int =>
  case l
    of nil => 1
     | x::l => if x = 0 then 0
               else x * mul l fi
```

If we expect a list of length n to have exactly one 0 (distributed uniformly), we've reduced the (expected) number of recursive calls by $n/2$. We are still, however, performing $n/2$ multiplications (again, expected case), each of whose result will be 0. We'd like to jump out the entire sequence of recursive calls, not just the current one.

For reasons that will become clear in a moment, we first transform the function by η-expansion:

```
fn l : int list =>
  let mul = fun mul(l : int list) : int =>
    case l
      of nil => 1
       | x::l => if x = 0 then 0
                 else x * mul l fi
  in
    mul l
  end
```

Now, using the `callcc` and `throw` constructs from above, we can write

```
fn l : int list =>
  callcc ret =>
    let mul = fun mul(l : int list) : int =>
      case l
        of nil => 1
         | x::l => if x = 0 then throw 0 to ret
                   else x * mul l fi
    in
      mul l
    end
  end
```

In this example, we are throwing a value *backward* to a previous point in evaluation, and moreover, we don't really use `ret` for anything particularly interesting. We could have easily written a similar short-circuiting function using exceptions. Next, we'll see an example where that is definitely *not* the case.

1.3 Composition

Remember that continuations are *values*: even though we can't write a value of type τ `cont` in the concrete syntax, they may be manipulated just like any other value.

We'd like to write a function `compose` that combines a function with a continuation, resulting in a new continuation. Specifically, a function with the following type. (Why does this make sense?)

$$\texttt{compose : (}\tau'\texttt{ -> }\tau\texttt{) -> }\tau\texttt{ cont -> }\tau'\texttt{ cont}$$

We begin as follows:

$$\texttt{fn f : }\tau'\texttt{ -> }\tau\texttt{ => fn k : }\tau\texttt{ cont =>}$$

Now what do we do? Let's inspect the types and see what we *can* do.

`throw[`τ'`]` (something of type τ) `to k` (...this will have type τ'...)
`f` (applied to something of type τ') (...this will have type τ...)

Finally, we know we want to return a value of type τ' `cont`, and there is only one way to create such a value:

`callcc k' =>` (something of type τ') `end`

Let's start from the end and work backwards. The `k'` above holds the value we'd like to return, but we can't simply write

```
callcc k' => k' end
```

(Remember from lecture that such an expression is not well-typed.)

So how else can we save the continuation (and return it later)? Well, the only other thing we can *do* with a continuation is to `throw` it!

`callcc k' => throw[`τ'`] k' to ? end`

(Why do we give the `throw` expression type τ'?) Of course, we need somewhere to throw this continuation, so let's use another `callcc`.

`callcc ret => ... callcc k' => throw[`τ'`] k' to ret end end`

Now that we have captured the continuation we want, let's go back and consider what we'd do if someone actually threw to it. First we'd apply `f`:

`callcc ret => ... f (callcc k' => throw[`τ'`] k' to ret end) end`

What remains? We have only to `throw` some value of type τ to `k`. The result of the application of `f` is just such value.

```
fn f : τ' -> τ => fn k : τ cont =>
  callcc ret =>
    throw[τ' cont] (f (callcc k' => throw[τ'] k' to ret end))
    to k
  end
```

(Convince yourself that this function typechecks. What's the type of `ret`? (τ' `cont cont`) Finally, does `compose` ever return? Did we ever expect it to?)

Clearly, we could not accomplish a feat such as `compose` with exceptions!

1.4 Threads

You saw briefly in lecture how to make a 'cooperative' threading library using callcc. We'll review it.

The primitives we want are described by the signature

```
sig
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

`fork` takes a function and starts executing it in a new thread, putting the current thread to sleep. When it wakes up, it will continue executing from where the fork ended. `yield` simply puts the current thread to sleep, which will continue executing from after the yield when it wakes back up again. `exit` terminates the current thread.

What `callcc` is used for is to express the idea of 'where I should start executing again once I'm woken up'. `throw` is used to actually wake up a thread.

The signature of the queue that holds all our threads looks like this:

```
sig
  type 'a queue

  val new : unit -> 'a queue
  val enqueue : 'a queue * 'a -> unit
  val dequeue : 'a queue -> 'a
  val clear : 'a queue -> unit
end
```

Pretty straightforward.

Now we can start coding up the thread library itself. What a thread is going to be is a continuation that receives no data in particular, i.e. unit:

```
structure T :> THREADS =
struct
```

```
  type thread = unit cont
...
```

Initializing the queue is trivial:

```
...
  val readyQueue : thread Q.queue = Q.new ()
...
```

Now we're going to need a pair of functions that accomplishe 'start running the next runnable thread.' and 'make the following a runnable thread'. The first means waking it up by throwing it a value of the type of continuation that it is, which is unit, so we throw it `()`. The second is simply enqueueing the given thread, which is a unit continuation: it must have come from a `callcc` somewhere.

```
...
  fun dispatch () =
      let
val t = Q.dequeue readyQueue
                handle Q.Dequeue => raise NoRunnableThreads
      in
throw t ()
      end
  fun enq t = Q.enqueue (readyQueue, t)
...
```

Now here comes the actual guts of the thread library. Look how simple they are! Notice how `yield` is a special case of `fork`: It's the forking-off of the empty job.

```
...
  fun exit () = dispatch ()
  fun fork f = callcc (fn parent => (enq parent; f (); exit ()))
  fun yield () = callcc (fn parent => (enq parent; dispatch ()))
end
```

The 'current thread' is represented by the actual current execution of the ML program. All the items in our queue are inactive threads, or, more accurately, the entry points into the current execution point of all our inactive threads, represented as a continuation.

So `exit` works by removing something from the queue transferring control to it. The current thread, the current computation is abandoned in favor of the next thread that wants to run. Nothing else is pushed onto the queue, so the current thread will never see the light of day again. This happens even if an `exit` is buried deep inside a thread's code! (Compare this to raising an exception or calling `exit` in `C` code) A thread also ends if it just happens to 'fall off the end' of its code by returning `()`, but `exit` is more powerful.

What `fork` does is enqueues the current position, and starts executing its argument `f`. We first do a `callcc` to capture our current continuation, enqueue it onto the runnable-thread queue, call `f` on the unit value, and if `f` ever finishes (it may not!) we pick something off the runnable queue to run instead.

What `yield` does is very similar. It just doesn't insert any new computation, but goes directly to the runnable queue to find some new work to do. This just has the obvious effect of letting other work in other threads get done before coming back to the current computation.

1.5 Other Stuff if I Have Time

CallCC and classical logic? A favorite topic of mine, may not be one of yours.