1 First-Class Continuations

Some languages expose continuations as first-class values. Examples of such languages include Scheme and SML/NJ. In the latter, there is a module that defines a continuation type α cont representing a continuation expecting a value of type α. There are two functions for manipulating continuations:

- callcc : $(\alpha\,\mathsf{cont} \to \alpha) \to \alpha$ (callcc f) passes the current continuation to the function f
- throw : $\alpha\,\mathsf{cont} \to \alpha \to \beta$ (throw k v) sends the value v to the continuation k.

The call (callcc f) passes the current continuation corresponding to the evaluation context of the callcc itself to the function f of type $\alpha\,\mathsf{cont} \to \alpha$. The current continuation k is of type $\alpha\,\mathsf{cont}$. When called with this continuation, f may evaluate to a value of type α, and that is the value of the expression (callcc f) that called it. However, the continuation k passed to f may be called with a value v of type α by (throw k v) with the same effect. It is up to the evaluation context of the callcc to determine which. Thus (callcc $\lambda k.\,3$) and (callcc $\lambda k.\,\mathsf{throw}\,k\,3$) have the same effect.

1.1 Semantics of First-Class Continuations

Using the translation approach we introduced earlier, we can easily describe these mechanisms. Suppose we represent a continuation value for the continuation k by tagging it with the integer 7. Then we can translate callcc and throw as follows:

$$
\begin{aligned}
[\![\mathsf{callcc}\,e]\!]\,\rho k &= [\![e]\!]\,\rho(\mathsf{check\text{-}fun}\,(\lambda f.\,f\,(7,k)\,k)) \\
[\![\mathsf{throw}\,e_1\,e_2]\!]\,\rho k &= [\![e_1]\!]\,\rho(\mathsf{check\text{-}cont}\,(\lambda k'.\,[\![e_2]\!]\,\rho k'))
\end{aligned}
$$

The key to the added power is the non-linear use of k in the callcc rule. This allows k to be reused any number of times.

1.2 Implementing Threads with Continuations

Once we have first-class continuations, we can use them to implement all the different control structures we might want. We can even use them to implement (non-preemptive) threads, as in the following code that explains how concurrency is handled in languages like OCaml and Concurrent ML:

```
type thread = unit cont

let ready : thread queue = new_queue (* a mutable FIFO queue *)
let enqueue t = insert ready t
let dispatch() = throw (dequeue ready) ()

let spawn (f : unit -> unit) : unit =
  callcc (fun k -> (enqueue k; f(); dispatch()))
let yield() : unit = callcc (fun k -> enqueue k; dispatch())
```

The interface to threads consists of the functions spawn and yield. The spawn function expects a function f containing the work to be done in the newly spawned thread. The yield function causes the current thread to relinquish control to the next thread on the ready queue. Control also transfers to a new thread when one thread finishes evaluating. To complete the implementation of this thread package, we just need a queue implementation. CML has preemptive threads, in which threads implicitly yield automatically after a certain amount of time; this requires just a little help from the operating system.