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 $\alpha$ cont representing a continuation expecting a value of type $\alpha$. There are two functions for manipulating continuations:

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

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

$$\llbracket \text{callcc } e \rrbracket \rho k = \llbracket e \rrbracket \rho (\text{check-fun} (\lambda f. f (7, k) \ k))$$
$$\llbracket \text{throw } e_1 \ e_2 \rrbracket \rho k = \llbracket e_1 \rrbracket \rho (\text{check-cont} (\lambda k'. \llbracket e_2 \rrbracket \rho k'))$$

The key to the added power is the non-linear use of $k$ in the $\text{callcc}$ 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:

```ocaml
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())
```

1
The interface to threads consists of the functions \texttt{spawn} and \texttt{yield}. The \texttt{spawn} function expects a function \( f \) containing the work to be done in the newly spawned thread. The \texttt{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.