

1 State in CPS

...

2 Exposing continuations in the language

Continuations are great for talking about semantics, but they are also useful when programming.

2.1 `setjmp` and `longjmp`

`setjmp` is a C function which takes a pointer to a buffer. Its operation is to save the state of all registers (including the program counter) into the specified buffer and return 0. `longjmp`, also a C function, takes as an argument a pointer to a buffer (which is presumed to be a buffer which has already been filled with a previous call to `setjmp`) and a value. When invoked, it restores all of the registers to the values saved in the buffer and returns the value passed in to the point in the program where `setjmp` was called (in effect, the program resumes executing right where `setjmp` was called, except the call will return the value passed in to `longjmp`). These functions can be used for error handling. `setjmp` is called before code that may result in an error. If an error occurs in computation, `longjmp` is called in order to restore initial state and handle the error. For instance:

```
if (setjmp(&jmpbuf))
    // error handling code goes here
else
    do computation();
    // if error occurs in compute(), call
    // longjmp(jmpbuf, e), where e is the error code
```

The functions `setjmp` and `longjmp` can be translated using continuations as follows:

$$\begin{aligned} \llbracket \text{setjmp } e \rrbracket \rho k &= \llbracket e \rrbracket \rho (\text{CHECK-LOC } (\lambda \sigma. k (\text{INT } 0) (\text{UPDATE } \sigma \ l \ (\text{CONT } k)))) \\ \llbracket \text{longjmp } e \ e' \rrbracket \rho k &= \llbracket e \rrbracket \rho (\text{CHECK-LOC } (\lambda l. \llbracket e' \rrbracket \rho \\ &\quad (\lambda v \sigma. \text{CHECK-CONT}(\text{LOOKUP } \sigma \ l) (\lambda k'. k' \ v \ \sigma)))) \end{aligned}$$

The translation of `setjmp` stores a continuation at a new location, while the translation of `longjmp` restores a continuation from a program location. This is roughly equivalent to restoring the registers and program state of the executing program.

2.2 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 a continuation type α `cont`, representing a continuation expecting a value of type α . There are two functions for manipulating continuations:

callcc: $(\alpha \text{ cont} \rightarrow \alpha) \rightarrow \alpha$

(`callcc` f) passes the current continuation to the function f .

throw: $\alpha \text{ cont} \rightarrow \alpha \rightarrow \beta$

(**throw** k v) sends the value v to the continuation k .

Because `callcc` passes the current continuation, corresponding to the evaluation context of the `callcc` itself, invocation of the passed continuation makes the `callcc` expression itself seem to return again. It's up to the evaluation context of the `callcc` to decide whether it's seeing the original return from f or a later invocation of the passed continuation.

2.3 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:

$$\begin{aligned} \llbracket \text{callcc } e \rrbracket \rho k &= \llbracket e \rrbracket \rho (\text{CHECK-FUN}(\lambda f. f (7, k) k)) \\ \llbracket \text{throw } e_1 e_2 \rrbracket \rho k &= \llbracket e_1 \rrbracket \rho (\text{CHECK-CONT}(\lambda k'. \llbracket e_2 \rrbracket \rho k')) \end{aligned}$$

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.

2.4 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 SML/NJ-like code that explains how Concurrent ML (CML) is implemented:

```
type thread = unit cont
ready: thread queue = new_queue (* a mutable FIFO queue *)
enqueue(t) = insert ready t
dispatch() = throw (dequeue ready) ()
spawn(f: unit→unit): unit =
  callcc( fn(k) => (enqueue k; f(); dispatch()))
yield(): unit = callcc (fn(k) => enqueue k; dispatch())
```

The interface to threads is 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.