Callcc and Coroutines

November 13, 2001

CS 131: Programming Languages

Primitive Continuations

- Some functional languages allow programs to capture and manipulate run-time continuations
  - i.e., continuations as ordinary values
  - Required in the Scheme language definition
    - Long history of "first-class everything" in Scheme.
  - Also available in SML/NJ.

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw  : 'a cont -> 'a -> 'b
```

`callcc` is short for `call-with-current-continuation`

Continuations in SML/NJ

- A value of type `'a cont` is a computation waiting to resume when it is given a value of type `'a`.
  - In many ways, like a function of type `'a -> ans`
  - But, here they do not have function types.
- The code `throw k v` does the following:
  - Replaces the continuation of the `throw`
    (i.e., whatever we were planning to do afterwards)
    with the continuation `k`.
  - Starts this continuation off with the value `v`.

Continuations in SML/NJ

- The code `callcc f` behaves just like `f(K)` where `K` is the continuation of the `callcc`.
- In other words, the code `callcc f`
  - Grabs the continuation of the `callcc`
    (i.e., whatever we are going to do with the result of the `callcc`)
  - Calls `f` with this continuation as the argument
  - Return the function's value (if any)

Examples

```
val example1 : int =
      3 + (callcc (fn k => 2 + throw k 1))

val example2 : int =
      3 + (callcc (fn k => 2 + 1))

val example3 : int =
      3 + (callcc (fn k => throw k 4))

val example4 : int =
      3 + (callcc (fn k => raise (throw k 3)))
```

Another List-Multiplying Function

```
fun mult lst =
    callcc (fn (k_return : int cont) =>
               let
                  fun mult' [] = 1
                    | mult' (0::_) = throw k_return 0
                    | mult' (n::ns) = n * (mult9' ns)
               in
                  mult' lst
               end)
```

Tricky Problem

- Define the function

  `compose : ('a cont) -> ('b->'a) -> ('b cont)`

  such that

  `throw (compose k f) v`

  behaves the same as

  `throw k (f v)`

Comments on `callcc`

- First-class continuations have been called the "functional `goto`"
  - In many ways, even worse than `goto` since the place where you're jumping to is a run-time value!
- Next class: implementing cooperative multitasking in ML using `callcc`
  - a.k.a. coroutines
  - a.k.a. non-preemptive user-level threads

setjmp/longjmp

- Closest C equivalents to `callcc/throw`

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf k;
int f (int i) {
   if (i==0) longjmp(k, 42);
   else return i*f(i-1);
}
void main() {
   int result = setjmp(k);
   switch(result) {
     case 0:  f(5); break; // setjmp returns 0 first time
     default:  printf("f(5) = %d\n", result); break;
   }
}
```

stores run-time position into the buffer k

setjmp/longjmp

- Warning: setjmp does not actually capture the entire continuation!
  - Not the stack, but just the stack pointer.
  - Hence, the following code usually segfaults:

```
#include <setjmp.h>
jmp_buf k;
int f (int i) {
   if (i==0) return (setjmp(k));
   else return i*f(i-1);
}
void main() {
   int result = f(5);
   longjmp(k,42);
}
```

Coroutines

- Co-operative multitasking
  - Multiple "threads of control"
  - Each thread runs until done or deciding to temporarily yield
    - No pre-emption
  - *User-level* threading, lightweight but invisible to the kernel.
- Interface:

```
spawn : (unit -> unit) -> unit
exit  : unit -> 'a
yield : unit -> unit
```

Setup: Queues

- We assume we have an implementation of *imperative* queues

```
type 'a queue
val mkQueue : unit -> 'a queue
val enqueue : 'a queue * 'a -> unit
val dequeue : 'a queue -> 'a option
```

Ready Queue

- We maintain a queue of all the threads that are waiting to run as soon as they get a turn
  - We represent each such thread as a value of type `unit cont`
  - Starts out empty

```
val readyQ : (unit cont) queue = mkQueue ()
```

Terminating a coroutine

- The function `exit` discards the current coroutine and starts executing the next thread in the ready queue.

```
exception OutOfThreads

fun exit () =
    (case (dequeue readyQ) of
       NONE => raise OutOfThreads
     | SOME t => throw t ())
```

Yield

- Grab the state of the current thread and put it on the ready queue, then start the next thread.

```
fun yield() =
    callcc(fn parent =>
             (enqueue (readyQ, parent);
              exit())
```

Spawn

- The function `spawn` takes a function `f` and creates a new thread whose only job is to execute `f()` (and then `exit()` if this function ever completes).

- Complication:
  - Code is simpler if we create a new thread that returns from the spawn and continues on, while the current thread starts running the function f.

Spawn

```
fun spawn f =
    callcc(fn parent =>
             (enqueue (readyQ, parent);
              f();
              exit())
```

Spawn Revisited

- If we really want to create a new thread that runs the child…

```
fun spawn' f =
  let val child_continuation =
        callcc (fn k =>
                   (callcc(fn child =>
                             throw k child);
                    f ();
                    exit()))
in
  enqueue (readyQ, child_continuation)
end
```

Simple Producer/Consumer

```
local
   val buf : int ref = ref ~1
in
   fun producer n  = (buf := n;
                      yield ();
                      producer (n+1))
   fun consumer () = (print (Int.toString (!buf));
                      print "\n";
                      yield ();
                      consumer ())
   fun run () : unit =
        (spawn' consumer; producer 0)
end
```

Improved Example

```
local
   val buf : (int option) ref = ref NONE
in
   fun prod2 n =
      (case !buffer of
         NONE   => (buf := SOME n; yield(); prod2 (n+1))
       | SOME _ => (yield (); prod2 n))
   fun cons2 () =
      (case !buffer of
         NONE   => (yield(); cons2 ())
       | SOME n => (print (Int.toString n);
                    buf := NONE; yield(); cons2()))
   fun run2 () : unit =
      (spawn cons2; prod2 0)
end
```