Callcc and Coroutines

February 28, 2001
CS 131: Programming Languages

Review

• Continuation
  - "What should happen next"
  - "What to do with the result of the current computation"
• Can be represented explicitly as a function

```
fun mult5 lst =
  let
    fun mult5'([], k) = k 1
    | mult5'(n::ns, k) = mult5'(ns, k o (fn a => n * a))
  in
    mult5' (lst, fn a => a)
  end
```

Abstraction

• We can talk more generally about the continuation of some piece of code
  - What is going to happen to its result?
  - Abstraction of the processor's state

```
(2 * (3+4)) - 1
```

```
print (Int.toString (5*3))
```

Continuations in SML/NJ

• We can grab the continuation of an arbitrary piece of code, and manipulate it like a value

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```
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.
• The code \texttt{throw k v} does the following
  – Discards the continuation of the \texttt{throw} (i.e., whatever we were going to do with the result of the \texttt{throw})
  – Makes \texttt{k} the current continuation.
  – Start this continuation off with the value \texttt{v}.

Examples

\begin{verbatim}
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)))
\end{verbatim}

Continuations in SML/NJ

• The code \texttt{callcc f} does the following
  – Grabs the continuation of the \texttt{callcc} (i.e., whatever we are going to do with the result of the \texttt{callcc})
  – Applies the function \texttt{f} to this continuation.
  – Return the function’s value (if any) and proceed on with the continuation of the \texttt{callcc}.

Another List-Multiplying Function

\begin{verbatim}
fun mult9 lst =
  callcc (fn (k_return : int cont) =>
    let
      fun mult9' [] = 1
      | mult9' (0::ns) = throw k_return 0
      | mult9' (n::ns) = n * (mult9' ns)
    in
      mult9' lst
    end)
\end{verbatim}
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)

Solution

Coroutines

• Co-operative multitasking
  - Multiple “threads of control”
  - Each thread runs until it finishes or it decides to temporarily yield
  - No pre-emption
• 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

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

Exit

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

```haskell
exception OutOfThreads

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

Spawn

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

- 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`.

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

• Grab the state of the current thread and put it on the ready queue, then start the next thread.
  - Like spawn, except we don’t execute a child

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

Spawn Revisited

• If we really want to create a new thread that runs the child...

```ml
fun fork' 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

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

Busy-Waiting Example

```ml
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 =
    (fork cons2; prod2 0)
end
```