Continuations

Amer Diwan

What and why

• Continuations are a very powerful mechanism
  – A “non-local” jump, somewhat like exceptions but really more powerful
  – Can implement exceptions, co-routines, time-travel, ... in terms of continuations
  – ... but continuations themselves are hard to implement
• Supported in modern functional languages
  – SML
  – At least some variants of LISP
Outline

• Illustrate continuations using a range of examples
• Explore some interesting uses of continuations, particularly co-routines
• Next lecture: implementation aspects of continuations

What is a continuation?

• ...it is the “rest” of the computation
• ...it’s the computation that happens with the result of an expression

\[ x + y \]

The continuation of “x” says

... take its value and add “y” to it

\[ \text{fn } v \Rightarrow v + y \]
A more elaborate example

• Without continuations:
  – let fun prod_aux nil = 1
    | prod_aux (0::_) = 0
    | prod_aux (h::t) = h * (prod_aux t)
  in
    fun product l = prod_aux l
  end
• What computation does it do when we call it on [1,2,3,4,0,5]?
  – 1 * (2 * (3 * (4 * (0))))

Would like to avoid the wasted work

• Would like to avoid computing 1 * 2 * 3 * 4 * 0
• Two alternatives:
  – When “0” is detected, raise an exception that avoids the wasted computation
  – Use a continuation
• This example can be done equally well with exceptions. Later examples will demonstrate more of the power of continuations
Continuations in example

- let fun prod_aux nil = 1
  | prod_aux (0::_) = 0
  | prod_aux (h::t) = h * (prod_aux t)

in
  fun product l = prod_aux l
end

- What is the continuation of the recursive call to prod_aux?
  - Multiply it with all previous elements in the list

A high-level view of using a continuation

- As we walk through the list, note down the multiplies we need to do
- If we reach the end of the list without encountering a 0, then do the multiplication
- Otherwise, throw away the “notes” and return a 0
- Continuations capture the concept of “notes”
Nitty-gritty details

- k is the continuation that represents the “rest” of computation

  let
  
  fun prod_aux nil k = k 1
  (if we reach end of the list, then evaluate the “notes”)
  | prod_aux (0::_) k = 0
  (if we see a zero, then throw away the “notes” and return 0)
  | prod_aux(h::t) k = prod_aux t (fn result => k (h * result))
  (otherwise, create a new function that multiplies the current head
  with the previously buffered computation)

  in
  
  fun product l = product l (fn x => x)
  end

A closer look at k for list [1,2,3]

- Let ‘l’ be the list [1,2,3]. Ignore the “0” case...
  
  let fun prod_aux nil k = k 1
   | prod_aux(h::t) k = prod_aux t (fn result => k (h * result)))
  in fun product l = product l (fn x => x) end

- Initially k₁ is fn x => x (i.e., no more computation)
- After 1st recursive call, k₂ is fn result => k₁ (1 * result)
- After 2nd recursive call, k₃ is fn result => k₂ (2 * result)
- After 3rd recursive call, k₄ is fn result => k₃ (3 * result)
- The nil case applies k₄ to l => k₃ (3 * 1)
  => k₂ (2 * (3*1))
  => k₁ (1 * (2 * (3*1)))
  => (1 * (2 * (3 * 1)))
What if it was [1,2,3,0]

- When we would see a ‘0’ we would return a ‘0’ and throw ignoring the continuation
- Is this really a “faster way” of doing list multiply?
  - Probably not: trades wasted multiplications for calls and extra parameter passing
  - But it is a powerful model and an elegant way to express other more interesting computation

Language support for capturing continuations

- Modern function languages allow programs to manipulate their continuations
- `val callcc : (‘a cont -> ‘a) -> ‘a`
capture the continuation of an expression, and pass it to the parameter function (‘a cont -> ‘a). Return a ‘a.
- `val throw : ‘a cont -> ‘a -> ‘b`
  Restore the continuation passed to throw and pass it an argument of type ‘a
An example

- if callcc(fn k => a orelse b) then foo() else goo()
- callcc captures the rest of the computation that will use the result of ‘a orelse b’
- if ‘a orelse b’ throws ‘k’ with argument ‘i’, then it will appear as if callcc returned with value ‘i’
- if ‘a orelse b’ does not throw ‘k’ then it will appear as if callcc returned with whatever ‘a orelse b’ computes

A more concrete example

- fun product l = callcc(fn exit =>
  let fun prod_aux nil = 1
  | prod_aux(0::t) = throw exit 0
  | prod_aux(h::t) = h * prod_aux t
  in prod_aux l
  end)

- The magic:
  – throw exit 0 has the effect of terminating the expression in callcc causing callcc to return 0
  – otherwise, computation happens as normal
Continuations as co-routines

• Assume that there are two arrays, A and B
  – each element has three parts: an integer value, a
    “put” semaphore, and a “get” semaphore
  – thread 1 reads from array A, doubles the value and
    writes it in array B
  – thread 2 reads from array B, halves the value and
    puts it in array A
  – both use the get and put semaphores for exclusive
    access

Thread 1

• fun thread1(i,j) =
  let val {value=Av, put=Ap, get=Ag} = A sub i
  val {value=Bv, put=Bp, get=Bg} = B sub i
  val x = (P Ag; !Av)
  in V Ap; P Bp; Bv = 2*x; V Bg;
    yield();
    thread1((i+1) mod Alen, (j+1) mod Blen)
  end
The magic is in the “yield”

• fun yield() =  
  if random()  
    then ()  
    else callcc(fn k => (enqueue k; dispatch()))
  
• fun dispatch() = let val head::rest = !queue  
    in queue := rest; throw head()
  end

Various points...

• A continuation may be stored in a variable or returned from a function  
  – It may escape!  
  – Cannot pop activation records on return: some continuation may need them  
  • Similar to closures, may have to allocate activation records on the heap
Discussion

• Have you seen call/cc like features in common languages?

• Other uses of continuations?

Next lecture: efficiently implementing continuations and closures

• Reading:
  – Representing control in the presence of first-class continuations; R. Hieb, R. Kent Dybvig, and Carl Bruggeman; PLDI 90