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
fn v => v + y
A more elaborate example

• Without continuations:
  – let fun prod_aux nil = 1
     | prod_aux (0::nil) = 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 = prod_aux 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 ignore 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
Another example: 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 j
  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
Another continuation example

- Scenario: Would like to search a game tree for a winning position
- Let’s ignore “cycles” for this example--but it is not hard to extend this for cycles
- Let’s say a position with a “0” value is a winning value

An example tree:

```
BNode(10, BNode(5, Leaf(7), Leaf(3)), BNode(6, Leaf(0), Leaf(1)))
```

A typical algorithm: 10, 5, 7, (backtrack), 5, 3, (backtrack), 5, (backtrack), 10, 6, 0, (success!)
A straightforward implementation

- Each call to traverse returns a success or failure
- If it returns success, then don’t search remaining branch--simply return success
- If it returns failure, then search remaining branches
- If all outgoing branches have been searched, return failure

Problems with straightforward implementation

- Annoying codes to check and return
- “Success” needs to be propagated through all the levels of recursion to the top level
- “Failure” requires backtracking some number of levels up
- Can be made to work with exceptions, but let’s use continuations instead
Try 1

• datatype 'a Tree =
  Leaf of 'a | BNode of 'a * 'a Tree * 'a Tree
• fun travl (Leaf(v)) sk = if v=0 then throw sk() else ()
  | travl (BNode (v, l, r)) sk =
    if v=0 then throw sk()
    else (travl l sk; travl r sk)

• How does this work?
• What happens if “success” is never found?
• How about implementing backtracking using continuations?

Try 2

• fun travl (Leaf(v)) bk sk =
  if v=0 then throw sk() else throw bk()
  | travl (BNode(v, l, r)) bk sk =
    if v=0 then throw sk()
    else (callcc(fn k => travl l k sk); travl r bk sk)

• Two continuations are passed now:
  – to indicate success
  – to mark a place to backtrack
Rest of the code

- fun useit atree =
  (callcc(fn k => trav atree k); print "found!\n")
- fun trav t succk =
  (callcc(fn k => travl t k succk); print "Not ")

What really does the backtrack-continuation represent?

```
print "Not"
```

```
      10
     / \
    5   7
   / \  /
  7   3 6
```
Discussion

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

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