Continuations

COMP 320

13 CPS

Continuations and Trees

Problem: Given a binary tree, find a value d satisfying some property d.

If such a value exists, return SOME (d). Otherwise return NONE

We will use the familiar definition of binary trees

```
datatype 'a tree =
     Empty
   | Node of 'a tree * 'a * 'a tree
```

A simple solution

```
(* Finding an element d in the tree T s.t. p(d) is true *)
(* find: ('a -> bool) -> 'a tree -> 'a option *)
fun find p T =
   case T of
      Empty => NONE
    | Node (L, d, R) =>
           if  (p d) then SOME(d)
           else
             (case (find p L) of
                   NONE => find(p, R)
                |  SOME(d') => SOME(d'))
```

What’s the problem?

This solution works but …

If we find the value, say d’, in the left subtree, we pass the information, SOME (d’) back and it propagates up the tree

Once we have found an element, we would like to return

Using continuations gives us control of the call stack

The continuation keeps track of what to do when we have not found a value which satisfies p

CPS version of find

```
(* findCPS : ('a->bool) -> 'a tree -> (unit-> ' a option) -> 'a option*)
 fun findCPS p Empty cont =
   case T of
      Empty => cont ()
     | Node(L, d, R) =>
         if (p d) then SOME(d)
         else findCPS p L (fn () => findCPS p R cont)

 fun find p t = findCPS p t (fun() -> NONE)
```

What did we accomplish?

The continuation keeps track of what we still have to do

In this case, it keeps track of the tree we still have to traverse

At a leaf, we call the continuation and continue with the problem that is on top of the stack of continuations

If we are at a node Node (L, d, R) and (p d) is true, we return Some (d) and discard the continuation

If d does not satisfy the predicate p, the remaining task, searching the right subtree, is in the continuation

The function starts with the initial continuation fn () => None

Example

```
val t = Node
          (Node (Empty,3,Empty),5,Node (Empty,7,Empty)),
           10,
           Node(Empty, 15, Empty))

val p = (fn x => x = 7)

findCPS p t (fn () => NONE)
```

Regular Expressions

The set of regular expressions is given by the grammar:

$$r \;::= 0|1|a|r_1 r_2|r_1 + r_2|r^*$$

Examples (from Prof. Pientka "Programming Language Paradigms"

```
a(p*)l(e+y) matches apple, apply
g(1+r)(e+a)y matches grey, gray, gay
g(1+o)*(gle) matches google, gogle, goooogle, gle
b(ob0+oba) matches boba but not bob
```

Regular Expression Definition

```
datatype regexp =
    Zero
  | One
  | Char of char
  | Times of regexp * regexp
  | Plus of regexp * regexp
  | Star of regexp
```

Matching Patterns

Problem: Implement a pattern matcher

Given a regular expression, r, and a character string

Return true if the string is in L(r) and false otherwise

For convenience we work with a list of characters (type char list) instead of a string

The function explode : string -> char list can convert a string to a list of characters

Language Definition

```
L(0)         = the empty set
L(1)         = {[]}
L(c)         = {[c]}
L(r1+r2)     = {cs | cs is in L(r1) or cs is in L(r2)}
L(r1 r2)     = {cs | for some p in L(r1) and s in L(r2), cs==p@s}
L(r*)        = the smallest set which contains “” and the list of
               strings cs where, for some p in L(r) and s in L(r*),
               cs==p@s
```

Getting Started

```
fun match (r : regexp) (cs : char list) : bool =
    case r of
        Zero => false
      | One => (case cs of [] => true | _ => false)
      | Char c => (case cs of
                       [c‘] :: chareq (c,c’) | _ => false)
      | Plus (r1,r2) => match r1 cs orelse match r2 cs
      | Times (r1,r2) => ???
```

Match with continuation

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
    case r of
        Zero => false
      | One => k cs
      | Char c => (case cs of
                       []  => false
                     | c' :: cs' => chareq (c,c') andalso k cs')
      | Plus (r1,r2) => match r1 cs k orelse match r2 cs k
      | Times (r1,r2) => match r1 cs (fn cs' => match r2 cs' k)
      | Star r =>
            let fun matchstar cs' = k cs' orelse match r cs' matchstar
            in
                matchstar cs
            end
```

Initializing the continuation

```
fun chareq (c,c') = case Char.compare (c,c') of EQUAL => true | _ => false


local
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
    (* as defined on previous slide *)
in
    fun accepts (r : regexp) (s : string) : bool =
        match r (String.explode s) (fn [] => true | _ => false)
end
```

Examples – note the bindings

```
val true = accepts (Times (Plus(Char #"a",Char #"a") , Char #"b")) "ab"
val true = accepts (Times (Plus(Char #"b",Char #"a") , Char #"b")) "ab"
val true = accepts (Times (Char #"a",Plus(Char #"a",Char #"b"))) "ab"
val true = accepts (Star (Times (Char #"a", Char #"b"))) "ababab"
val false = accepts (Times (Plus(Char #"a",Char #"a") , Char #"a")) "ab"
val false = accepts (Times (Plus(Char #"b",Char #"c") , Char #"a")) "ab"
val false = accepts (Times (Char #"a",Plus(Char #"d",Char #"c"))) "ab"
val false = accepts (Char #"a") "ab"
val false = accepts (Times (Char #"a" , Char #"b")) "a"
val true = accepts (Times (Times(Char #"a",Char #"b") , Char #"c")) "abc"
val true = accepts (Times (Char #"a", Times(Char #"b",Char #"c"))) "abc"
```

Staging

Consider the function

```
fun exp (e : int) : int -> int =
  fn b => case e of
                   0 => 1
                 | _ => b* (exp (e-1) b)
```

Because of currying, we can define

```
val square = exp 2
val cube = exp 3
```

Evaluation

The value of square is:

```
exp 2
|-> fn b => case 2 of (0 => 1 | _ => b*(exp(2-1) b)
```

Functions are values, so the evaluation stops there

However we can prove that `exp 2 == fn b => b*b`

Now exp 2 would take longer to execute because of the recursive calls but they are equivalent because they produce the same result for any argument

Staging

A multi-staged function does useful work when applied to only some of its arguments

Then applying to the arguments specializes the staged function generating code specific to those arguments

This can improve efficiency when the specialized function is used frequently

A staged exponentiation

```
fun staged_exp (e : int) : int -> int =
  case e of
     0 => (fn _ => 1)
   | _ => let val onestep = staged_exp (e-1)
          in
             fn b => b * onestep b
          end
```

This function delays asking for the base until the exponent has been processed

There is no recursion on 2 every time it is applied

Evaluating the function

```
staged_exp 2
|->* let val onestep = (staged_exp (2-1) in fn b => b*onestep b end
|->* let val onestep = (let val onestep =
        (staged_exp (1-1) in fn b => b*onestep b end) in fn b => b*onestep b end
|->* let val onestep = (let val onestep =
        (fn _ => 1) in fn b => b*onestep b end) in fn b => b*onestep b end
|-> let val onestep = (fn b => b* ((fn _ => 1) b)) in fn b => b*onestep b end
|-> fn b => b * ((fn b => b * ((fn _ => 1) b)) b)
```