CIS552: Advanced Programming

Handout 20

Tail Recursion

Recall...

Ordinary factorial function:

```
fact n = if n==0 then 1 else fact(n-1)
```

Recall...

Tail-recursive factorial:

```
fact' n a = if n==0 then a
            else fact' (n-1) (a*n)

fact'' n = fact' n 1
```

The second one will be compiled to much more efficient code, because the compiler can see that the recursive call to `fact'` is in tail position—i.e., its result is the result of the whole body of `fact'`.

This means that the stack frame for the current call to `fact'` is not needed any more, so the recursive call can just re-use the same stack frame. (The "call" instruction becomes a "jump.")

I.e., `fact'` will be compiled to a loop.

Tail Position

But what, exactly, is this notion of "tail position"?

Not "rightmost subexpression," because this also describes the (non-tail) recursive call in the original `fact`.

And conversely, tail calls may also occur in non-rightmost positions, textually:

```
fact4 n a = if n/=0 then fact (n-1) (a*n)
            else a
```

We can make the notion precise by introducing the idea of continuations.

Continuations

Continuations

At each point during a computation, we can think of (1) some subcomputation that is eventually going to yield a value and, (2) some context that is waiting for this value and is going to use it to finish computing the final value of the whole program.

(2) is called the continuation of (1).

E.g., the continuation of the subexpression `fact 3` in the computation of `fact 6` is.

```
6 * (5 * (4 * □)),
```

where □ indicates the place where the value of `fact 3` will be used.

Making Continuations Explicit

A continuation can be thought of as a function from the result of the subexpression to the final result of the whole computation. I.e., we can write the continuation from the previous slide as

```
\x -> 6 * (5 * (4 * x))
```

Making Continuations Explicit

We can use this observation to write another version of `fact` that makes these continuations explicit:

```
fact_cps n k =
  if n==0
    then k 1
    else fact_cps (n-1) (\x -> k (n*x))
```

Note that all calls in `fact_cps` are in tail position. I.e., every call can be compiled as a jump. (A continuation can be described as a "goto with arguments.")

Q: So when does the stack grow?

CPS

This programming style is called continuation-passing style: every function is explicitly passed its continuation—i.e., another function to which it should send its result.

Another Example

Here's a CPS variant of another familiar function:

```
length_cps [] k     = k 0
length_cps (_:xs) k = length_cps xs (\x -> k (x+1))
```

Q: What is the type of `length_cps`?

Continuations for control

Continuation-passing style makes the control structure of the program explicit.

This can be applied to achieve a wide variety of useful effects.

Continuations for control

```
prodlist [] = 1
prodlist (x:xs) = x * prodlist xs

main1 = print $ prodlist
                  [1,2,3,4,5,6,7,8,9,
                   0,
                   10,11,12,13,14,15,16,17,18,19,
                   20,21,22,23,24,25,26,27,28,29,
                   30,31,32,33,34,35,36,37,38,39
                  ]
```

How many multiplications?

Continuations for control

```
prodlistCAux []     k  =  k 1
prodlistCAux (0:_)  _  =  0
prodlistCAux (x:xs) k  =  prodlistCAux xs
                                 (\r -> k (x * r))

prodlistC xs = prodlistCAux xs (\x -> x)

main3 = print $ prodlistC
                  [1,2,3,4,5,6,7,8,9,
                   0,
                   10,11,12,13,14,15,16,17,18,19,
                   20,21,22,23,24,25,26,27,28,29,
                   30,31,32,33,34,35,36,37,38,39
                  ]
```

Now how many?

CPS Transform

It is possible to rewrite any program in continuation-passing style.

Indeed, there is a mechanical procedure (called a CPS transform) that will take an arbitrary program and produce an equivalent CPS program.

This transformation plays a critical role in some compilers for functional languages.

Continuations for Backtracking

It is sometimes useful to define functions taking multiple continuations.

For example, programs that perform some kind of search can often be expressed very naturally using continuations. A searching function is passed two continuations:

- a success continuation that tells it what to do if this subtask succeeds, and
- a failure continuation that tells it how to "unwind" to a previous choice point, if something fails.

Example: Searching in CPS

```
data Tree a b = Leaf a b | Node (Tree a b) (Tree a b)
myTree = Node (Leaf 5 3) (Leaf 2 4)


findk :: Eq a => a -> (Tree a b) -> (Maybe b -> r) -> r -> r
findk a t sk fk =
  case t of
    Leaf a' b | a==a' -> sk (Just b)
              | a/=a' -> fk
    Node t1 t2        -> findk a t1 sk (findk a t2 sk fk)

find a t = findk a t (\b -> b) Nothing

main4 =
  do print (find 1 myTree)
     print (find 2 myTree)
```

Example: Searching in CPS

To make the failure continuation more interesting, let's keep track of how many nodes had to be searched to find the given key (or run out of nodes to search).

```
findk :: Eq a => a -> (Tree a b) -> Int ->
                     (Maybe (b,Int) -> r) -> (Int->r) ->
                     r
findk a t count sk fk =
  case t of
    Leaf a' b | a==a' -> sk (Just (b,count))
              | a/=a' -> fk count
    Node t1 t2        -> findk a t1 (count+1)
                           sk (\c -> findk a t2 c sk fk)

find a t = findk a t 1 (\b -> b) (\_ -> Nothing)

main5 =
  do print (find 1 myTree)
     print (find 2 myTree)
```

Call/CC

Many functional languages (including Scheme, the SMLNJ implementation of Standard ML, and Haskell) provide an operator for gaining explicit access to the “current continuation” at any point in a program.

This operator is traditionally called `call/cc` (“call with current continuation”).

It can be used for many amazing and mind-bending programming tricks.

In Haskell, the call/cc operator is (of course) packaged in a monad — the continuation monad, `Cont`.