Continuation Passing Style

15-150 M21

Lecture 0628
28 June 2021

- ✓ Basics of Functional Computation
- ✓ Induction and Recursion
- ✓ Polymorphism & Higher-Order Functions
- Functional Control Flow
- The SML Modules System
- Applications & Connections

Attunement: Converting to Tail-Recursive Form

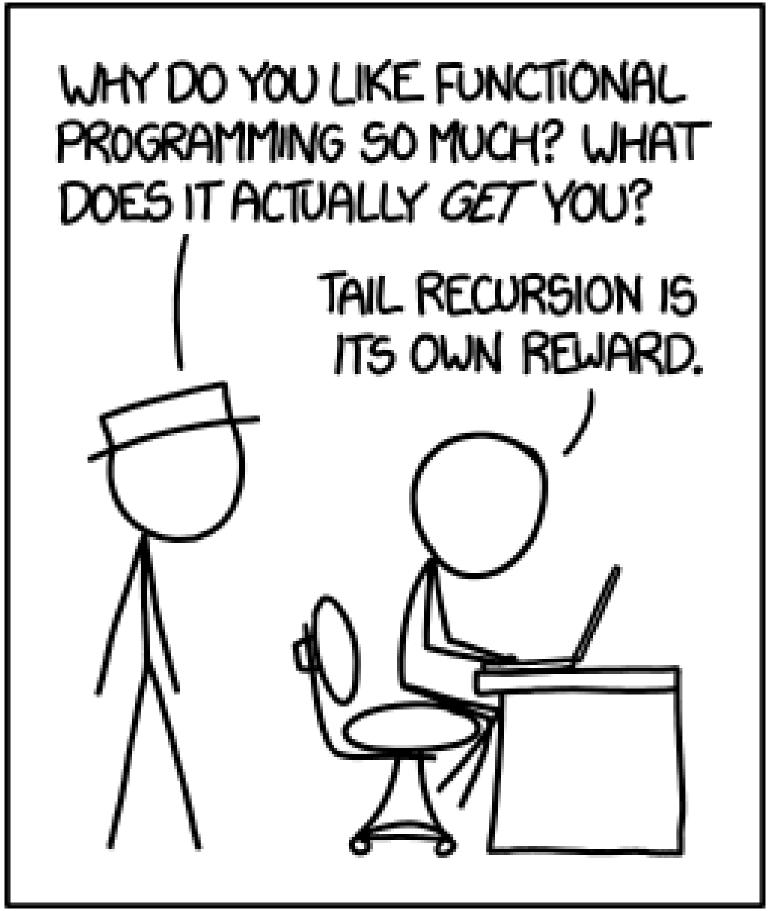
Recall: Tail Calls and Tail Recursion

Defn. A function application `f(x)` is in **tail position** in an expression `e` if, whenever `f(x)` is evaluated as part of evaluating `e`, the overall value of `e` is the value of `f(x)`.

- **Example:** `case L of [] => false | (x::xs) => f(x)`
- **Non-Example:** `if f(x) then 7 else 5`

A recursive function is said to be **tail recursive** if all of its recursive calls are in tail position.

```
fun foldl g z [] = z
  | foldl g z (x::xs) = foldl g (g(x,z)) xs
```



- Sometimes, it's asymptotically faster (`trev` vs. `rev`)
- Code can be optimized to make use of less stack space

```
foldl (op^) "!" ["H","E","L","L","O"]
 ⟹ foldl (op^) "H!" ["E","L","L","O"]
 ⟹ foldl (op^) "EH!" ["L","L","O"]
 ⟹ foldl (op^) "LEH!" ["L","O"]
 ⟹ foldl (op^) "LLEH!" ["O"]
 ⟹ foldl (op^) "OLLEH!" []
 ⟹ "OLLEH!"
```

Could make exp tail recursive

```
fun exp 0 = 1
  | exp n = 2 * exp(n-1)
```

```
fun texp (0,acc) = acc
  | texp (n,acc) = texp(n-1,2*acc)
```

```
fun square x = x * x
fun pow 0 = 1
  | pow n =
      case (n mod 2) of
        0 => square(pow(n div 2))
      | _ => 2*square(pow(n div 2))
```

Idea:

Use a more sophisticated accumulator which "remembers" to square (or square-and-double) the result at the end

0 Continuations

```
t1 ->(t2-> 'a) -> 'a
```

```
add : int -> int -> (int -> 'a) -> 'a
```
REQUIRES: true
ENSURES: `add m n k` $\cong$ `k(m+n)`

```
mul : int -> int -> (int -> 'a) -> 'a
```
REQUIRES: true
ENSURES: `mul m n k` $\cong$ `k(m*n)`

0628.0 (continuations.sml)

```
fun add m n k = k(m+n)
fun mul m n k = k(m*n)
```

0628.1 (continuations.sml)

```
fun foo u v w x y z =
        mul u     w      (fn res1 =>
        add v     res1   (fn res2 =>
        mul x     y      (fn res3 =>
        add res2  res3   (fn res4 =>
        mul res4  z      Fn.id ))))
```

0628.2 (continuations.sml)

```
fun foo' u v w x y z k =
      mul u    w    (fn res1 =>
      add v    res1 (fn res2 =>
      mul x    y    (fn res3 =>
      add res2 res3 (fn res4 =>
      mul res4 z    k))))
```

Today's Slogan:

Functions are accumulators

```
fun exp 0 = 1    |   exp n = 2 * exp(n-1)
```

```
expCPS : int -> (int -> 'a) -> 'a
```

REQUIRES: `n>=0`

ENSURES: `expCPS n k` $\cong$ `k(exp n)`

0628.3 (accum.sml)

```
fun expCPS 0 k = k 1
  | expCPS n k =
      expCPS (n-1) (fn res => k(2*res))
```

```
expCPS 3 Fn.id
⟹ expCPS 2 (fn exp2 => Fn.id(2*exp2))
⟹ expCPS 1
    (fn exp1 =>
     (fn exp2 => Fn.id(2*exp2))
     (2*exp1)
    )
⟹ expCPS 0
    (fn exp0 =>
     (fn exp1 =>
      (fn exp2 => Fn.id(2*exp2))
      (2*exp1)
     )
     (2*exp0)
```

```
⟹ (fn exp0 =>
    (fn exp1 =>
      (fn exp2 => Fn.id(2*exp2))
      (2*exp1)
    )
    (2*exp0))
  ) 1
⟹ (fn exp1 =>
    (fn exp2 => Fn.id(2*exp2))
    (2*exp1)
  ) (2*1)
⟹ (fn exp2 => Fn.id(2*exp2)) (2*2)
⟹ Fn.id(2*4)
⟹ 8
```

Thm. 1 For all types `t`, all values `k : int -> t`, and all values `n` with `n>=0`,

$$\texttt{expCPS n k} \cong \texttt{k(exp n)}$$

Proof. by simple induction on `n`.

BC `n=0`. Let `k` be arbitrary.

$$\texttt{expCPS 0 k} \cong \texttt{k 1} \cong \texttt{k(exp 0)}$$

by defn of `expCPS` and `exp`.

Proof.(continued)

IS `n=m+1` for some value `m:int` with `m>=0`.

IH **For all values** `g : int -> t`,

$$\texttt{expCPS m g} \cong \texttt{g(exp m)}$$

Let `k` be arbitrary.

```
expCPS (m+1) k
≅ expCPS m (fn res => k(2*res))          (defn expCPS)
≅ (fn res => k(2*res)) (exp m)           IH
≅ k(2 * exp m)                           (exp m valuable for m>=0)
≅ k(exp (m+1))                           (defn exp)
```

For each function `f : t1 -> t2`, we can define its "**CPS version**" which takes a continuation and performs the same task as `f`.

CPS (continuation passing style):

- CPS functions always take in continuation(s) as arguments
- Recursive CPS functions are always tail recursive
- CPS functions only ever call their continuations in tail position

Advantages to this approach

- Tail recursion: this is a technique to make any function tail recursive
- Explicitly name the result of recursive call
- Make the control flow explicit (and therefore manipulable)

0628.4 (accum.sml)

```
fun powCPS 0 k = k 1
  | powCPS n k =
    (case (n mod 2) of
      0 =>
        powCPS (n div 2) (fn res=>k(res*res))
     | _ =>
        powCPS (n-1) (fn res => k(2*res)))
```

Key Skill: CPS Conversion

Things you need:

“Direct-style” implementation

&

CPS spec

```
fun map f [] = []
  | map f (x::xs) = f(x) :: map f xs
```

```
mapCPS : ('a -> 'b) -> 'a list -> ('b list -> 'c)
-> 'c
```

REQUIRES: `f` is total

ENSURES: `mapCPS f L k` $\cong$ `k(map f L)`

0628.7 (accum.sml)

```
fun mapCPS f [] k = k []
  | mapCPS f (x::xs) k =
        mapCPS f xs (fn res => k((f x)::res))
```

```
filterCPS : ('a -> bool) -> 'a list ->
  ('a list -> 'b) -> 'b
```

REQUIRES: `p` is total

ENSURES: `filterCPS p L k` $\cong$ `k(filter p L)`

0628.5 (accum.sml)

```
fun filterCPS p [] k = k []
  | filterCPS p (x::xs) k =
      case (p x) of
         true  => filterCPS p xs
                     (fn res => k(x::res))
       | false => filterCPS p xs k
```

0628.6 (accum.sml)

```
fun filterCPS' p [] k = k []
  | filterCPS' p (x::xs) k =
      let
         fun k' res = if p x
                      then k(x::res)
                      else k(res)
      in
        filterCPS' p xs k'
      end
```

5-minute break

1 Continuation Control Flow

Summary so far

Given `f : t1 -> t2`, we can define its **CPS version**,

```
fCPS   :   t1 -> (t2 -> 'a) -> 'a
```

defined by the equivalence

$$\texttt{fCPS X k} \cong \texttt{k(f(X))}$$

Consider:

If we have a direct-style function

```
foo : t1 -> t2 option
```

then what does its CPS version

```
fooCPS : t1 -> (t2 option -> 'a) -> 'a
```

do?

```
t1 ->
(t2 option -> 'a)(t2 -> 'a) -> (unit -> 'a)
-> 'a
```

Backtracking with success and failure continuations

We'll now be supplying *two* continuations. If `t2` is the "result" type of the function (i.e. the type of data we want to pass into the continuation) and `t3` some other type, we'll supply:

```
sc : t2 -> t3     (* "success continuation" *)
fc : unit -> t3 (* "failure continuation" *)
```

So we can structure our code like this:

```
fun foo x sc fc =
   tryFirstThing x sc (fn () =>
   trySecondThing x sc (fn () =>
   ...
   tryNthThing x sc fc)...))
```

```
search : ('a -> bool) -> 'a tree -> ('a -> 'b) ->
(unit -> 'b) -> 'b
```

REQUIRES: `p` is total

ENSURES: `search p T sc fc` $\cong$ `sc x` where `x` is the first element of `T` (the first in a preorder traversal of `T`) such that `p x` $\cong$ `true`. If there is no such `x`, then `search p T sc fc` $\cong$ `fc()`

0628.8 (search.sml)

```
fun search p Empty sc fc = fc ()
  | search p (Node(L,x,R)) sc fc =
      if p x then sc x else
        search p L sc (fn () =>
        search p R sc fc)
```

0628.9 (search.sml)

```
datatype direction = LEFT | RIGHT

fun search' p Empty sc fc = fc ()
  | search' p (Node(L,x,R)) sc fc =
      if p x then sc [] else
        search' p L
          (fn res => sc(LEFT::res))
          (fn () =>
        search' p R
          (fn res => sc(RIGHT::res))
          fc)
```

- Can give functions continuations to specify what to do with their result
- Can integrate continuation into the recursion of the function, obtaining the "CPS version" of the function
- Recursive CPS functions are always tail recursive
- For searching functions that would normally return an **option**, we use a "success" and "failure" continuation in writing the CPS version

Next Time

- “Super CPS”
- CPS iteration

Thank you!