Lecture 12 Continuation Passing Style (CPS)

Definition of Continuation

A **continuation** is a function argument that encapsulates "what to do with the result".

Key Point: Rather than return the result directly, `f` hands the result off to `k`, as follow:

```
fun f {a bunch of args} k =
  let
    {a bunch of computations}
  in
    k {result of computations}
  end
```

Expressive Power: Because `k` appears as an argument the function `f` can manipulate it, e.g. treat it like a functional accumulator argument, frequently producing tail-recursive code.

Caution: If `f` is recursive, it cannot wait for return values, but must build new continuations that expect such return values.

Simple Examples

```
fun add (x, y, k) = k (x + y)

add (3, 4, fn r => r) => 7
add (3, 4, fn r => Int.toString r) => "7"
```

Some More Examples

A normal implementation of a recursive `sum` over a list.

```
fun sum [] = 0
  | sum (x::xs) = x + (sum xs)
```

Implementation using **continuation**.

```
fun csum [] k = k 0
  | csum (x::xs) k = csum xs (fn s => k (x + s))
```

Evaluation Trace of `csum`

```
    csum [2, 3] (fn s => s)
  = csum [3] (fn s' => (fn s => s) (2 + s'))
  = csum nil (fn s'' => (fn s' => (fn s => s) (2 + s')) (3 + s''))
  = (fn s'' => (fn s' => (fn s => s) (2 + s')) (3 + s'')) 0
  = (fn s' => (fn s => s) (2 + s')) (3 + 0)
  = (fn s' => (fn s => s) (2 + s')) 3
  = (fn s' => (fn s => s)) (2 + 3)
  = (fn s => s) 5
  = 5
```

Question. What is the connection between `sum` and `csum`?

Consider the `spec` of `csum`:

```
(* csum : int list -> (int -> 'a) -> 'a
 * req : true
 * ens : csum L k === k (sum L)
 *)
```

> Side Note. `sum` could be written as high-order: `val sum = foldl (op +) 0`.

Some Even More Examples

```
datatype tree = Empty | Node of tree * int * tree
fun inorder (Empty, acc) = acc
  | inorder (Node(L, x, R), acc) = inorder (L, x::(inorder (R, acc)))

(* treematch : tree -> int list -> bool
 * ens : treematch T L = true if inorder (T, nil) = L | false otherwise *)
fun treematch T L = (inorder (T, nil) = L)
```

Problem. This implementation results in the fact that we have to traverse the entire tree no matter what.

Re-Implement using Continuation

```
(* [helper] prefix : tree -> int list -> (int list -> bool) -> bool
 * req : k is total
 * ens : prefix T L k = true if L = L1 @ L2 such that inorder (T, nil) = L1
 *                                                     and k L2 = true
 *                      and false otherwise *)
fun prefix Empty L k = k L
  | prefix Node(l,x,r) L k = prefix l L (fn L2 => case L2 of
                                                    [] => false
                                                  | (y::ys) => (x = y) andalso (prefix r ys k)
```

Success & Failure Continuations

```
(* search : ('a -> bool) -> 'a tree -> ('a -> 'b)    success cont.
 *                                   -> (unit -> 'b)  failure cont.
 *                                   -> 'b
 * req : p, sc, f, are total
 * ens : search p T sc f = sc x if p x holds for some x in T | f () otherwise *)
fun search _ Empty _ f = f ()
  | search p (Node(l,x,r)) sc f =
    if p x then sc x
           else search p l sc (fn () => search p r sc f)
```