15-150
Spring 2018
Dilsun Kaynar
LECTURE 12
Continuations
Intuition

- A continuation is a functional argument that controls flow of expression evaluation. It is often used to abstract away the “rest of the computation”.
- Useful in backtracking search
- They act like “functional accumulators” in tail-recursive functions.
Simple cps functions

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

fun mult (x, y, k) = k (x * y)
Type of \texttt{add}

\begin{verbatim}
fun add (x,y,k) = k (x + y)
\end{verbatim}

\begin{itemize}
\item \texttt{add : int * int * (int -> 'a) -> 'a}
\end{itemize}

the overall return type is the return type of the continuation
Using a cps function

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

continuation

```haskell
add (3, 4, fn r => r)
```

What are the evaluation steps?
add (3, 4, \texttt{fn} r1 =>
    add (5, 6, \texttt{fn} r2 =>
        mult (r1, r2, \texttt{fn} r3 => r3)))

What if we wanted to return the result as a string?
add (3, 4, fn r1 =>
  add (5, 6, fn r2 =>
    mult (r1, r2, fn r3 => Int.toString r3)))
Alternatively

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

```haskell
fun mult (x, y) k = k (x * y)
```
(* sum : int list -> int
  REQUIRES: true
  ENSURES:  sum L returns the sum of all the elements in L *)

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

(* Using tail-recursion: *)
(* tsum : int list * int -> int
  REQUIRES: true
  ENSURES:  tsum (L, acc) == (sum L) + acc *)

fun tsum ([] : int list, acc : int) : int = acc
  | tsum (x::xs, acc) = tsum(xs, x + acc)

(* Using continuation-based passing style: *)
(* csum : int list -> (int -> 'a) -> 'a
  REQUIRES: k is total
  ENSURES:  csum L k == k (sum L) *)

fun csum ([] : int list) (k: int -> 'a) : 'a = k(0)
  | csum (x::xs) k = csum xs (fn s => k(x + s))
fun csum ([] : int list) (k: int -> 'a) : 'a = k(0) 
| csum (x::xs) k = csum xs (fn s => k(x + s))

csum [2,3] (fn s => s)

= csum [3] (fn s' => (fn s => s)(2 + s'))

= csum [] (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 => s) (2 + 3)

= 5
Comparing tree contents to list contents

```ml
datatype tree = Empty | Node of tree * int * tree

(* val inorder :  tree * int list --> int list 
   REQUIRES: true
   ENSURES:  inorder(T, acc) == L @ acc, where L consists of the 
            elements of T as encountered in an in-order traversal of T. *
*)
fun inorder(Empty, acc) = acc |
   inorder(Node(left, n, right), acc) = 
       inorder(left, n::inorder(right, acc))

(* treematch : tree -> int list -> bool 
   REQUIRES: k is total 
   ENSURES: treematch T L returns true if L consists of the 
            elements of T as encountered in an in-order traversal of T, 
            and returns false otherwise.
 *)

fun treematch T L = (inorder(T, nil) = L)
```
Matching tree contents to prefix of a list

(* prefix : tree -> int list -> (int list -> bool) -> bool

REQUIRES: k is total
ENSURES: prefix T L k ==>
{ true, if L == L1 @ L2, such that
{ the inorder traversal of T is equal to L1,
{ and k(L2) == true.
{ false, otherwise.
*)

fun prefix (Empty) L k = k(L)
| prefix (Node(left, n, right)) L k =
  prefix left L (fn L2 => case L2 of [] => false
  | y::ys =>
    (n=y) andalso (prefix right ys k))

(* treematch' : tree -> int list -> bool
REQUIRES: true
ENSURES: treematch' T L returns true if L consists of the
elements of T as encountered in an in-order traversal of T,
and returns false otherwise.
*)

fun treematch' T L = prefix T L List.null
Search problems

- Implemented using 2 continuations:
  - success (what to do if search is successful)
  - failure (takes unit as argument, implements backtracking)
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* search : ('a -> bool) -> 'a tree -> ('a -> 'b) -> (unit -> 'b) -> 'b
  REQUIRES: p, sc, and f are total.
  ENSURES: search p T sc f == {sc(x) if p(x)==true for some x in T
   {f()   otherwise
    (if more than one x satisfies p(x)==true, then use
     the first encountered in a pre-order traversal of T).
  *)

fun search _ Empty _ f = f()
| search p (Node(left, x, right)) sc f =
   if p(x) then sc(x)
   else search p left sc (fn () => search p right sc f)

(* findeven : int tree -> string
  REQUIRES: true
  ENSURES: findeven(T) returns the string representation of the
           first even integer found in a pre-order traversal of T,
           if there is such an integer. Otherwise, findeven(T)
           returns "none found".
  *)

fun findeven T =
   search (fn n => n mod 2 = 0) T Int.toString (fn () => "none found")