Higher-Order Functions

CS 131: Programming Languages

September 11, 2001

Cons vs. Append

- There are two binary operators used with lists
  - The :: operator, called cons
  - The @ operator, called append
- Note:

```
[1,2,3] :: [] = [[1,2,3]]
[1,2,3] @ [] = [1,2,3]
```

- Also, only `::` can appear in patterns.

Cons vs. Append

- The type of @ is:
- The type of `::` is:

Applying a Function to a List

- Problem: Apply some function `f` to every element of a list, return the list of results
  - That is, given the input

    $[x_1, \ldots, x_n]$

    return

    $[f(x_1), \ldots, f(x_n)]$.

```
fun loop []      =
  | loop (x::xs) =
```

Applying a Function to a List

- New Problem: Write a function map that *given* f, returns a function that applies f to every element of a list.

```
fun map f =
  let
     fun loop []      = []
       | loop (x::xs) = (f x)::(loop xs)
  in
     loop
  end
```

What is the type of map ?

- The argument can be any function.
- If we assume that `f : 'a -> 'b`, what is the type of the locally defined function `loop`?
- Then what is the type of `map`?

Doubling Lists

```
val doubler : int list -> int list
            = map (fn x => x*2)
val l       = doubler [1,2,3]
```

```
val l = map (fn x => x*2) [1,2,3]
```

```
fun double x = x*2
val l = map double [1,2,3]
```

Higher-Order Functions

- A function that takes a function as its argument or returns a function as its result is said to be a *higher-order function*.
  - e.g., map is higher-order
- Let's look at some more examples

Building Functions that Add

- Consider the following functions:

```
fun addone(x) = x + 1
fun addtwo(x) = x + 2
fun addsix(x) = x + 6
```

- Can we generalize this construction?
- Goal: a function that, given n, returns the function which adds n to its argument

Building Functions that Add

- It may help to consider the fully-expanded code for the functions on the previous slide:

```
val addone = (fn x => x+1)
val addtwo = (fn x => x+2)
val addsix = (fn x => x+6)
```

- Exercise: Define
  `add : int -> (int->int)`

Using add

```
val addone = add 1
val addtwo = add 2
val addsix = add 6

fun increment_list lst =
    map (add 1) lst

val increment_list' =
    map (add 1)
```

Types for Curried Functions

- The type of add is
  `int -> (int -> int)`
- Function types are right associative, so can write
  `int -> int -> int`
- There are two ways to think about this type.
  - The function add takes an int and returns a function on ints
    `add 4 : int->int`
  - The function add takes two integer arguments *in succession*
    `add 4 7 : int`

Syntax For Curried Functions

- Functions like add that do nothing but return another function are said to be *curried*.
- SML has special syntax for defining curried functions
  - Function argument patterns are separated by spaces

```
fun add n m = n+m

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

Staged Computation

- Consider the following variation of `add`, which adds the square of its argument:

```
fun addsquare 0 m = m
  | addsquare n m = (n*n) + m
```

Equivalently:

```
fun addsquare n m =
  (case n of
     0 => m
   | _ => (n*n)+m)
```

Staged Computation

- If we now execute the code

```
val f : int->int = addsquare 6
val l = map f [1,2,3,4,5]
```

then how many times do we compare 6 against 0?

- How many times do we compute 6*6?

Staged Computation

- More efficient version:

```
fun addsquare n =
  (case n of
     0 => (fn m => m)
   | _ => (fn m => (n*n)+m))
```

Staged Computation

- Even more efficient:

```
fun addsquare n =
  (case n of
     0 => (fn m => m)
   | _ => let
              val n2 = n*n
            in
              fn m => n2+m
            end)
```

Function Composition

- Goal: a function `compose` that, given functions `f` and `g`, returns their composite.
  - Recall: composite of `f` and `g` is the function which given `x`, returns `f(g(x))`.
  - Also, what is the type of this function?

Functions and Re-binding

- Consider the following definitions:

```
val x = 3
fun addx (y:int) = y+x
```

- Now, what is the value of `addx 2` ?

Functions and Re-binding

- Consider the following definitions:

```
val x = 3
fun addx (y:int) = y+x
val x = 5
```

- Now, what is the value of `addx 2` ?

Functions and Re-binding

- Consider the following input:

```
fun add1 x = x+1
fun add2 x = add1(add1(x))
val x = add2 4

fun add1 x = x+3
val y = add2 4
```

- Now, what are the values of x and y ?

The exists Function

```
(* exists : ('a -> bool) -> 'a list -> bool *)

fun exists p [] = false
  | exists p (x::xs) = (p x) orelse (exists p xs)

fun exists p =
  let fun loop [] = false
        | loop (x::xs) = (p x) orelse (loop xs)
  in
      loop
  end
```

The all Function

```
(* all : ('a -> bool) -> 'a list -> bool *)

fun all p [] = true
  | all p (x::xs) = (p x) andalso (all p xs)

fun all p =
  let fun loop [] = true
        | loop (x::xs) = (p x) andalso (loop xs)
  in
      loop
  end
```

The find Function

```
(* find : ('a -> bool) -> 'a list -> 'a option *)

fun find p [] = NONE
  | find p (x::xs) =
      if (p x) then (SOME x) else (find p xs)

fun find p =
  let fun loop [] = NONE
        | loop (x::xs) =
            if (p x) then (SOME x) else (loop xs)
  in
      loop
  end
```

The `partition` Function

```
(* partition : ('a -> bool) ->
                       'a list -> 'a list * 'a list  *)

fun partition p []      = ([], [])
  | partition p (x::xs) =
      let
         val (yes,no) = partition p xs
      in
         if p x then
            (x::yes, no)
         else
            (yes, x::no)
      end
```

A Question of Style

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

fun map2 (f,[])     = []
  | map2 (f,x::xs) = (f x) :: (map2 (f,xs))

fun map3 []      f = []
  | map3 (x::xs) f = (f x) :: (map3 xs f)

fun map4 ([],f)     = []
  | map4 (x::xs,f) = (f x) :: (map4 (f,xs))
```