Higher-Order Functions

CS 131: Programming Languages
January 24, 2001

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 = 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

```
make_adder : int -> (int->int)
```

Using make_adder

```
val addone = make_adder 1
val addtwo = make_adder 2
val addsix = make_adder 7

fun increment_list lst =
    map (make_adder 1) lst

val increment_list' lst =
    map (make_adder 1)
```

Syntax For Curried Functions

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

```
fun make_adder n m => n+m

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

Types for Curried Functions

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

Curried and Uncurried Functions

- Compare the types of these definitions

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

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

Function Composition

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

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 definitions:

```
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 (pyes,pno) = partition p xs
      in
         if p x then
            (x::pyes, pno)
         else
            (pyes, x::pno)
      end
```