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 \texttt{map} that given \( f \), returns a function that applies \( f \) to every element of a list.

\begin{verbatim}
fun map f =
  let
    fun loop []   = []
    | loop (x::xs) = (f x)::(loop xs)
  in
    loop
  end
\end{verbatim}

What is the type of \texttt{map}?

• The argument can be any function.
• If we assume that \( f : 'a \rightarrow 'b \), what is the type of the locally defined function \texttt{loop}?
• Then what is the type of \texttt{map}?
Doubling Lists

\[
\text{val doubler = map (fn x => x*2)} \\
\text{val l = doubler [1,2,3]}
\]

\[
\text{val l = map (fn x => x*2) [1,2,3]}
\]

\[
\text{fun double x = x*2} \\
\text{val l = map double [1,2,3]}
\]

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., \text{map} is higher-order

- Let's look at some more examples
Building Functions that Add

• Consider the following functions:

\[
\begin{align*}
\text{fun } \text{addone}(x) &= x + 1 \\
\text{fun } \text{addtwo}(x) &= x + 2 \\
\text{fun } \text{addsix}(x) &= x + 6
\end{align*}
\]

• 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:

\[
\begin{align*}
\text{val } \text{addone} &= (\text{fn } x \Rightarrow x+1) \\
\text{val } \text{addtwo} &= (\text{fn } x \Rightarrow x+2) \\
\text{val } \text{addsix} &= (\text{fn } x \Rightarrow x+6)
\end{align*}
\]

• Exercise: Define

\[
\text{make_adder : int } \rightarrow (\text{int} \rightarrow \text{int})
\]
Using \texttt{make\_adder}

\begin{verbatim}
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)
\end{verbatim}

\section*{Syntax For Curried Functions}

\begin{itemize}
  \item Functions like \texttt{make\_adder} that do nothing but return functions are said to be curried.
  
  \item SML has special syntax for defining curried functions
  \begin{itemize}
    \item Function argument patterns are separated by spaces
  \end{itemize}
\end{itemize}

\begin{verbatim}
fun make\_adder n m => n+m

fun map f [] = []  
| map f (x::xs) = (f x)::(map f xs)
\end{verbatim}
Types for Curried Functions

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

Curried and Uncurried Functions

- Compare the types of these definitions

```plaintext
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:

```plaintext
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
  ```

• Now, what are the values of `x` and `y`?
The \texttt{exists} Function

\begin{verbatim}
(* \texttt{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
\end{verbatim}

The \texttt{all} Function

\begin{verbatim}
(* \texttt{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
\end{verbatim}
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