Lecture 10

Higher-Order Functions (Part I)
Previously...

- Polymorphic types and instances
- Typing *rules* and *most general types*
- *The type guarantee*

If e has type t and e ==> v then v is a value of type t

An expression of type t can be used (safely!) at any instance of t
Curried functions

(* add : int * int -> int *)
fun add (x: int, y: int): int = x + y

(* plus : int -> int -> int *)
fun plus (x: int) : int -> int = fn (y:int) => x + y
Curried functions

(* add : int * int -> int *)
fun add (x: int,y: int):int = x + y

(* plus : int -> int -> int *)
fun plus (x: int) : int -> int = fn (y: int) => x + y

val incr3 : int->int =  plus 3

val 7 =  incr3 4

val 13 =  plus 7 6
Adding sugar to curry

(* add : int * int -> int *)
fun add (x:int,y:int):int = x + y

(* plus : int -> int -> int *)
fun plus (x:int) : int -> int = fn (y:int) => x + y

We could also write plus in the following more compact form:

fun plus (x:int) (y:int) : int = x + y
so far

If \( f : t_1 \times t_2 \rightarrow t_3 \) and \( f_c \) is its curried version then

\[
f_c : t_1 \rightarrow t_2 \rightarrow t_3
\]
Example: filter

(* filter : ('a -> bool) -> 'a list -> 'a list
  REQUIRES: p is total
  ENSURES: filter p L returns the elements of L that satisfy p
           (in same order as in L).*)

fun filter (p : 'a -> bool) ([ ] : 'a list) : 'a list = [ ]
  | filter p (x::xs) = if p(x) then x:(filter p xs)
    else filter p xs

This function is predefined in SML as List.filter
Example: filter

(* filter : ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L returns the elements of L that satisfy p
            (in same order as in L).
*)

fun filter (p : 'a -> bool) ([ ] : 'a list) : 'a list = [ ]
  | filter p (x::xs) = if p(x) then x::(filter p xs) else filter p xs

val [2,~6, 10] = filter (fn n => n mod 2 = 0) [1,2,~5,~6,11,10,13]
val keepevens = filter (fn n => n mod 2 = 0)
val [2,~6, 10] = keepevens [1,2,~5,~6,11,10,13]
map spec

(* map : ('a -> 'b) -> ('a list -> 'b list)
  REQUIRES  true
  ENSURES   For all n ≥ 0,
            map f [x₁, ..., xₙ] == [f x₁, ..., f xₙ]
*)

For all n ≥ 0, all types t₁ and t₂,
all values f : t₁ -> t₂, and all values x₁, ..., xₙ : t₁,
  map f [x₁, ..., xₙ] = [f x₁, ..., f xₙ].
map

(* map : ('a -> 'b) -> ('a list -> 'b list) *)

fun map f [ ] = [ ]
| map f (x::R) = (f x) :: (map f R)

map (incr3) : int list -> int list
map (incr3) [1, 2, 3] evaluates to [4, 5, 6]
combining data

• Given a *collection* of data, such as a list or tree
• We may want to *combine* the data, using a *binary operation* and a *base value*

We’ll talk about lists today… but there are similar ways to deal with trees, etc…
combining lists

• Suppose we have a function

\[
F : t_1 \times t_2 \rightarrow t_2
\]

and we want to combine the items of a list

\[
[x_1, \ldots, x_n] : t_1 \text{ list}
\]

with a value \( z : t_2 \)

to get the value of

\[
F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))
\]
a solution

- A **polymorphic** function

  \[
  \text{foldr} : (\text{\texttt{\char`\'a} \texttt{\char`\*} \texttt{\char`\textbackslash b} \rightarrow \texttt{\char`\textbackslash b}) \rightarrow \texttt{\char`\textbackslash b} \rightarrow \texttt{\texttt{\char`\'a} \texttt{list}} \rightarrow \texttt{\char`\textbackslash b}
  \]

  such that

  for all types \( t_1, t_2 \),

  all \( n \geq 0 \), and all values

  \[
  F : t_1 \times t_2 \rightarrow t_2, \ [x_1,\ldots,x_n] : t_1 \text{ list}, \ z : t_2, \\
  \text{foldr} \ F \ z \ [x_1,\ldots,x_n] = F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))
  \]

  (combines from *right to left*)
another solution

• A **polymorphic** function

\[
\text{foldl} : (\text{'a} \times \text{'b} \to \text{'b}) \to \text{'b} \to \text{'a list} \to \text{'b}
\]

such that

for all types \( t_1, t_2 \),
all \( n \geq 0 \), and all values

\[
F : t_1 \times t_2 \to t_2, \ [x_1, \ldots, x_n] : t_1 \ \text{list}, \ z : t_2, \]

\[
\text{foldl} \ F \ z \ [x_1, \ldots, x_n] = F(x_n, F(x_{n-1}, \ldots, F(x_1, z)\ldots))
\]

(combines from *left to right*)
foldr F z [x₁,...,xₙ] = F(x₁, F(x₂, ..., F(xₙ, z)...))

foldl F z [x₁,...,xₙ] = F(xₙ, F(xₙ₋₁, ..., F(x₁, z)...))
**foldr**

(* foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)

fun foldr F z [ ] = z
   | foldr F z (x::xs) = F(x, ______)

foldr F z [x₁,...,xₙ] = F(x₁, F(x₂, ..., F(xₙ, z)...))

(combines from right to left)
**foldr**

(* foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)

```plaintext
fun foldr F z [ ] = z
    | foldr F z (x::xs) = F(x, foldr F z xs)
```

foldr F z [x₁,...,xₙ] = F(x₁, F(x₂, ..., F(xₙ, z)...))

(combines from right to left)
foldl

(* foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)

fun foldl F z [ ] = z
| foldl F z (x::xs) = _______________

foldl F z [x₁,...,xₙ] = F(xₙ, F(xₙ₋₁,..., F(x₁,z)...))

(combines from left to right)
foldl

(* foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)

fun foldl F z [ ] = z
|   foldl F z (x::xs) = foldl F (F(x,z)) xs

foldl F z [x_1,...,x_n] = F(x_n, F(x_{n-1},..., F(x_1,z)....))

(combines from left to right)
Example: maxgain

fun pairup L = …

fun gain (buy, sell) = sell - buy

fun bestgain L = foldr Int.max neginf (map gain (pairup L))

val SOME(neginf) = Int.minInt
foldr \((\text{op } @)\) [ ] [[1,2], [ ], [3,4]] = [1,2,3,4]
foldl \((\text{op } @)\) [ ] [[1,2], [ ], [3,4]] = [3,4,1,2]

In general, when is
foldr \(f\) = foldl \(f\)?
wishful thinking

Problem: we need a function

\[
\text{silly : string list -> string}
\]

to transform \[
["the ","course ","is "]
\]

into \[
"the course is almost over"
\]

\[
\text{fun silly L = foldr (op ^) "almost over" L}
\]