Lecture 10 Curried Functions & Higher-Order Functions

Review of Function Definition, Evaluation, & Binding

Consider the definition of a function:

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

What happens upon this definition?

Namespace `add` is bind to a lambda expression **and** the prior environment.

Now consider another definition:

```
fun plus (x : int) : int -> int = fn (y : int) => x + y
(* The type of function plus:
 * plus : int -> int -> int
 * OR int -> (int -> int)
 * They are the same because functions are right-associative. *)
```

Namespace `plus` is bind to a lambda expression in which there is another lambda expression, and the prior environment (including the binding of `add`).

Now consider the binding `val incr 3 = plus 3`.

What is bind to `incr 3`?

Let's consider the evaluation trace:

```
    plus 3
==> (extend the env) [env when plus defined] [3/x] body of plus
==> [env...] [3/x] (fn y => x + y)
```

Therefore, `incr 3` is the value `fn y => x + y` and the `[env...]` **and** the binding `[3/x]` .

Now consider the call `incr 3 4` .

Evaluation trace:

```
    incr 3 4
==> [env when incr 3 defined] [4/y] body of incr 3
==> [env...] [3/x] [4/y] x + y
==> (find and substitute the bindings) 3 + 4
==> 7
```

What this means? Now we can call `plus 3 4` , but we can also call `plus 3` and later give it `4` .

Currying

Syntactic Sugar.

```
fun plus (x : int) (y : int) : int = x + y
(* This is the exact same definition as plus above. *)
```

Definition. `plus` is the **curried** form of `add` .

- `add` has type `int * int -> int`
- `plus` has type `int -> int -> int`

> Note: Currying allows us to hide things in the environment. The user cannot see the environment bindings, but they are affecting the function call.

Higher-Order Functions

Consider a function definition:

```
(* filter : ('a -> bool) -> 'a list -> 'a list
 * REQ: P is total
 * ENS: filter P L consists of all elements in L satisfying P, preserving order
 *)
fun filter (P : 'a -> bool) ([] : 'a list) : 'a list = []
  | filter P (x::xs) =
    (case P x of
      true => x :: (filter P xs)
    | false => filter P xs)
```

Now, consider how we can use this function.

```
    filter (fn n => n mod 2 = 0) [1, 4, 7, 8, ~2]
==> [4, 8, ~2]

val keep_evens = filter (fn n => n mod 2 = 0)
(* keep_evens : int list -> int list *)
```

This is because `filter : ('a -> bool) -> 'a list -> 'a list`. Here, `filter (...)` takes in `int -> bool`, so the left-out part is `'a list -> 'a list` and further `'a` is constrained to `int`.

Consider `filter (fn _ => true)`. This application has type `'a list -> 'a list`. It is extensionally equivalent to the identity function on lists.

The Map on List

Consider another function definition.

```
(* map : ('a -> 'b) -> 'a list -> 'b list
 * REQ: true (we may want f total)
 * ENS: map f [x1,...,xn] === [f(x1),...,f(xn)]
 *)
fun map (f : 'a -> 'b) ([] : 'a list) : 'b list = []
  | map f (x::xs) = (f x) :: (map f xs)
```

```
    map Int.toString [1, 2, 3]
==> ["1", "2", "3"]

val convert_to_string = map Int.toString
(* convert_to_string : int list -> string list *)
convert_to_string [2, 4, ~1] ==> ["2", "4", "~1"]
```

Another Important List Function

```
(* foldl / foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
 * foldl f z [x1,...xn] === f(xn,... f(x3, f(x2, f(x1, z))))
 * e.g. foldl (op +) 0 [1, 2, 3, 4] = 10 (foldr same here)
 * e.g. foldl (op -) 0 [1, 2, 3, 4] = 2
 *      foldr (op -) 0 [1, 2, 3, 4] = ~2
 *)
fun foldl (F : 'a * 'b -> 'b) (z : 'b) ([] : 'a list) : 'b = z
  | foldl F z (x::xs) = foldl F (F (x, z)) xs

fun foldr (F : 'a * 'b -> 'b) (z : 'b) ([] : 'a list) : 'b = z
  | foldr F z (x::xs) = F (x, (foldr F z xs))
```

Some Other Interesting Functions

```
List.exists : ('a -> bool) -> 'a list -> bool // returns false on empty list
List.forall : ('a -> bool) -> 'a list -> bool // returns true on empty list
```

I Don't Know What This is Doing

```
    stock prices [20, 25, 24, 30, 20]
==> [(20, 30), (25, 30), (24, 30), (30, 20)] by pair-up
==> [10, 5, 6, ~10] by sell - buy
==> 10 (max)
```

```
fun best_gain (L : int list) : int = foldr Int.max -Inf (map gain (pair_up L))
```

Side Note. `Int.minInt` in ML.