CS 421 Lecture 16

- Functional programming
- Higher-order functions

History of functional languages

- LISP, APL (1960)
- ML (1976) – Milner, "A theory of type polymorphism in programming"
- SASL (1976) – lazy evaluation
- SCHEME (1975) – Guy Steele – dialect of LISP with higher-order functions ("lexical scope")
- Standard ML, CAML (1980's)
- Erlang (1987) – Ericsson
- Haskell (1990) – lazy evaluation
- Python, ...

Functional languages

- Expressions (rather than statements)
  - Absence of side effects
  - “Large values”
- Dynamic memory allocation
- Recursion
- Static type checking with polymorphic types (ML, Haskell)
- **Higher-order functions**, aka “functions as values” (Scheme, ML, Haskell, Python, …)
- Lazy evaluation (Haskell)

Higher-order functions

- Functions are a type of value ("first-class functions")
  - Define anonymously
  - Pass as arguments
  - Bind to names
  - Assign to variables
  - Return from functions

Anonymous functions in Ocaml

- Notation: "fun x -> e" – Ocaml expression whose values is a function.
- "let f = fun x -> e" is equivalent to "let f x = e"

Passing functions as arguments

Higher-order functions in List module:

```
    map : (α->β)->α list -> β list
```

applies a function to each element of a list –

map f [x_1;x_2;...x_n] = [f x_1;f x_2;...f x_n]

E.g. `let lis = [1;2;3;4]`

```
    let incr x = x + 1
    map incr lis => [2;3;4;5]
```

or equivalently

```
    map (fun x -> x + 1) lis
```

Passing functions as arguments

$$\texttt{fold\_right f } [x_1; x_2; \ldots x_n] \texttt{ x}$$
$$= \texttt{f } x_1 \texttt{ (f } x_2 \texttt{ (...(f } x_n \texttt{ z)...))}$$

$$\texttt{fold\_right : } (\alpha\texttt{->}\beta\texttt{->}\beta)\texttt{->}(\alpha \texttt{ list})\texttt{->}\beta\texttt{->}\beta$$

`fold_right (fun x y -> x+y) lis 0` **=>** `10`

(Note: can use "(+)" for function argument.)

Passing functions as arguments

```
fold_right (fun x -> fun y -> x :: y) lis []
  => lis
fold_right (fun x -> fun y -> x :: y) lis lis
   => lis @ lis
fold_right
 (fun x -> fun y -> (x + (hd y))::y) lis [0]
  [1;2;3;4]  => [10;9;7;4;0]
fold_right
  (fun x -> fun (y::ys) -> (x + y)::ys) lis [0]
```

Passing functions as arguments

- Define `f`, `z` such that `fold_right f lis z` = the pair of lists `(l1,l2)` where `l1` contains the elements of `lis` that are < 0, and `l2` contains the rest

```
f = fun x ->
      fun (l1,l2) ->
        if x < 0
        then (x::l1,l2)
        else (l1,x::l2)
```

Passing functions as arguments

$\texttt{fold\_left : } (\alpha\texttt{->}\beta\texttt{->}\alpha)\texttt{->}\alpha\texttt{->}\beta\texttt{ list->}\alpha$

$\texttt{fold\_left f [}x_1\texttt{;}x_2\texttt{;...}x_n\texttt{] x}$

$\texttt{= f f(...(f z } x_1\texttt{) x2)...) } x_n$

`fold_left (+) lis 0` **=> sum of** `lis`

Defining higher-order functions

```
let rec map f lis =
  if lis = [] then []
  else f (hd lis)::map f (tl lis)

let rec fold_right f lis z =
  if lis = [] then z
  else f (hd lis)
    (fold_right f (tl lis) z)
```

Understanding higher-order functions

Two approaches: Substitution, or environment/closure model

Consider: let add1 = map (fun x -> x+1)

Returns: fun lis -> if lis = [] then []

else f (hd lis)::map f (tl lis)

But this has "f" as a free variable.

Question: when add1 is applied, where does the value of f come from?

Substitution model

Replace free variable with its value:

```
   map (fun x -> x+1)
 = fun lis -> if lis = [] then []
       else (fun x -> x+1) (hd lis)::map (fun x -> x+1) (tl lis)
```

(Note: no free variables any more.)

Environment/closure model

Put free variables in a data structure called an *environment*:

{f $\rightarrow$ fun x -> x+1}

Keep expression and environment together in a pair:

(fun lis -> if lis = [] then []
else f (hd lis)::map f (tl lis), {f $\rightarrow$ fun x -> x+1})

This pair is called a *closure*.

After applying map to the function, the value is always kept in the form of the closure, never as just the expression.