News

• Ravi is back as a TA, but he will be doing things remotely

• Grading all assignments

• Answering questions Webboard and email

• Rushi will do Section

max function

let max x y = if x < y then y else x;;

(* return max element of list l *)
let list_max l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (max h curr) t
  in
  helper 0 l;;

concat function

(* concatenate all strings in a list *)
let concat l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (curr ^ h) t
  in
  helper "" l;;
What's the pattern?

```
let list_max l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (max h curr) t
  in helper 0 l;;
```

```
let concat l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (curr ^ h) t
    | h::t -> helper (curr h) t
  in helper "" l;;
```

fold

```
(* to help us see the pattern: *)
let list_max l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (max h curr) t
  in helper 0 l;;

let rec fold f curr l =
  match l with
  [] -> curr
  | h::t -> fold f (f h curr) t;;
```

```
(* fold, the coolest function there is! *)
let rec fold f curr l =
  match l with
  [] -> curr
  | h::t -> fold f (f h curr) t;;
```
Examples of fold

```ocaml
let list_max = fold max 0;;
let concat = fold (^) "";;
let multiplier = fold (*) 1;;
```

Examples of fold

```ocaml
let fact n = multiplier (interval 1 n);;
```

Notice how all the recursion is buried inside two functions: `interval` and `fold`!
### Examples of fold

```ocaml
let cons x y = x::y;;
let f = fold cons [];;
(* same as: *)
  let f l = fold cons [] l *)
```

### Benefits of higher-order functions

Identify common computation “patterns”

- **Iterate** a function over a set, list, tree ...

- **Accumulate** some value over a collection

Pull out (factor) “common” code:

- **Computation Patterns**
- **Re-use** in many different situations

### Funcs taking/returning funcs

Higher-order funcs enable **modular code**

- Each part only needs **local** information

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses list</td>
<td></td>
</tr>
</tbody>
</table>

Uses meta-functions:
- map, fold, filter
  - With locally-dependent funs (lt h), square etc.
  - Without requiring Implement. details of data structure

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>list</td>
</tr>
</tbody>
</table>

Provides meta-functions:
- map, fold, filter
  - to traverse, accumulate over lists, trees etc.
  - Meta-functions don’t need client info
Different way of thinking

“Free your mind”
-Morpheus

- Different way of thinking about computation
- Manipulate the manipulators

Parametric types

aka: what’s up with those ’a ???

What is the deal with ’a ?

These meta-functions have strange types:

map: (’a → ’b) → ’a list → ’b list

filter: (’a → bool) → ’a list → ’a list

Why?

Polymorphism

• Poly = many, morph = kind

    let swap (x, y) = (y, x)

    ’a * ’b → ’b * ’a

• ’a and ’b are type variables!
• For-all types: For all ’a, ’b: ’a * ’b → ’b * ’a

• ’a,’b can be instantiated with any type:

    w/ int, string : int * string → string * int
    w/ char, int list : char * int list → int list * char
    w/ int → int , bool :(int → int) * bool → bool * (int → int)
Instantiation at Use

map: \((a \to b) \to a\ list \to b\ list\)

let f x = x + 10;;
let fm = map f;;

let f x = x^" like";;
let fm = map f ["cat"; "dog"; "burrito"];;

Instantiation at Use: be Careful

map: \((a \to b) \to a\ list \to b\ list\)

let f x = x^" like";;
let fm = map f [1;2;3;4];;

Polymorphic ML types

• Implicit for-all at the “left” of all types
  - Never printed out, or specified
  
  map: \(\forall a, b. (a \to b) \to a\ list \to b\ list\)

• Typing rule:

\[
\begin{array}{c}
e: \forall a. T \\
\hline
e: T [ a \mapsto T_f ] \text{ where } T_f \text{ does not contain type vars already in } T
\end{array}
\]
Example

\[ e : \forall \ 'a . T \]
\[ e : T [ 'a \mapsto T_f ] \]
\[ e_1 : T \to T_2 \]
\[ e_2 : T_1 \]
\[ e_1 \ e_2 : T_2 \]
\[ i : \text{int} \]
\[ s : \text{string} \]

Assume: \( \text{swap} : \forall \ 'a . \forall \ 'b . \ 'a * \ 'b \to \ 'b * \ 'a \)

Show: \( \text{swap}(1, "a") : \text{string} * \text{int} \)

Polymorphism enables Reuse

- Can reuse generic functions:

\[
\text{swap} : 'a * 'b \to 'b * 'a \\
\text{rev} : 'a \text{list} \to 'a \text{list} \\
\text{length} : 'a \text{list} \to \text{int} \\
\text{filter} : ('a \to \text{bool}) \to 'a \text{list} \to 'a \text{list} \\
\text{partition} : ('a \to \text{bool}) \to 'a \text{list} \to ('a \text{list} * 'a \text{list}) \\
\text{map} : ('a \to 'b) \to 'a \text{list} \to 'b \text{list}
\]

Not just functions ...

- Data types are also polymorphic!

\[
\text{type} 'a \text{list} = \\
\text{Nil} \\
| \text{Cons of ('a * 'a \text{list})}
\]

- Type is instantiated for each use:

\[
\text{Cons}(1,\text{Cons}(2,\text{Nil})) : \\
\text{Cons}("a",\text{Cons}("b",\text{Nil})) : \\
\text{Cons}((1,2),\text{Cons}((3,4),\text{Nil})) : \\
\text{Nil}:
\]
Not just functions ...

- Data types are also polymorphic!
  ```
  type 'a list =
  Nil
  | Cons of ('a * 'a list)
  ```

- Type is instantiated for each use:
  ```
  Cons(1,Cons(2,Nil)) : int list
  Cons("a",Cons("b",Nil)) : string list
  Cons((1,2),Cons((3,4),Nil)) : (int*int) list
  ```
  ```
  Nil : 'a list
  ```

Datatypes with many type variables

- Multiple type variables
  ```
  type ('a,'b) tree =
  Leaf of ('a * 'b)
  | Node of ('a,'b) tree * ('a,'b) tree
  ```

- Type is instantiated for each use:
  ```
  Leaf("joe",1) :
  Leaf("william",2) :
  Node(...,...) :
  ```
  ```
  Node(Leaf("joe",1),Leaf(3.14, "pi")):
  ```

Polymorphic Data Structures

- Container data structures independent of type!
  ```
  'a list
  ('a , 'b) Tree
  ('a , 'b) Hashtbl ...
  ```

- Appropriate type instantiated at each use:
  ```
  'a list
  ('a , 'b) Tree
  ('a , 'b) Hashtbl ...
  ```

- Appropriate type instantiated at use
  - No casting as in Java

- Static type checking catches errors early
  - Cannot add int key to string hashtable

- Generics: feature of Java,C#...
Other kinds of polymorphisms

• That was OCaml...

• But what about other kinds of polymorphisms..

Other kinds of polymorphisms

• Sub-type polymorphism
  
  ```c
  void f(Shape s)
  ```
  
  - Can pass in any sub-type of Shape

• Parametric polymorphism
  
  ```c
  void process_list_elems(list[T])
  ```
  
  - Can pass in ANY T
  - this is the kind in OCaml!

Other kinds of polymorphisms

• Bounded polymorphism
  
  - Like parametric, except can provide a bound
    ```c
    void process_list_elems(list[T])
    WHERE T <= Printable
    ```
  - But wait... isn’t this subtype polymorphism?
  - No, for example:
    ```c
    bool ShapeEq(T a, T b) WHERE T <= Shape
    ```
  - Can call on (Rect, Rect), (Circle, Circle) but not (Rect, Circle)

Summary of polymorphism

• Subtype

• Parametric

• Bounded
Back to OCaml

- Polymorphic types allow us to reuse code
- However, not always obvious from staring at code
- But... Types never entered w/ program!

Type inference

aka: how in the world does Ocaml figure out all the types for me ???

Example 1

```ocaml
let x = 2 + 3;;
let y = string_of_int x;;
```

Example 2

```ocaml
let x = 2 + 3;;
let inc y = x + y;;
```
Example 2

```ocaml
let x = 2 + 3;;
let inc y = x + y;;
```

Example 3

```ocaml
let foo x =
  let (y,z) = x in
  z-y;;
```

Example 3

```ocaml
let foo x =
  let (y,z) = x in
  z-y;;
```

Example 4

```ocaml
let rec cat l =
  match l with
  | [] -> ""
  | h::t -> h^(cat t)
```
Example 4

ML doesn’t know what the function does, or even that it terminates. ML only knows its type!

```ocaml
let rec cat l =
  match l with
  | [] -> ""
  | h::t -> h^(cat t)
```

Example 5

```ocaml
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t)
```

Inferring types with ‘a

- Introduce unknown type vars
- Figure out equalities that must hold, and solve these equalities
- Remaining types vars get a forall and thus become the ‘a, ‘b, etc.
Example 6

```ocaml
let compose (f, g) x = f (g x)
```

Example 7

```ocaml
let rec fold f cur l =
    match l with
    | [] -> cur
    | h::t -> fold f (f h cur) t
```