Another pattern: Accumulation

```ocaml
let max x y = if x > y then x else y ;
let listMax l =
    let rec help cur l =
        match l with
        | [] -> cur
        | h::t -> help (max cur h) t
    in
    helper 0 l;;

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

What’s the pattern?
Examples of fold

```ocaml
let listMax = fold max 0

let concat = Currying! This is a function! 
fold (^) ""
fold (*) 1

let multiplier = Currying! This is a function! 
Currying! This is a function! 
fold (*) 1
```

Funcs taking/returning funcs

Identify common computation “patterns”
- Filter values in a set, list, tree ...
- Iterate a function over a set, list, tree ...
- Map
- Accumulate some value over a collection
- Fold

Pull out (factor) “common” code:
- Computation Patterns
- Re-use in many different situations

Another fun function: “pipe”

```ocaml
let pipe x f = f x
let (|>) x f = f x

Compute the sum of squares of numbers in a list ?
let sumOfSquares xs = 
xs |> map (fun x -> x * x) 
|> fold_left (+) 0
```

Another Pattern: Synchronize

```ocaml
acquire(lock); S; release(lock);
synchronize(lock){
S;
}
synchronize(fun () -> … )
```
**Funcs taking/returning funcs**

Identify common computation “patterns”
- Filter values in a set, list, tree ...
- Convert a function over a set, list, tree ...
- 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

**Functions are “first-class” values**

- Arguments, return values, bindings ...
- What are the benefits ?

Parameterized, similar functions
(e.g. Testers)

Compose Functions:
Flexible way to build Complex functions from primitives.

**Parametric types**

aka: what’s up with those ’a ?

Everywhere:
- Javascript,
- Google/Mapreduce, Yahoo/Hadoop,
- C++0x
**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 -> 'b) -> 'a list -> 'b list
  ```

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

  `f : int -> int` first arg. of map: a->b

  **Instantiated:** ’a with int, ’b with int

  ```
  let f x = x^“ like”;;
  ```
  
  ```
  let fm = map f [1;2;3;4];;
  ```

  **f : string -> string** first arg. of map: a->b

  **Instantiated:** ’a with str, ’b with str’

  So, list must be ’a list = string list!

**Instantiation at Use: be careful**

- **map:** 
  
  ```
  ('a ! 'b) ! 'a list ! 'b list
  ```

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

  **f : string -> string** first arg. of map: a->b

  **Instantiated:** ’a with str, ’b with str’
Polymorphic ML types

- Poly = many, morph = kind
- Possible ML types:
  - tv = 'a | 'b | 'c | ...
  - $T = \text{int} \mid \text{bool} \mid \text{string} \mid \text{char} \mid ...$
  - $T_1 \times T_2 \times ... T_n \mid T_1 \rightarrow T_2 \rightarrow tv$

- Implicit for-all at the “left” of all types
  - Never printed out

  map: For all 'a, 'b : ('a ! 'b) ! 'a list ! 'b list

Polymorphism enables Reuse

- Can reuse generic functions:
  - map : 'a * 'b ! 'b * 'a
  - filter : ('a ! bool) ! 'a list ! 'a list
  - rev : 'a list ! 'a list
  - length : 'a list ! int
  - swap : 'a * 'b ! 'b * 'a
  - sort : ('a ! 'a ! bool) ! 'a list ! 'a list
  - fold: ...

- If function (algorithm) is independent of type, can reuse code for all types!

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

```haskell
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!
- Appropriate type is instantiated at each use:

```
'a list
('a , 'b) tree
('a , 'b) hashtbl ...
```

- Appropriate type instantiated at use
  - No unsafe casting as in C++/Java

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

- Generics: in Java,C#,VB (borrowed from ML)

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 proc_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 proc_elems(list[T]) T extends Printable
  ```
  - Hey... isn’t this subtype polymorphism?
  - No, for example:
    ```c
    bool ShapeEq(T a, T b) T extends Shape
    ```
  - Can call on
    - (Rect, Rect)
    - (Circle, Circle)
  - But not (Rect, Circle)

Summary of polymorphism

- Subtype
- Parametric
- Bounded = Parametric + Subtype  
  (In Java/C#)

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 ???

Polymorphic Types

- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness?
- Types (almost) never entered w/ program!

Polymorphic Type Inference

- Computing the types of all expressions
  - At compile time: Statically Typed

- Each binding is processed in order
  - Types are computed for each binding
  - For expression and variable bound to
  - Types used for subsequent bindings

- How is this different from values?
Values NOT computed statically (e.g. fun values)

Polymorphic Type Inference

- Every expression accepted by ML must have a valid inferred type

- Can have no idea what a function does, but still know its exact type

- A function may never (or sometimes terminate), but will still have a valid type
Example 1

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

Example 2

```ml
let x = 2 + 3;;
let y = string_of_int x;;
let inc y = x + y;;
```

Example 3

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

Example 4

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

*ML doesn’t know what the function does, or even that it finishes only its type!*
Example 4

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

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

string list -> string
```

Example 5

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

('a -> 'b) -> 'a list -> 'b list
```

Introduce unknown tyvar: Unify, solve, Remaining tyvar gets a “forall”

Example 6

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

('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

Example 7

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

('a * 'b -> 'c) -> 'b -> 'a list -> 'c
```
let fool f g x =  
  if f x  
  then x  
  else g x

let foo2 f g x =  
  if f x  
  then x  
  else foo2 f g (g x)