

CSE 130: Fall 2011

Programming Languages

*Lecture 7: Polymorphism
(or, whats the deal with 'a)*

Ranjit Jhala
UC San Diego



News

- Midterm Nov 1
 - In class
 - Open book etc.
 - Practice materials on Webpage

Another pattern: Accumulation

```
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

let listMax = **fold** max 0

Currying! This is a function!

let concat = **fold** (^) ""

Currying! This is a function!

Pick correct
base case!

let multiplier = **fold** (*) 1

Currying! This is a function!

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”

```
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
```

Tail Rec ?

Another Pattern: Synchronize

```
acquire(lock);  
S;  
release(lock);
```

```
synchronize(lock) {  
    S;  
}
```

```
let synchronize lock s =  
    acquire(lock);  
    s();  
    release(lock)  
  
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 ... map
- Iterate a function over a set, list, tree ...
- Accumulate some value over a collection fold

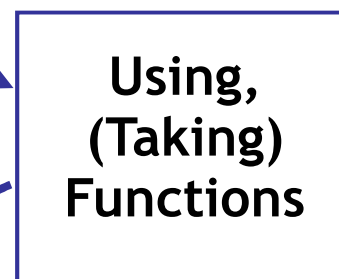
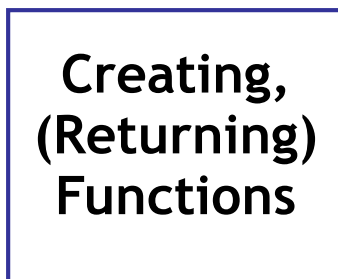
Pull out (factor) “common” code:

- 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)*

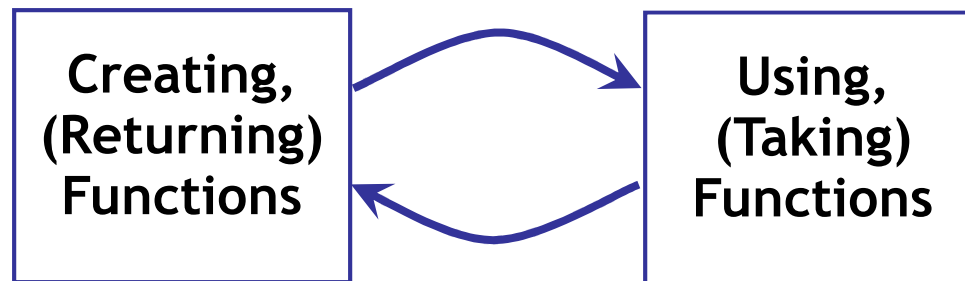


*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*

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

Functions are “first-class” values

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



Everywhere:

- Javascript,
- Google/Mapreduce, Yahoo/Hadoop,
- C++0x

Parametric types

aka: what's up with those 'a' ?

Java/C#

What is the deal with 'a' ?

These meta-functions have strange types:

map: $(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$

filter: $(\text{'a} \rightarrow \text{bool}) \rightarrow \text{'a list} \rightarrow \text{'a list}$

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: $('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

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

```
let fm = map f;;
```

$f : \text{int} \rightarrow \text{int}$ first arg. of map: $a \rightarrow b$
Instantiated : 'a with int, 'b with int

```
let f x = x ^ " like";;
```

```
let fm = map f ["cat"; "dog"; "burrito"];;
```

$f : \text{string} \rightarrow \text{string}$ first arg. of map: $a \rightarrow b$
Instantiated : 'a with str, 'b with str

Instantiation at Use: be careful

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

```
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 !

Polymorphic ML types

- Poly = *many*, morph = *kind*
- Possible ML types:

$tv = 'a \mid 'b \mid 'c \mid \dots$

$T = int \mid bool \mid string \mid char \mid \dots$

$T_1 * T_2 * \dots T_n \mid T_1 ! T_2 \mid tv$

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

map: *For all 'a , 'b :* $('a ! 'b) ! 'a \text{ list } ! 'b \text{ list}$

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)) :

Cons("a", Cons("b", Nil)) :

Cons((1,2), Cons((3,4), Nil)) :

Nil : 'a list

Cons(2, Nil) : int list

int int list
Cons(1, —)

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
```

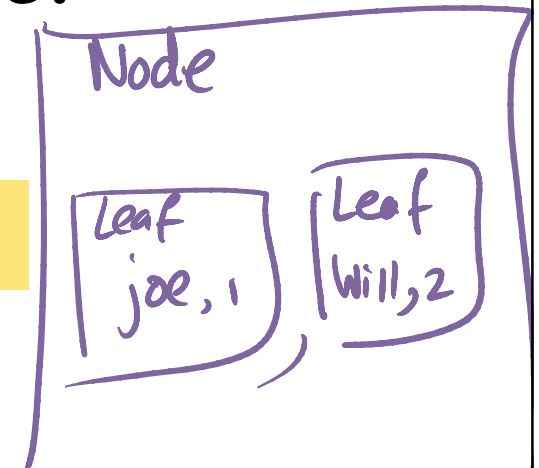
- Type is instantiated for each use:

Leaf("joe",1) :

Leaf("william",2) :

Node(...,...) :

~~*Node*(*Leaf*("joe",1),*Leaf*(3.14,"pi")):~~



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) : (string,int) tree

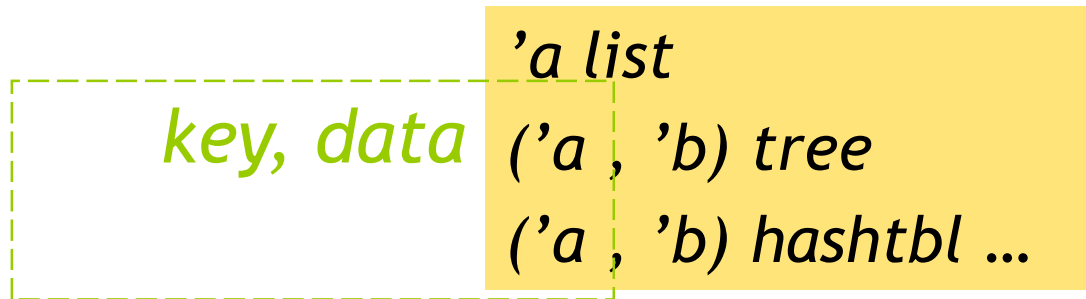
Leaf("william",2) : (string,int) tree

Node(...,...) : (string,int) tree

Node(Leaf("joe",1),Leaf(3.14, "pi"))

Polymorphic Data Structures

- Container data structures independent of type !
- Appropriate type is **instantiated** at each **use**:



- 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)

Java 1.5

Other kinds of polymorphisms

- That was OCaml...
- But what about other kinds of polymorphisms..

Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

- Can pass in any sub-type of Shape

- Parametric polymorphism

`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
`void proc_elems(list[T]) T extends Printable`
 - Hey... isn't this subtype polymorphism?
 - No, for example:
`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, *Value* but still know its exact type
- A function may never (or sometimes terminate), but will still have a valid type *Value undefined*

Example 1

```
let x = 2 + 3;;
```

```
let y = string_of_int x;;
```

Example 2

```
let x = 2 + 3;;
```

```
let y = string_of_int x;;
```

```
let inc y = x + y;;
```

Example 3

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

Example 4

```
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!

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

string list -> string

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

Example 5

$('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

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

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

Example 6

$(b \rightarrow c) * (a \rightarrow b) \rightarrow (a \rightarrow c)$

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

Example 7

$(\text{'a} * \text{'b} \rightarrow \text{'b}) \rightarrow \text{'b} \rightarrow \text{'a list} \rightarrow \text{'b}$

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

```
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)
```