Parametric types

aka: what’s up with those ‘a ???

What is the deal with ’a ?

These meta-functions have strange types:

map: $(a \to b) \to [a] \to [b]

filter: $(a \to bool) \to [a] \to [a]

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] \to [b]

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] \to [b]

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: For all ’a , ’b . $(a \to b) \to [a] \to [b]

• Typing rule:

  $e: \forall \ 'a. \ T$
Polymorphic ML types

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

  \[
  \text{map: For all } 'a, 'b: ('a \to 'b) \to 'a \text{ list } \to 'b \text{ list}
  \]

- Typing rule:

  \[
  \begin{align*}
  e: \forall 'a. T & \\
  e: T ['a \mapsto T_{conc}] & \text{ where } T_{conc} \text{ is a concrete type (no 'a or 'b, etc)}
  \end{align*}
  \]

Example

- Assumption:
  \[
  \begin{align*}
  \text{swap: } \forall 'a \cdot \forall 'b: 'a * 'b \to 'b * 'a
  \end{align*}
  \]

- Typing rule:

  \[
  \begin{align*}
  e: \forall 'a. T & \\
  e: T ['a \mapsto T_{conc}] & \text{ where } T_{conc} \text{ is a concrete type (no 'a or 'b, etc)}
  \end{align*}
  \]

Polymorphism enables Reuse

- Can reuse generic functions:

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

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:

  \[
  \begin{align*}
  \text{Cons(1,Cons(2,Nil))} : \text{ int list} \\
  \text{Cons("a",Cons("b",Nil))} : \text{ string list} \\
  \text{Cons((1,2),Cons((3,4),Nil))} : \text{ (int} *\text{int) list} \\
  \text{Nil : 'a list}
  \end{align*}
  \]
Not just functions ...

• Data types are also polymorphic!

```java
type 'a list =
    Nil |
    Cons of ('a * 'a list)
```

• In Java:

```java
interface List<T>;
class Nil<T> implements List<T> {
    List<T> next;
}
class Cons<T> implements List<T> {  
    T data;
    List<T> next;
}
```

Datatypes with many type variables

• Multiple type variables

```java
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) Hashbl ...
```

• Appropriate type instantiated at use
  - No casting

• 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**
  - 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!

- **Bounded polymorphism**
  - Like parametric, except can provide a bound
  - void proc elems(list[T]) WHERE T <= Printable
  - In Java syntax:
    <T extends Printable> void p(list<T> l) {...}

- Hey... isn't this subtype polymorphism?
  - Can't I just do?
    void proc elems(list[Printable])
  - Yes, in this case, but in general...

- **Bounded polymorphism**
  - Say we have:
    T print_elem(T) WHERE T <= Printable
  - and we have
    - a Car car which is printable, and
    - a Shark shark which is printable
  - The following typechecks:
    - print_elem(car).steering_wheel
    - print_elem(shark).teeth
  - But not if print_elem returns Printable

- **Bounded polymorphism**
  - Or as another example:
    bool ShapeEq(T a, T b) WHERE T <= Shape
  - Can call on
    - (Rect, Rect)
    - (Circle, Circle)
  - But not (Rect, Circle)
F-bounded polymorphism

- Comparable types and sort on them

One option:

```java
interface Comparable { bool lt(Object); }
void sort(list<Comparable> l) { ... }
```

But, this leads to several problems:

1. Everything is comparable to everything
   - Leads to annoying instanceof tests in `lt`
   - Even if you have `bool lt(Comparable)`

Another option:

```java
interface Comparable<T> { bool lt(T); }
Class Dog extends Comparable<Dog> { bool lt(Dog){..} }
Class Cat extends Comparable<Cat> { bool lt(Cat){..} }
```

But now what does sort take?
F-bounded polymorphism

- Another option:
  
  ```java
  interface Comparable<T> { bool lt(T); }
  Class Dog extends Comparable<Dog> { bool lt(Dog){..} }
  Class Cat extends Comparable<Cat> { bool lt(Cat){..} }
  ```

- But now what does sort take?
  - Easy but doesn’t quite work:
    ```java
    void sort(list<Comparable<Object> > l)
    ```
  - F-bound:
    ```java
    void sort(list<T extends Comparable<T> > l) {
      ... l.get(i).lt(l.get(j) ...
    }
    ```

Summary of polymorphism

- Subtype
- Parametric
- Bounded
- F-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 ???

Inferring types

- 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 1

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

```ml
let x = 2 + 3;;
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 terminates.
ML only knows its type!
Example 5

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

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
```