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

• ‘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;;

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

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];;

Polymorphic ML types

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

map: For all ‘a , ‘b . (‘a → ‘b) → ‘a list → ‘b list

• Typing rule:

\[ e: \forall \xi \phi \square \]
Polymorphic ML types

• Implicit for all at the “left” of all types
  - Never printed out, or specified
  
  map: For all 'a, 'b: ('a → 'b) → 'a list → 'b list

• Typing rule:

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

Example

\[
\begin{align*}
  e: \forall \alpha. T \\
  e: T \left[ \alpha \mapsto \text{conc} \right]
\end{align*}
\]

Polymorphism enables Reuse

• Can reuse generic functions:

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

Not just functions ...

• Data types are also polymorphic!

\[
\begin{align*}
  \text{type 'a list} = \\
  \text{Nil} \mid \text{Cons of ('a * 'a list)}
\end{align*}
\]

• Type is instantiated for each use:

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

- Data types are also polymorphic!

```
// In Java:
interface List <T> {
    class Nil <T> implements List<T> {
    }
    class Cons <T> implements List<T> {
        T data;
        List<T> next;
    }
}
```

Datatypes with many type variables

- Multiple type variables

```
// Type is instantiated for each use:
Leaf("joe",1) : (string, int) tree
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:

- 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!

Other kinds of polymorphisms

• 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...

Other kinds of polymorphisms

• Bounded polymorphism
  - Or as another example:
    ShapeEq(Shape, Shape)
    (Rect, Rect)
  - Can call on
    • (Rect, Rect)
    • (Circle, Circle)
  - But not (Rect, Circle)

Other kinds of polymorphisms

• 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

F-bounded polymorphism

• Comparable types and sort on them
**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:
  - Everything is comparable to everything
  - Can accidentally override the wrong `lt`, for example in `Cat` class, define `lt(Cat)`

**Another option:**

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

**Another option:**

```java
interface Comparable<T> { bool lt(T); }
Class Dog extends Comparable<Car> { 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)
    void sort(list<T extends Comparable <T> > l) {
      ...
    }
    ```
  - F-bound:
    ```java
    void sort(list<T extends Comparable <T> > l) {
      ...
    }
    ```

**Summary of polymorphism**

- Subtype
- Parametric
- Bounded
- F-bounded