Parametric types

aka: what’s up with those 'a ???

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

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

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

• Typing rule:

\[ e : \forall \text{'a}.T \]
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:

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

Example

\[
\begin{align*}
e & : \forall \ 'a, 'T \\
& \text{T} : \exists \ 'a \rightarrow \text{T}_{\text{conc}} \\
e1 & : 'T1 \rightarrow 'T2 \\
e2 & : 'T1 \\
\text{Assume: swap: } \forall \ 'a, 'b. \ 'a * 'b \rightarrow 'b * 'a
\end{align*}
\]

\[
\begin{align*}
i & : \text{int} \\
\text{Show: swap(1,"a") : string * int}
\end{align*}
\]

Polymorphism enables Reuse

- Can reuse generic functions:

  • swap: 'a * 'b → 'b * 'a
  • rev: 'a list → 'a list
  • length: 'a list → int
  • filter: ('a → bool) → 'a list → 'a list
  • partition: ('a → bool) → 'a list → ('a list * 'a list)
  • map: ('a → 'b) → 'a list → 'b list

Not just functions ...

- Data types are also polymorphic!

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

- Type is instantiated for each use:

\[
\begin{align*}
\text{Cons}(1,\text{Cons}(2,\text{Nil})) : \text{int list} \\
\text{Cons("a",Cons("b",\text{Nil})) : string list} \\
\text{Cons((1,2),Cons((3,4),\text{Nil})) : (int*int) list} \\
\text{Nil : 'a list}
\end{align*}
\]

Not just functions ...

- Data types are also polymorphic!

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

- Type is instantiated for each use:

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

• Data types are also polymorphic!

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

• In Java:

\[
\text{interface List }\langle T \rangle ; \\
\text{class Nil }\langle T \rangle \text{ implements List }\langle T \rangle ; \\
\text{class Cons }\langle T \rangle \text{ implements List }\langle T \rangle \{ \\
\text{T data;} \\
\text{List }\langle T \rangle \text{ next;} \\
\}
\]

Datatypes with many type variables

• Multiple type variables

\[
\text{type } ('a,'b) \text{ tree } = \\
\text{Leaf of } ('a \times 'b) \\
\text{| Node of } ('a,'b) \text{ tree } * ('a,'b) \text{ tree}
\]

• Type is instantiated for each use:

\[
\text{Leaf("joe",1)} : ('a,'b) \text{ tree} \\
\text{Leaf("william",2)} : ('a,'b) \text{ tree} \\
\text{Node(....,...)} : ('a,'b) \text{ tree}
\]

\[
\text{Node(Leaf("joe",1),Leaf(3.14, "pi"))}:
\]

Polymorphic Data Structures

• Container data structures independent of type!
  • Appropriate type is instantiated at each use:
    \[
    \text{'a list} \\
    \text{('a , 'b) Tree} \\
    \text{('a , 'b) Hashtbl ...}
    \]
  • 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) {...}

• 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

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
  - 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:
    bool ShapeEq(T a, T b) WHERE T <= Shape
  - Can call on
    • (Rect, Rect)
    • (Circle, Circle)
  - But not (Rect, Circle)
F-bound 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)`

2. 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<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

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

Example 2

```
T_x = int ≤
T_y = int ≤
T_inc = T_y → T_int ≤
T_z = int × int → int

For all equals any types: T_x = int, T_y = int
```

Example 3

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

Example 3

```
\[ \begin{array}{l}
  T_x = \text{int} \\
  T_y = \text{int} \\
  T_z = \text{int} \\
  T_{yx} = \text{int} \\
  T_{xz} = \text{int} \\
  T_{yz} = \text{int} \\
  T_{x(y,z)} = \text{int} \\
  \end{array} \]
```

Example 4

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

Example 4

```
\[ \begin{array}{l}
  T_{\text{cat}} = T_1 \rightarrow T_\text{cat} \\
  T_1 = \text{a list} \\
  T_{h::t} = T_h \rightarrow T_{\text{cat}} \\
  \end{array} \]
```

ML doesn't know what the function does, or even that it terminates. ML only knows its type!
Example 5

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

Example 6

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

Example 7

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