Parametric types

aka: what’s up with those ’a ???
What is the deal with ‘a’?

These meta-functions have strange types:

**map**: \((\texttt{‘}a\rightarrow \texttt{‘}b) \rightarrow \texttt{‘}a\ \texttt{list} \rightarrow \texttt{‘}b\ \texttt{list}\)

**filter**: \((\texttt{‘}a\rightarrow \texttt{bool}) \rightarrow \texttt{‘}a\ \texttt{list} \rightarrow \texttt{‘}a\ \texttt{list}\)

Why?
Polymorphism

- Poly = *many*, morph = *kind*

\[
\text{let } \text{swap} \ (x, y) = (y, x) \quad \text{For all } 'a, 'b: 'a * 'b \rightarrow 'b * 'a
\]

- 'a and 'b are type variables!

- For-all types: 

\[
\text{For all } 'a, 'b: 'a * 'b \rightarrow 'b * 'a
\]

- 'a, 'b can be instantiated with any type:

\[
\begin{align*}
\text{w/ int, string} & : \text{int} \times \text{string} \rightarrow \text{string} \times \text{int} \\
\text{w/ char, int list} & : \text{char} \times \text{int list} \rightarrow \text{int list} \times \text{char} \\
\text{w/ int--int, bool} & : (\text{int} \rightarrow \text{int}) \times \text{bool} \rightarrow \text{bool} \times (\text{int} \rightarrow \text{int})
\end{align*}
\]
Instantiation at Use

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

```ocaml
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:** \((\mathbf{a} \to \mathbf{b}) \to \mathbf{a} \text{ list} \to \mathbf{b} \text{ list}\)

```
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

  \[
  \text{map: } \forall \ 'a, 'b. \ (\ 'a \rightarrow 'b) \rightarrow 'a \ \text{list} \rightarrow 'b \ \text{list}
  \]

• Typing rule:

  \[
  e: \forall 'a. T \\
  \]
Polymorphic ML types

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

  map: \( \text{For all } 'a, 'b . \quad ('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list} \)

• Typing rule:

\[
\begin{array}{c}
\text{e: } \forall 'a. T \\
\hline
\text{e: } T [ 'a \mapsto T_{\text{conc}} ] \text{ where } T_{\text{conc}} \text{ is a concrete type (no } 'a \text{ or } 'b, \text{ ect)}
\end{array}
\]
Example

\[
\begin{align*}
  e & : \forall 'a. T \\
  \Rightarrow & \quad e : T [ \ 'a \mapsto T_{\text{conc}} ] \\
  e_1 & : T_1 \to T_2 \\
  e_2 & : T_1 \\
  i & : \text{int} \\
  s & : \text{string} \\
\end{align*}
\]

Assume: \( \text{swap} : \forall 'a. \forall 'b. \ 'a * 'b \to \ 'b * 'a \) 

Show: \( \text{swap}(1, "a") : \text{string} * \text{int} \)
Example

Assume:

\[
\text{swap}: \forall \ 'a \ . \ \forall \ 'b . \ 'a * 'b \rightarrow 'b * 'a
\]

Show:

\[
\text{swap}(1,"a") : \text{string} * \text{int}
\]
Polymorphism enables Reuse

- Can reuse generic functions:

  swap : `'a * 'b → 'b * 'a
  \texttt{rev}: \ 'a list → \ 'a list
  \texttt{length}: \ 'a list → \ int
  \texttt{filter}: (\ 'a → bool) → \ 'a list → \ 'a list
  \texttt{partition}: (\ 'a → bool) → \ 'a list → (\ 'a list * \ 'a list)
  \texttt{map}: (\ 'a → \ 'b) → \ 'a list → \ 'b list
Not just functions ...

• Data types are also polymorphic!

```ocaml
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 :
```
Not just functions ...

• Data types are also polymorphic!

```ocaml
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
```
Not just functions ...

• Data types are also polymorphic!

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

• In Java:
Not just functions ...

• Data types are also polymorphic!

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

• In Java:

```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

```plaintext
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

```haskell
type ('a, 'b) tree =
  Leaf of ('a * 'b)
|  Node of ('a, 'b) tree * ('a, 'b) tree
```

- Type is instantiated for each use:

  ```haskell```
  ```
  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:

  "a list
  ("a , "b) Tree
  ("a , "b) Hashtable ...

- 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
  \[
  \text{void } f(\text{Shape } s)
  \]
  - Can pass in any sub-type of Shape

• Parametric polymorphism
  \[
  \text{void proc_elems(list}[T]\text{])}
  \]
  - can pass in ANY T
  - this is the kind in OCaml!
Other kinds of polymorphisms

- Bounded polymorphism
  - Like parametric, except can provide a bound
    ```java
    void proc_elems(list[T]) WHERE T <= Printable
    ```
  - In Java syntax:
    ```java
    <T extends Printable> void p(list<T> l) {...}
    ```
Other kinds of polymorphisms

• Bounded polymorphism
  - Like parametric, except can provide a bound
  ```java
  void proc_elems(list[T]) WHERE T <= Printable
  ```
  - In Java syntax:
  ```java
  <T extends Printable> void p(list<T> l) {...}
  ```
  - Hey... isn’t this subtype polymorphism?
  - Can’t I just do?
  ```java
  void proc_elems(list[Printable])
  ```
  - Yes, in this case, but in general...
Other kinds of polymorphisms

• Bounded polymorphism
  - Say we have:
    \[ T \, print\_elem(T) \, \text{WHERE} \, T \, \leq \, \text{Printable} \]
  - and we have
    • a \textbf{Car} \textit{car} which is printable, and
    • a \textbf{Shark} \textit{shark} which is printable
Other kinds of polymorphisms

• Bounded polymorphism
  - Say we have:
    \[ T \text{ print}_\text{elem}(T) \text{ WHERE } T \leq \text{ Printable} \]
  - and we have
    • a Car car which is printable, and
    • a Shark shark which is printable
  - The following typechecks:
    • \text{print}_\text{elem}(\text{car}).steering\_wheel
    • \text{print}_\text{elem}(\text{shark}).teeth
  - But not if \text{print}_\text{elem} returns \text{ Printable}
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-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
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)`
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
  (2) Can accidentally override the wrong `lt`
    - for example in `Cat` class, define `lt(Cat)`
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){..} }
```
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?
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

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

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

let \( x = 2 + 3 ;; \)
let \( \text{inc} \ y = x + y ;; \)

For fn call equate arg types: \( T_x = \text{int}, T_y = \text{int} \)
\( \text{inc} = \text{int} \rightarrow \text{int} \)
\( T_{\text{inc}} = \text{int} \rightarrow \text{int} \)
\( T_x = \text{int} \)
\( T_y \)
Example 3

\[
\text{let } \text{foo } x = \\
\text{let } (y,z) = x \text{ in } \\
z-y;;
\]
Example 3

\[
\text{let } x =  \\
\text{let } (y, z) = x \text{ in }  \\
z - y;;
\]
Example 4

let rec cat l =
  match l with
  | [] -> ""
  | h::t -> h^(cat t)
Example 4

```ml
let rec cat l =
  match l with
  | [] -> ""  // 3
  | 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 5

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

1. \( T_{map} = T_f \rightarrow T_e \rightarrow T_{map\text{-}\text{out}} \)

2. \( T_{::} = 'a \rightarrow 'a\text{ list} \rightarrow 'a\text{ list} \quad Ta=T_h \)
   \( T_f=T_h\text{ list} \quad T_e=T_h\text{ list} \)

3. \( T_f = T_h \rightarrow T_{f\text{-}\text{out}} \)

4. \( T_{map\text{-}\text{out}} = T_{f\text{-}\text{out}}\text{ list} \)
   \( T_{map} = (T_h \rightarrow T_{f\text{-}\text{out}}) \rightarrow T_h\text{ list} \rightarrow T_{f\text{-}\text{out}}\text{ list} \)
   \( = (a \rightarrow 'b) \rightarrow 'a\text{ list} \rightarrow 'b\text{ list} \)
Example 6

```haskell
let compose (f,g) x = f (g x)
```
Example 6

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

1. $\Gamma_{\text{comp}} = \Gamma_f \times \Gamma_g \rightarrow \Gamma_x \rightarrow \Gamma_{\text{comp output}}$
2. $\Gamma_f = \Gamma_{g\text{ret}} \rightarrow \Gamma_{\text{comp output}}$
3. $\Gamma_g = \Gamma_x \rightarrow \Gamma_{\text{ret}}$

$\Gamma_{\text{comp}} = (\Gamma_{\text{ret}} \rightarrow \Gamma_{\text{comp output}}) \times (\Gamma_x \rightarrow \Gamma_{\text{ret}}) \rightarrow \Gamma_x \rightarrow \Gamma_{\text{comp output}}$

('a -> 'b) $\times$ ('c -> 'a) $\rightarrow$ 'c $\rightarrow$ 'b
Example 7

```ocaml
let rec fold f cur l =
  match l with
  | [] -> cur
  | h::t -> fold f (f h cur) t
```
Example 7

(1) let rec fold f cur l =

match l with

(2) [] -> cur

(3) h::t -> fold f (f h cur) t

(1) \( T_{\text{fold}} = \overline{T_f} \rightarrow T_{\text{tm}} \rightarrow T_e \rightarrow T_{\text{fold\_nt}} \)

(2) \( T_{\text{tm}} = T_{\text{fold\_nt}} \)

(3) \( T_e = T_h \text{\_list} \quad T_f = T_h \text{\_list} \)

(4) \( T_f = T_h \rightarrow T_{\text{tm}} \rightarrow T_{\text{tm}} \)

\[
T_{\text{fold}} = (T_h \rightarrow T_{\text{tm}} \rightarrow T_{\text{tm}}) \rightarrow T_{\text{tm}} \rightarrow T_h \text{\_list} \rightarrow T_{\text{tm}}
\]

(\( 'a \rightarrow 'b \rightarrow 'b \) \( \rightarrow 'b \rightarrow 'a \text{\_list} \rightarrow 'b \))