Polymorphism

Polymorphism

- 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) {...}
```

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 on next slide…

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

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 with bounded poly:
    - `print_elem(car).steering_wheel`
    - `print_elem(shark).teeth`
  - But not if we use subtype poly (ie: if `print_elem` returns `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)`**
  - If we instead used Subtype poly would have:

    `bool ShapeEq(Shape a, Spape b)`
  - And this would allow **`(Rect, Circle)`**

F-bounded polymorphism

- Comparable types and sort on them

F-bounded polymorphism

- Comparable types and sort on them
- One option:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

    ```
    void sort(list<Comparable<Object> >l)
    ```

  - F-bound:

    ```
    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

```
let x = 2 + 3;;

let y = string_of_int x;;
```

Example 2

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

Example 2

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

$T_x = int$

T_y

$T_{inc} = T_y \to T_{ret}$

$T_+ = int \times int \to int$

$T_{inc} = int \to int$

For fn call equate arg types: $T_x = int$, $T_y = int$

$T_{ret} = int$

Example 3

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

Example 3

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

① T_x

① $T_{foo} = T_x \to T_{ret}$

② $T_x = T_y * T_z$

③ $T_{-} = int * int \to int$

$T_z = int$

$T_y = int$

$T_{ret} = int$

$T_{foo} = int * int \to int$

Example 4

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

Example 4

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

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

① $T_{cat} = T_l \to T_{ret}$ → $T_{cat} = \text{string list} \to \text{string}$

② $T_{::} = \underbrace{'a}_{T_h} * \underbrace{'a\ list}_{T_t} \to \underbrace{'a\ list}_{T_l}$

$T_l = T_h\ list$

$T_t = T_h\ list$

③ $T_{\wedge} = \underbrace{\text{string}}_{T_h} * \underbrace{\text{string}}_{T_{ret}} \to \underbrace{\text{string}}_{T_{ret}}$

$T_{ret} = \text{string}$

$T_h = \text{string}$

Example 5

```
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 $T_{map} = T_f \to T_l \to T_{mapret}$

② $T_{::} = {'a} * {'a}\ list \to {'a}\ list \qquad T_a = T_h$

$T_h \qquad T_t = T_h\ list \qquad T_l = T_h\ list$

③ $T_f = T_h \to T_{fret}$

④ $T_{mapret} = T_{fret}\ list$

$T_{map} = (T_h \to T_{fret}) \to T_h\ list \to T_{fret}\ list$

$= ({'a} \to {'b}) \to {'a}\ list \to {'b}\ list$

Example 6

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

Example 6

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

① $T_{comp} = T_f * T_g \rightarrow T_x \rightarrow T_{comp\ ret}$

② $T_f = T_{g\ ret} \rightarrow T_{comp\ ret}$

③ $T_g = T_x \rightarrow T_{g\ ret}$

$T_{comp} = (T_{g\ ret} \rightarrow T_{comp\ ret}) * (T_x \rightarrow T_{g\ ret}) \rightarrow T_x \rightarrow T_{comp\ ret}$

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

Example 7

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

Example 7

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

(1) $T_{fold} = T_f \to T_{cur} \to T_l \to T_{foldret}$

(2) $T_{cur} = T_{foldret}$

(3) $T_l = T_h \text{ list} \qquad T_t = T_h \text{ list}$

(4) $T_f = T_h \to T_{cur} \to T_{cur}$

$$T_{fold} = (T_h \to T_{cur} \to T_{cur}) \to T_{cur} \to T_h \text{ list} \to T_{cur}$$

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