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)`

7

F-bounded polymorphism

- Comparable types and sort on them

8

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

9

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)`

10

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)`

11

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){..} }
```

12

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?

13

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) ...
}
```

14

Summary of polymorphism

- Subtype
- Parametric
- Bounded
- F-bounded

15

Back to OCaml

- Polymorphic types allow us to reuse code
- However, not always obvious from staring at code
- But... Types never entered w/ program!

16

Type inference

aka: how in the world does Ocaml figure out all the types for me ???

17

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.

18

Example 1

```
let x = 2 + 3;;

let y = string_of_int x;;
```

19

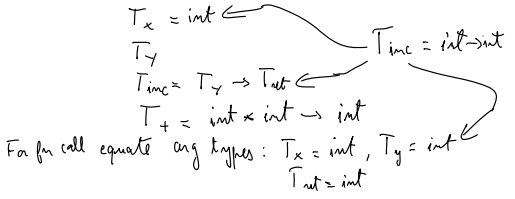
Example 2

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

20

Example 2

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



21

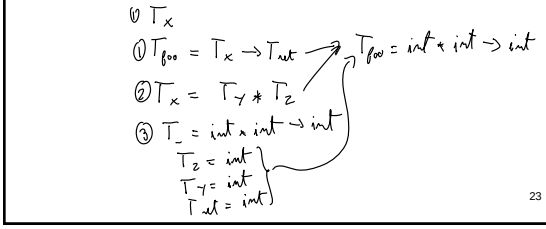
Example 3

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

22

Example 3

```
let ①foo x =②
  let (y,z) = x in
③ z-y;;
```



23

Example 4

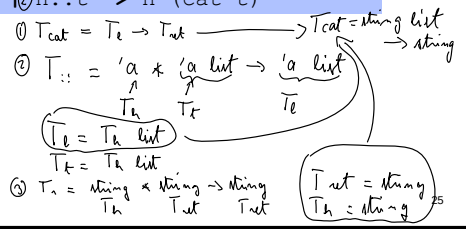
```
let rec cat l =
  match l with
    [] -> ""
  | h::t -> h^(cat t)
```

24

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)
```



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 Tmap = Tf → Te → Tmapret

② T:: = 'a * 'a list → 'a list Ta = Th
Th Tt = Th list Te = Th list

③ Tf = Th → Tfret

④ Tmapret ≃ Tfret list
Tmap = (Th → Tfret) → Th list → Tfret list
= ('a → 'b) → 'a list → 'b list

Example 6

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


Example 6

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

① ② ③

① Tcomp = Tf * Tg → Tx → Tcompret

② Tf = Tgret → Tcompret

③ Tg = Tx → Tgret

Tcomp = (Tgret → Tcompret) * (Tx → Tgret) → Tx → Tcompret
('a → 'b) * ('c → 'a) → 'c → '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 $T_{fold} = T_f \to T_{cur} \to T_e \to T_{fold\,out}$

② $T_{cur} = T_{fold\,out}$

③ $T_l = T_h\ list \quad T_t = T_h\ list$

④ $T_f = T_h \to T_{cur} \to T_{cur}$

$T_{fold} = (T_h \to T_{cur} \to T_{cur}) \to T_{cur} \to T_h\ list \to T_{cur}$
$('a \to 'b \to 'b) \to 'b \to 'a\ list \to 'b$