

CSE 130: Fall 2011

Programming Languages

Lecture 9: Type Inference

Red

Ranjit Jhala
UC San Diego



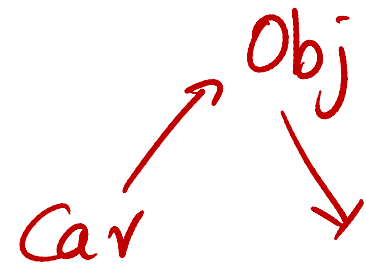
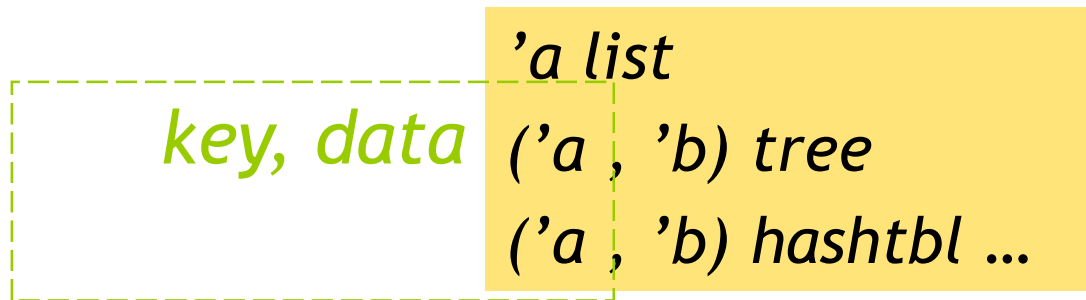
News

*Lex/Yacc }
Parsing }*

- PA4 Updated
 - Do second question before midterm ...
- “Late Day” = 1 second after 5pm
- Midterm next week in class

Polymorphic Data Structures

- Container data structures independent of type !
- Appropriate type is **instantiated** at each **use**:



- Appropriate type instantiated at use
 - No **unsafe casting** as in C++/Java

o.drive()

- **Static type checking catches errors** early
 - Cannot add **int** key to **string** hashtable
- **Generics**: in Java, C#, VB (borrowed from ML)

Other kinds of polymorphisms

- That was OCaml...
- But what about other kinds of polymorphisms..

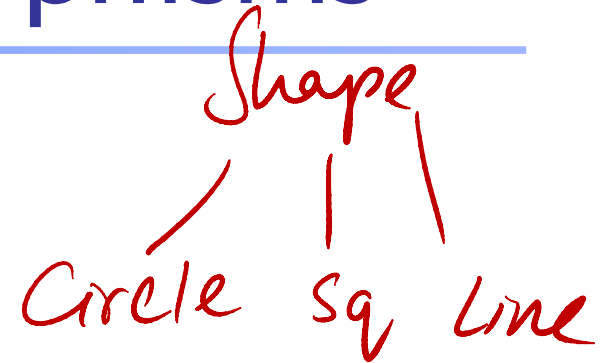
Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

- Can pass in any sub-type of Shape

f Circle c



Circle c = new Circle(c)
f(c)

- Parametric polymorphism

`void proc_elems(list[T])`

- can pass in ANY T

- this is the kind in OCaml! *<T> f(T s)*

Other kinds of polymorphisms

- Bounded polymorphism
 - Like parametric, except can provide a bound
`void proc_elems(list[T]) T extends Printable`
 - Hey... isn't this subtype polymorphism?
 - No, for example:
`bool ShapeEq(T a, T b) T extends Shape`
 - Can call on
 - `(Rect, Rect)`
 - `(Circle, Circle)`
 - But not `(Rect, Circle)`

Summary of polymorphism

- Subtype

- Parametric *Type Classes : Haskell*

- Bounded = Parametric + Subtype
(In Java/C#)

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 ???

Polymorphic Types

- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness ?
- Types (almost) never entered w/ program!

Polymorphic Type Inference

- Computing the types of all expressions
 - At **compile** time : **Statically Typed**
- Each binding is processed in order
 - Types are computed for each binding
 - For expression and variable bound to
 - Types used for subsequent bindings
- How is this different from values ?

Values NOT computed statically (e.g. fun values)

Polymorphic Type Inference

- Every expression accepted by ML must have a valid inferred type
- Can have no idea what a function does, *Value* but still know its exact type
- A function may never (or sometimes terminate), but will still have a valid type *Value undefined*

Example 1

int → *int* → *int*

()

```
let x1:int = 2 + 3;;
```

```
let y:string = string_of_int x;;
```

Example 2

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

```
let y = string_of_int x;;
```

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

let inc = fun y → x + y

$T_{in} = T_y = \text{int}$

$T_{body} = T_{x+y} = \text{int}$

$T_{in} \rightarrow T_{body}$

$\text{int} \rightarrow \text{int}$

Example 3

$$T_{\text{foo}} = T_x \longrightarrow T_{\text{body}} \equiv \text{int} * \text{int} \rightarrow \text{int}$$

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

$$T_x = T_y * T_z = \text{int} * \text{int}$$

$$T_y = \text{int}$$

$$T_z = \text{int}$$

$$T_{\text{body}} = \text{int}$$

Example 4

$T_{\text{cat}} \equiv T_e \rightarrow T_{\text{body}} \equiv \text{string list} \rightarrow \text{string}$

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

$T_e \equiv \text{string list}$
 $T_{\text{body}} = \text{string}$
 $T_h = \text{string}$
 $T_t = \text{string list}$

XML doesn't know what the function does, or even that it finishes only its type!

Example 4

ML doesn't know what the function does, or even that it finishes only its type!
show list $\rightarrow$ string

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

string list $\rightarrow$ string

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

Example 5

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

$$T_{\text{map}} \equiv T_f \rightarrow T_{xs} \rightarrow T_{\text{body}} \equiv (X \rightarrow Y) \rightarrow X \text{ list} \rightarrow Y \text{ list}$$

```
let rec map f xs =  
  match xs with  
    [] -> []  
  | x :: xs' -> (f x) :: (map f xs')
```

$$T_{xs} = X \text{ list}$$

$$T_f \equiv \cancel{X} \rightarrow \cancel{Y}$$

$$T_{\text{body}} = Y \text{ list}$$

$T_x = X$ Introduce unknown tyvar: Unify, solve,
 $T_{xs'} = X \text{ list}$ Remaining tyvar gets a "forall"

Example 6

$$\exists (T_f^{in} \rightarrow T_f^{out}) \rightarrow (T_x \rightarrow T_f^{in}) \rightarrow T_x \rightarrow T_f^{out}$$

$$T_{comp} \equiv T_f \rightarrow T_g \rightarrow T_x \xrightarrow{(b \rightarrow c)} T_{body} \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

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

$$T_f \equiv T_f^{in} \rightarrow T_f^{out}$$

$$T_{body} = T_f^{out}$$

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

$$T_g \equiv T_g^{in} \rightarrow T_g^{out} \equiv T_x \rightarrow T_f^{in}$$

$$T_g^{in} = T_x^{in}$$

$$T_g^{out} = T_f^{in}$$

$$T_{\text{fold}} \equiv (T_{\text{cur}} \rightarrow X \rightarrow T_{\text{cur}}) \rightarrow T_{\text{cur}} \rightarrow X \text{ list} \rightarrow T_{\text{cur}}$$

Example 7 $\equiv$ $(('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b \text{ list} \rightarrow 'a)$

$$T_{\text{fold}} = T_f \rightarrow T_{\text{cur}} \rightarrow T_{xs} \longrightarrow T_{\text{cur}}$$

```
let rec fold f cur xs =
  match xs with
  | [] -> cur
  | x::xs' -> fold f (f cur x) xs'
```

$$T_{xs} = X \text{ list}$$

$$T_x = X$$

$$T_{xs'} = X \text{ list}$$

$$T_f \equiv T_{\text{cur}} \rightarrow X \rightarrow T_{\text{cur}}$$

(In Class Exercise A)

*split $\hat{=}$ $\forall^A$ A list $\rightarrow$ (~~B~~A list * A list)*

```
let rec split xs =  
  match xs with  
    | []    -> ([], [])  
    | [x]   -> ([x], [])  
    | y::(zxs') ->  
      let ys, zs = split xs' in  
      (y::ys, z::zs)
```

(In Class Exercise B)

```
let rec merge xs ys =  
  match (xs, ys) with  
    | ([], _) -> ys  
    | (_, []) -> xs  
    | (x::xs', y::ys') when x <= y  
      -> x :: (merge xs' ys)  
    | (x::xs', y::ys')  
      -> y :: (merge xs ys')
```

$\text{val split} :: \forall a. 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list}$
 $\text{val merge} :: \forall a. 'a \text{ list} \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$

(In Class Exercise C)

$T_{xs} \rightarrow T_{\text{Body}}$

$T_{xs} = X \text{ list}$

$T_{\text{Body}} = Y \text{ list}$

$T_{ys} = T_{zs} = X \text{ list}$

let rec msort $\textcircled{xs} =$

match xs **with**

| [] $\rightarrow$

$\textcircled{[]}$

$\forall X Y. X \text{ list} \rightarrow Y \text{ list}$

| x :: xs' $\rightarrow$

let ys, zs = split xs **in**

merge (msort ys) (msort zs)

$Y \text{ list}$

$Y \text{ list}$

$Y \text{ list}$

Example 11

```
let foo1 f g x =  
  if f x  
  then x  
  else g x
```

Example 12

```
let foo2 f g x =  
  if f x  
  then x  
  else foo2 f g (g x)
```