

CSE 130 : Spring 2011

Programming Languages

Lecture 8: *Type Inference*

Ranjit Jhala
UC San Diego



News

- PA4 is up (due 5/6)
 - Week **AFTER** midterm
- Midterm: Tue 5/3
 - In class, open book, etc.

Datatypes with many type variables

- Multiple type variables

```
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) : (string,int) tree

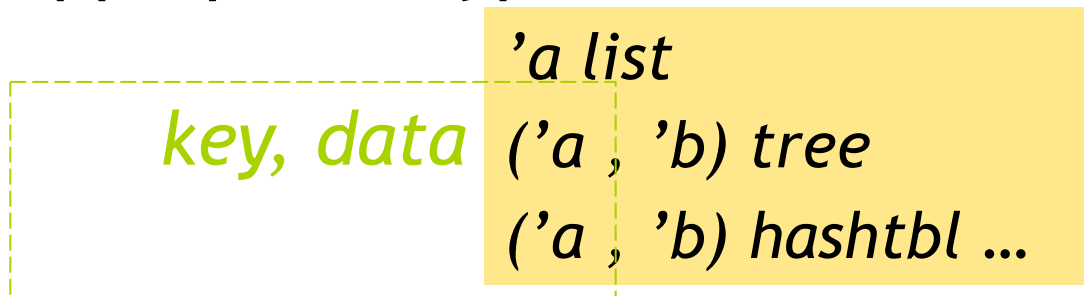
Leaf("william",2) : (string,int) tree

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**:

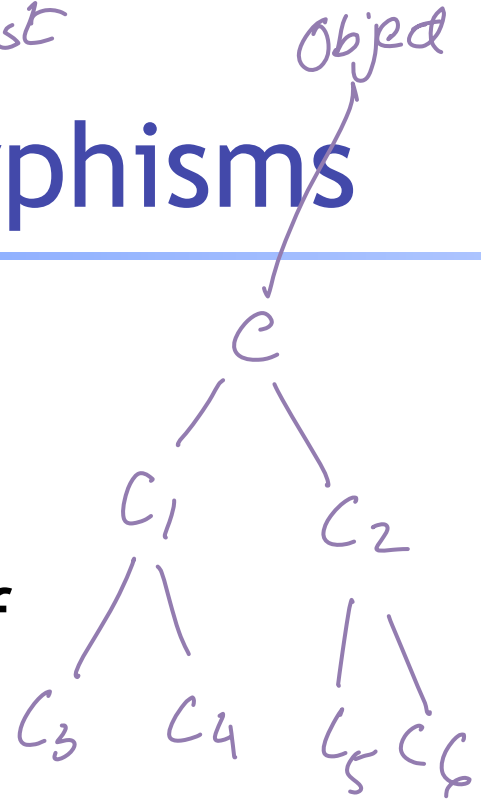


- Appropriate type instantiated at use
 - No **unsafe casting** as in C/C++/Java
- **Static type checking catches errors** early
 - Cannot add **int** key to **string** hashtable
- **Generics**: in Java, C#, VB (borrowed from ML)

List_rev : Object list $\rightarrow$ Object list
'a list $\rightarrow$ 'a list

Other kinds of polymorphisms

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



foo(int x, int y) { - - }

foo(float x, float y) { . . }

"overloading"

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!

list<T> reverse(list<T> xs) { ... }

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
- 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!

Another In-Class Exercise!

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

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

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

Example 2

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

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

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

Example 3

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

Example 4

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

*ML 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!

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

string list -> string

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

Example 5

($'a \rightarrow 'b$) $\rightarrow 'a$ list $\rightarrow 'b$ list

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

*Introduce unknown tyvar: Unify, solve,
Remaining tyvar gets a “forall”*

Example 6

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

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

Example 7

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

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

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

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