CSE 130: Programming Languages

Polymorphism

Ranjit Jhala
UC San Diego

Q: What is the value of `res` ?

```
let f g =
  let x = 0 in
  g 2
let x     = 100
let h y = x + y
let res = f h
```

(a) 0 (b) 2 (c) 100 (d) 102 (e) 12

Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment
- Static/Lexical: Determined from the program text
  - Without executing the program
- Very useful for readability, debugging:
  - Don't have to figure out "where" a variable got assigned
  - Unique, statically known definition for each occurrence

Immutability: The Colbert Principle

"A function behaves the same way on Wednesday, as it behaved on Monday, *no matter what happened on Tuesday!*"

Polymorphism

News

- Midterm on Friday
  - Double-sided “cheat sheet”
  - Printed, if you like
- PA4 due NEXT Friday @ 5p
  - **First half** relevant for Midterm

Polymorphism enables Reuse

- Can reuse generic functions:

```
map :'a * 'b -> 'b * 'a
filter: ('a -> bool) -> 'a list -> 'a list
rev: 'a list -> 'a list
length: 'a list -> int
swap: 'a * 'b -> 'b * 'a
sort: ('a -> 'a -> bool) -> 'a list -> 'a list
fold: ...
```

- If function (algorithm) is independent of type, can reuse code for all types !

Polymorphic Data Types

- Data types are also polymorphic!

```
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 :

Polymorphic Data Types

- Data types are also polymorphic!

```
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

Datatypes with many type variables

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

Q: What is the type of res ?

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

let res = Node ("alice", 5, Leaf, Leaf)
```

(a) `(int, string) tree`

(b) `('a,'b) tree`

(c) `int tree`

(d) type error

(e) `(string, int) tree`

Datatypes with many type variables

- Multiple type variables

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

- Type is instantiated for each use:

Node("alice", 2, Leaf, Leaf)

Node("charlie", 3, Leaf, Leaf)

Node("bob", 13,
, Node("alice", 2, Leaf, Leaf)
, Node("charlie", 3, Leaf, Leaf))

Q: What is the type of res ?

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

let res = Node("bob",13,Node(3, "alice",Leaf, Leaf)
                          ,Leaf)
```

(a) `(int, string) tree`
(b) `('a,'b) tree`
(c) `int tree`
(d) type error
(e) `(string, int) tree`

Datatypes with many type variables

- Multiple type variables

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

- Type is instantiated for each use:

Node("alice", 2, Leaf, Leaf)

Node("charlie", 3, Leaf, Leaf)

Node("bob", 13,
, Node("alice", 2, Leaf, Leaf)
, Node(3, "charlie", Leaf, Leaf))

A tricky question: consider this type

```
type ('a, 'b) wierdlist =
   Nil
 | Cons 'a* ('b, 'a) wierdlist
```

Which is a valid Ocaml Expression?

(a) `Cons(1, Cons("a", Cons(3.14, Nil)))`
(b) `Cons(1, Cons("a", Cons(1, Nil)))`
(c) `Cons(1, Cons("a", Cons("a", Nil)))`
(d) `Cons(1, Cons(1, Cons("a", Nil)))`
(e) `Cons(1, Cons(1, Cons(1, Nil)))`

Polymorphic Data Structures

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

'a list
('a , 'b) tree
('a , 'b) hashtbl ...

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

Type Inference

How 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
- Unlike values (determined at run-time)

Polymorphic Type Inference

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

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;;
```

Whats the type of `foo`?

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

(a) `int`

(b) `int * int`

(c) `int * int -> int`

(d) `int -> int -> int`

(e) `Error`

Example 4

```
let rec cat xs =
  match xs with
  | []     -> cat []
  | x::xs  -> x^(cat xs)
```

(a) `string -> string`

(b) `string`

(c) `string list -> string list`

(d) `string list -> string`

(e) `Error`

Example 5

```
let rec cat xs =
  match xs with
  | []      -> ""
  | x::xs -> x^(cat xs)
```

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

```
let rec cat xs =
  match xs with
  | []      -> ""
  | x::xs -> x^(cat xs)
```

```
let rec cat xs =
  match xs with
  | []      -> cat []
  | x::xs -> x^(cat xs)
```

Example 5

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

Example 5

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

"Generalize" Unconstrained Vars

('a->'b) -> 'a list -> 'b list

What is the type of (<+>)

```
let (<+>) f g x = g (f x)
```

(a) `'a -> 'b -> 'c -> 'd`

(b) `('a->'b)->('a ->'b)->('a ->'b)`

(c) `(int->char)->(char->bool)->(int->bool)`

(d) `(int->int)->(int->int)->(int->int)`

(e) `('a->'b)->('b ->'c)->('a ->'c)`

Example 6

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

$T_{comp} \equiv T_f \rightarrow T_g \rightarrow T_x \rightarrow T_{BODY}$

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

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

BODY

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

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

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

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

$T_g^{in} = T_x$

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

Example 7

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

Example 7

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

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

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

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

$T_{xs} = X\ list$

$T_x = X$

$T_{xs'} = X\ list$

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

(In Class Exercise A)

split : ∀A. A list → (A list * A list)

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

val split : ∀a 'a list → 'a list * 'a list
val merge :: ∀a 'a list → 'a list → 'a list

(In Class Exercise C)

$T_{xs} \to T_{Body}$

$T_{xs} = X$ list

$T_{Body} = Y$ list

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

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

∀XY. X list → Y list

Y list   Y list

Y list

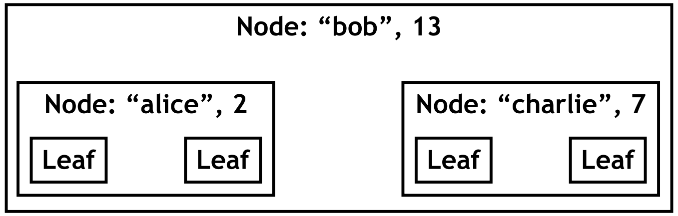
Example 11

```
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)
```

BST Property: *keys in left < key < keys in right*




```
Node("bob", 13
    , Node("alice", 2, Leaf, Leaf)
    , Node("charlie", 3, Leaf, Leaf))
```

Binary Search Trees

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

Node (key, value, left, right)

BST Property:

keys in left < key < keys in right

Exercise!

BST Property: *keys in left < key < keys in right*


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


Write a function to lookup keys…

```
val lookup:'a ->('a,'b)tree -> 'b option
```