Polymorphism

Q: What is the value of \( \text{res} \)?

\[
\begin{align*}
\text{let } f \ g = \\
&\quad \text{let } x = 0 \text{ in} \\
&\quad g \ 2 \\
\text{let } x = 100 \\
\text{let } h \ y = x + y \\
\text{let } \text{res} = f \ h
\end{align*}
\]

(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

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!
  ```ocaml
type 'a list =  
  | Nil 
  | Cons of ('a * 'a list)
  ```

- Type is instantiated for each use:
  ```ocaml
  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

- Multiple type variables
  ```ocaml
  type ('a, 'b) tree =  
  | Leaf  
  | Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree  
  ```

Q: What is the type of res?

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

- Type is instantiated for each use:
  ```ocaml
  let res = Node (“alice”, 5, Leaf, Leaf)
  ```

  (a) Node(“alice”, 2, Leaf, Leaf)
  (b) Node(“charlie”, 3, Leaf, Leaf)
  (c) Node(“bob”, 13,  
       , Node(“alice”, 2, Leaf, Leaf)   
       
       , Node(“charlie”, 3, Leaf, Leaf))
Q: What is the type of \texttt{res}?

(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 is instantiated for each use:
  ```ocaml
type ('a,'b) tree =
  Leaf
| Node of 'a* 'b * ('a,'b) tree * ('a,'b) tree
```

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

A tricky question: consider this type

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

• 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;;

What's 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

```ml
let rec cat xs =  
  match xs with  
  | []    -> ""  
  | x::xs -> x^(cat xs)
```
What is the type of \((<+>)\)

\[
\text{let } (<> \ f \ g \ x = g \ (f \ x) \n\]

(a) \('a \to 'b \to 'c \to 'd\)
(b) \('a \to 'b\)\(\to ('a \to 'b) \to ('a \to 'b)\)
(c) \(\text{int} \to \text{char} \to \text{char} \to \text{bool} \to \text{int} \to \text{bool}\)
(d) \(\text{int} \to \text{int} \to \text{int} \to \text{int}\)
(e) \('a \to 'b\)\(\to ('b \to 'c) \to ('a \to 'c)\)

Example 6

\[
\text{let compose } \ f \ g \ x = f \ (g \ x) \n\]

\[
\begin{align*}
T_f & \equiv T'_{in} \to T'_{out} \\
T_g & \equiv T_{out} \\
T_{comp} & \equiv T_f \to T_g \\
T_{body} & \equiv (b \to c) \to (a \to b) \to (a \to c) \\
T_{fold} & \equiv (\text{cur} \to \text{out}) \to \text{out} \to \text{out} \to \text{out} \\
T_{xs} & \equiv \text{list} \\
T_{fold} & \equiv T_f \to T_{cur} \to T_{xs} \to T_{cur}
\end{align*}
\]

Example 7

\[
\text{let rec } \text{fold } \ f \ \text{cur } \text{xs } = \\
\text{match } \text{xs } \text{with} \\
\quad [] \quad \to \text{cur} \\
\quad x::xs' \to \text{fold } \ f \ (f \ \text{cur } x) \ xs'
\]

\[
\begin{align*}
T_{xs} & = \text{list} \\
T_{x} & = \text{x} \\
T_{xs'} & = \text{list} \\
T_{fold} & \equiv T_{cur} \to X \to T_{cur} \\
T_{fold} & \equiv (\text{a} \to \text{b} \to \text{a}) \to \text{a} \to \text{b list} \\
T_{fold} & \equiv (\text{cur} \to \text{out}) \to \text{out} \to \text{out} \to \text{out} \\
T_{fold} & \equiv T_f \to T_{cur} \to T_{xs} \to T_{cur}
\end{align*}
\]
let rec split xs = 
  match xs with 
  | []  -> ([], []) 
  | [x] -> ([x], []) 
  | y::z::xs' -> 
    let ys,zs = split xs' in  
    (y::ys, z::zs)

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')

let rec msort xs = 
  match xs with 
  | []     -> [] 
  | x::xs' -> 
    let ys,zs = split xs in  
    merge (msort ys) (msort zs)
let foo2 f g x = 
  if f x  
  then x  
  else foo2 f g (g x)

**Example 12**

```plaintext
val foo2 : (a -> bool) -> (a -> bool) -> a -> a = 
  fun f g x ->
    if f x then x else foo2 f g (g x)
```

**Binary Search Trees**

**Node** (key, value, left, right)

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

**Exercise!**

Write a function to lookup keys...

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

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

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

```plaintext
Node ("bob", 13)
  , Node("alice", 2, Leaf, Leaf)
  , Node("charlie", 7, Leaf, Leaf)

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