

CSE 130: Fall 2012

Programming Languages

Lecture 9: *Finish Closures & Polymorphism*

Ranjit Jhala
UC San Diego



Free vs. Bound Variables

```
let a = 20;;

let f x =
  let a = 1 in
  let g z = a + z in
    a + (g x)
;;

f 0;
```

Inside a function:

A “bound” occurrence:

1. Formal variable
2. Variable bound in `let-in-end`

x, **a**, **z** are “bound” inside **f**

A “free” occurrence:

Not bound occurrence

nothing is “free” inside **f**

Where do bound-vars values come from?

```
let a = 20;;

let f x =
  let a = 1 in
  let g z = a + z in
    a + (g x)
;;

f 0;
```

Bound values determined when function is evaluated (“called”)

- Arguments
- Local variable bindings

Values of function application

Two questions about function values:

What is the *value*:

1. ... of a function ?

fun x -> e

2. ... of a function “application” (call) ? *(e1 e2)*

“apply” the argument *e2* to the (function) *e1*

Values of function application

Value of a function “application” (call) $(e1\ e2)$

1. Compute **closure** of $e1$
2. Execute **body** of **closure** with param $e2$

Free values found in **closure-environment**

Bound values by executing **closure-body**

Values of function application

Value of a function “application” (call) $(e1\ e2)$

1. Evaluate $e1$ in current-env to get (**closure**)
= **code** (**formal x** + **body e**) + **env E**
2. Evaluate $e2$ in current-env to get (argument) $v2$
3. Evaluate body e in env E extended with $x := v2$

Q: What is the value of **res** ?

```
let x = 1;;  
let y = 10;;  
let f y = x + y;;  
let x = 2;;  
let y = 3;;  
let res = f (x + y);;
```

```
f |-> formal := y  
      body  := x + y  
      env   := [x |-> 1]  
  
x |-> 2  
y |-> 3  
  
x + y ==> 5
```

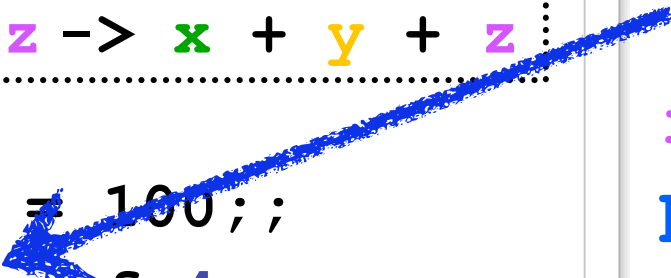
Application: $f \ (x + y)$

Eval **body** in **env** extended with **formal** |-> 5

Eval **x+y** in [**x** |-> 1, **y** |-> 5] ==> 6

Example

```
let x = 1;;  
let f y =  
  let x = 2 in  
  fun z -> x + y + z  
;;  
let x = 100;;  
let g = f 4;;  
let y = 100;;  
(g 1) ;;
```



Q: Closure value of g?

```
formal z  
body    x + y + z  
env     [x|->2, y|->4]
```

Eval **body** in **env** extended with **formal|-> 1**

Eval **x+y+z** in **[x|->2, y|->4, z|->1]** **====> 7**

Example 3

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

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 NEXT Week
 - Review Session on **Monday**
- PA4 Due NEXT Friday @ 5p
 - **Second half** relevant for Midterm
- Extra Office Hours
 - Details on Piazza

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

Options

- Consider the type:

```
type 'a option = None | Some of 'a
```

- Why is this useful?

Remember This Guy?

```
val assoc: 'b -> 'a -> ('a*'b) list -> 'b
```

We had to pass in “default” value (yuck!)

Instead, use “option” to return failure...

```
val assoc: 'a -> ('a*'b) list -> 'b option
```

Example: Calculator Revisited

```
type expr =  
  | Num of int  
  | Div of expr * expr
```

Can you write a function?

```
val eval : expr -> int
```

In Class Exercise

```
type expr =  
  | Num of int  
  | Div of expr * expr
```

Write an Evaluation function

```
val eval : expr -> int option
```

That returns *None* if a div-by-zero occurs

Datatypes with many type variables

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

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))*

Binary Search Trees

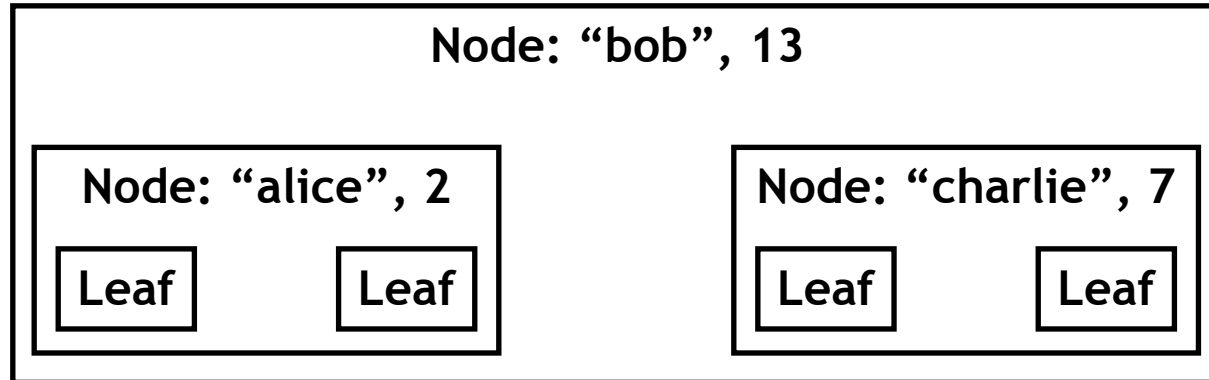
```
type ('a, 'b) tree =  
  Leaf  
| Node of 'a * 'b * ('a, 'b) tree * ('a, 'b) tree
```

Node (key, value, left, right)

BST Property:

keys in left < key < keys in right

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



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

In-Class 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
```

Polymorphic Data Structures

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

```
'a list  
( 'a , 'b ) tree  
( 'a , 'b ) hashtable ...
```

- **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;;
```

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

,

Example 7

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

Example 11

```
let fool 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)
```