

CSE 130 : Fall 2011

Programming Languages

Lecture 5: Environments and Closures

Ranjit Jhala
UC San Diego



Environments

“Phone book”

- Variables = “names”
- Values = “phone number”

1. Evaluate:

Find and use most recent value of variable

2. Extend:

Add new binding at end of “phone book”

Example

```
# let x = 2+2;;
val x : int = 4

# let y = x * x * x;;
val y : int = 64

# let z = [x;y;x+y];;
val z : int list = [4;64;68]

# let x = x + x ;;
val x : int = 8
```

New binding!

M	M
---	---

M	M
x	4 : int

M	M
x	4 : int
y	64 : int

M	M
x	4 : int
y	64 : int
z	[4;64;68] : int list

M	M
x	4 : int
y	64 : int
z	[4;64;68] : int list
x	8 : int

Environments

1. Evaluate: Use most recent bound value of var
2. Extend: Add new binding at end

How is this different from C/Java’s “store” ?

```
# let x = 2+2;;
val x : int = 4

# let f = fun y -> x + y;
val f : int -> int = fn

# let x = x + x ;
val x : int = 8

# f 0;
val it : int = 4
```

M	M
x	4 : int

M	M
x	4 : int
f	fn <code, ↑> : int->int

New binding:

- No change or mutation
- Old binding frozen in f

Environments

1. **Evaluate:** Use **most recent** bound value of var
2. **Extend:** Add **new** binding at end

How is this different from C/Java's "store" ?

```
# let x = 2+2;
val : int x = 4

# let f = fun y -> x + y;
val f : int -> int = fn

# let x = x + x ;
val x : int = 8;

# f 0;
val it : int = 4
```

M	M
x	4 : int

M	M
x	4 : int
f	fn <code, ↑> : int->int

M	M
x	4 : int
f	fn <code, ↑> : int->int
x	8 : int

Environments

1. **Evaluate:** Use **most recent** bound value of var
2. **Extend:** Add **new** binding at end

How is this different from C/Java's "store" ?

```
# let x = 2+2;
val x : int = 4

# let f = fun y -> x + y;;
val f : int -> int = fn

# let x = x + x ;
val x : int = 8

# f 0;
val it : int = 4
```

Binding used to eval (f ...)

M	M
x	4 : int
f	fn <code, ↑> : int->int
x	8 : int

Binding for subsequent x

Cannot change the world

Cannot "assign" to variables

- Can extend the env by adding a fresh binding
- Does **not** affect previous uses of variable

Environment at fun declaration **frozen** inside fun "value"

- Frozen env used to evaluate application (f ...)

Q: Why is this a good thing ?

```
# let x = 2+2;;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + x ;;
val x : int = 8;
# f 0;;
val it : int = 4
```

Binding used to eval (f ...)

M	M
x	4 : int
f	fn <code, ↑> : int->int
x	8 : int

Binding for subsequent x

Cannot change the world

Q: Why is this a good thing ?

A: Function **behavior** frozen at **declaration**

- Nothing entered afterwards affects function
- Same inputs **always produce same outputs**
 - Localizes debugging
 - Localizes reasoning about the program
 - No "sharing" means no evil aliasing

Examples of no sharing

Remember: No addresses, no sharing.

- Each variable is bound to a “fresh instance” of a value

Tuples, Lists ...

- Efficient implementation without sharing ?
- There is sharing and pointers but hidden from you
- Compiler’s job is to optimize code
- Efficiently implement these “no-sharing” semantics
- Your job is to use the simplified semantics
- Write correct, cleaner, readable, extendable systems

Function bindings

Functions are values, can bind using **val**

```
let fname = fun x -> e ;;
```

Problem: Can’t define recursive functions !

- fname is bound **after** computing rhs value
- no (or “old”) binding for occurrences of fname inside e

```
let rec fname x = e ;;
```

Occurrences of fname inside e bound to “this” definition

```
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```

Local bindings

So far: bindings that remain until a re-binding (“global”)

Local, “temporary” variables are useful inside functions

- Avoid repeating computations
- Make functions more readable

Let-in is an expression!

```
let x = e1 in
  e2
;;
```

Evaluating let-in in env *E*:

1. Evaluate expr *e1* in env *E* to get value *v* : *t*
2. Use extended *E* [*x* := *v* : *t*] (only) to evaluate *e2*

Local bindings

Evaluating let-in in env *E*:

1. Evaluate expr *e1* in env *E* to get value *v* : *t*
2. Use extended *E* [*x* := *v* : *t*] to evaluate *e2*

```
let
  x = 10
in
  x * x
;;
```

M	M
---	---

M	M
x	10 : int

M	M
---	---

Let-in is an expression!

Evaluating let-in in env E :

1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended $E [x := v : t]$ to evaluate e_2

```
let y =
  let
    x = 10
  in
    x * x
;;
```

M	M
---	---

M	M
x	10 : int

M	M
y	100 : int

Nested bindings

Evaluating let-in in env E :

1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended $E [x := v : t]$ to evaluate e_2

```
let
  x = 10
in
  (let
    y = 20
  in
    x * y)
+ x
;;
```

M	M
---	---

M	M
x	10 : int

M	M
x	10 : int
y	20 : int

M	M
x	10 : int

M	M
---	---

Nested bindings

```
let
  x = 10
in
  let
    y = 20
  in
    x * y
;;
```

```
let x = 10 in
let y = 20 in
  x * y
;;
```

Correct Formatting

Example

```
let rec filter (f,l) =
  if l = [] then []
  else
    let h = hd l in
    let t = filter (f, tl l) in
    if (f h) then h::t else t
```

Nested function bindings

```
let a = 20;;
```

```
let f x =  
  let y = 10 in  
  let g z = y + z in  
    a + (g x)  
  ;;
```

```
f 0;
```

Env **frozen** with function

- Used to evaluate fun application
- Values in application are those frozen in env at definition

Recap

- Variables are **names** for **values**
 - Environment: dictionary/phonebook
 - Most recent binding used
 - Entries never changed, new entries added
- Environment **frozen at fun definition**
 - Re-binding variables cannot change a function
 - Same I/O behavior at every call

Tuesday, October 11, 2011

Tuesday, October 11, 2011

Recap

- Build complex expressions with **local** bindings
 - **let-in** expression
 - The **let**-binding is visible (in scope) inside **in**-expression
 - Elsewhere the binding is not visible

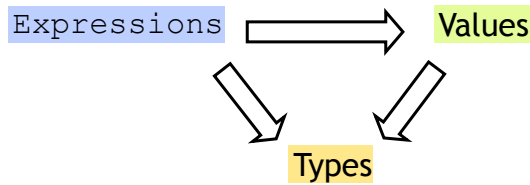
Tuesday, October 11, 2011

Static/Lexical Scoping

- For each occurrence of a variable, there is a **unique** place in program text where the variable was 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

Tuesday, October 11, 2011

Next: Functions



Q: What's the **value** of a **function** ?

Recap: Environments

“Phone book”

- Variables = “names”
- Values = “phone number”

1. **Evaluate:**

Find and use most recent value of variable

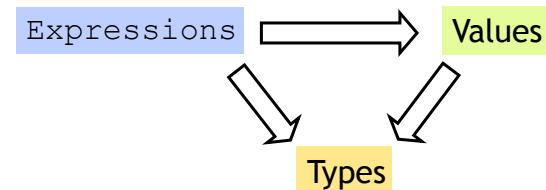
2. **Extend:** `let x = e ; ;`

Add new binding at end of “phone book”

Recap

- Variables are **names** for **values**
 - Environment: dictionary/phonebook
 - Most recent binding used
 - Entries never changed, new entries added
- Environment **frozen at fun definition**
 - Re-binding variables cannot change a function
 - Same I/O behavior at every call

Next: Functions *are "first-class"*



Q: What's the **value** of a **function** ?

Functions

Expressions

Two ways of writing function expressions:

1. Anonymous functions:

Parameter (formal) Body Expr
`let fname = fun x -> e`

2. Named functions:

Parameter (formal) Body Expr
`let fname x = e`

Function Application

Expressions

Application: fancy word for “call”

`(e1 e2)`

- Function value *e1*
- Argument *e2*
- “apply” argument *e2* to function value *e1*

Functions

Type

The type of any function is:

- *T1* : the type of the “input”
- *T2* : the type of the “output”

T1 -> *T2*

`let fname = fun x -> e`
T1 -> *T2*

`let fname x = e`
T1 -> *T2*

Functions

Type

The type of any function is:

- *T1* : the type of the “input”
- *T2* : the type of the “output”

T1 -> *T2*

T1, *T2* can be *any* types, including functions!

Whats an example of ?

- *int* -> *int*
- *int* * *int* -> *bool*
- (*int* -> *int*) -> (*int* -> *int*)

Type of function application

Application: fancy word for “call”

(*e1 e2*)

- “apply” argument *e2* to function value *e1*

$$\frac{e1 : T1 \rightarrow T2 \quad e2 : T1}{(e1 \ e2) : T2}$$

- **Argument** must have same type as “input” *T1*
- **Result** has the same type as “output” *T2*

Functions

Values

Two questions about function values:

What is the **value**:

1. ... of a function ?

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

(*e1 e2*)

Tuesday, October 11, 2011

Tuesday, October 11, 2011

Values of functions: Closures

- Function value = “**Closure**”
 - <code + environment at definition>
- **Body not evaluated** until application
 - but type-checking takes place when function is defined

```
# let x = 2+2;;  
val x : int = 4  
# let f = fun y -> x + y;;  
val f : int -> int = fn  
# let x = x + x ;;  
val x : int = 8  
# f 0;;  
val it : int = 4
```

Binding used to eval (*f ...*)

x	4 : int
f	fn <code, 
x	8 : int

Binding for subsequent *x*

Functions

Values

Two questions about function values:

What is the **value**:

1. ... of a function ?

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

(*e1 e2*)

Tuesday, October 11, 2011

Tuesday, October 11, 2011

Free (vs. Bound) Variables

```
let a = 20;;
```

```
let f x =  
  let y = 1 in  
  let g z = y + z in  
    a + (g x)  
;;  
f 0;;
```

Inside a function:

A “bound” occurrence:

1. Formal variable
 2. Variable bound in `let-in`
- `x`, `y`, `z` are “bound” inside `f`

A “free” occurrence:

- Not bound occurrence
- `a` is “free” inside `f`

Environment frozen in closure
used for values of free vars

Nested function bindings

```
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`

Nested function bindings

```
let a = 20;;
```

```
let f x =  
  let a = 1 in  
  let g z = a + z in  
    a + (g x)  
;;  
f 0;
```

Q: Where do **values** of **bound vars** come from?

Bound variable values determined when fun evaluated (“executed”)

- Arguments
- Local variable bindings

Values of function application

Application: fancy word for “call”

```
(e1 e2)
```

- “apply” the argument `e2` to the (function) `e1`

Application Value:

1. Evaluate `e1` in **current** env to get (function) `v1`
 - `v1` is **code + env**
 - code is (formal `x` + body `e`) , env is `E`
2. Evaluate `e2` in **current** env to get (argument) `v2`
3. Evaluate body `e` in env `E` **extended** by binding `x` to `v2`

Example 1

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

Example 2

```
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 (Code + Env) ?

Code:

- Formal: **z**
- Body expr: **x** + **y** + **z**

Frozen Environment:

- **x** bound to 2
- **y** bound to 4

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