Environments & Closures
Announcements

• HW#1 Grades
  • Point breakdown by problem (by email)
  • All course grades (on Gradesource by secret number)

• Email TAs if you haven’t received these
Results: Poll on Clickers

~45 votes

5 (11% of users) - Too Many Questions (Frequency)
29 (63% of users) - Right Number of Questions (Frequency)
10 (22% of users) - Too Few Questions (Frequency)
10 (22% of users) - Questions are Too Hard (Difficulty)
25 (54% of users) - Questions are Just Right (Difficulty)
6 (13% of users) - Questions are Too Easy (Difficulty)
Results: KQS Survey

~13 responses

KEEP Doing

• Pace of lectures
• Fair number of clicker questions
  • Enjoying clickers / discussion / talking with peers

QUIT Doing

• Group Seating
  • Sorry 😊 (But we can shuffle groups for 2nd half...)
• Explaining each wrong answer so slowly
• Waiting for last few votes
START Doing

• Timer countdown for clicker questions
  • Let’s try 20 seconds for vote after group discussion

• Posting slides before class
  • (I’ve been posting them ~1 to 3 hours before class)

• More full examples at interpreter

• Keep discussion section material same

• Highlighting answers to clicker questions
  • I’m still reluctant, because some are open-ended, buggy, and most can be evaluated in REPL

Results: KQS Survey

~13 responses
Higher-Order Functions are awesome...

... but how do they work?

Let’s start with the humble variable...
Q: What is the value of res?

(a) (0, 1)
(b) (100, 101)
(c) (0, 100)
(d) (1, 100)

```haskell
let x = 0      ;;
let y = x + 1  ;;
let z = (x, y) ;;
let x = 100    ;;
let res = z    ;;
```
let x = e;;

“Bind value of expr e to variable x”

# 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]

• Later on, most recently “bound” value used to evaluate x
• Sounds like C/Java? NO!
Environments (“Phone Book”)

How ML deals with variables:
• Variables = “names”
• Values = “phone number”

<table>
<thead>
<tr>
<th>x1</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x2</td>
<td>v2</td>
</tr>
<tr>
<td>x3</td>
<td>v3</td>
</tr>
<tr>
<td>x4</td>
<td>v4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
ML begins in a “top-level” environment

- Phone book has entries for some names (e.g. +, −, print_string)

ML program = sequence of let-bindings

Repeat for each binding, in order:
1. Evaluate expr e in current env to get value v : t
2. Extend env to bind x to v : t
Environments and Evaluation

```
# 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!
“Shadows” previous binding
Environments and Evaluation

“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”
   - **Old binding**, if any, persists but is “shadowed”
How is it different from C/Java’s “store”?

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

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

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

# f 0 ;;
- : int = 4
```

New binding:
- No change or mutation
- **Old binding** frozen in `f`
Cannot Change the World

Can extend env with fresh binding
- **Does not affect** previous uses of variable
- That is, cannot “assign” to variables

Env at fun decl **frozen** inside `<fun>` value
- Frozen env used to evaluate application \( f \ e \)

Q: Why is this a good thing?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>4 : int</td>
</tr>
<tr>
<td>( f )</td>
<td><code>&lt;fun&gt; : int -&gt; int</code></td>
</tr>
<tr>
<td>( x )</td>
<td>8 : int</td>
</tr>
</tbody>
</table>
Cannot Change the World

Can extend env with fresh binding
- Does not affect previous uses of variable
- That is, cannot “assign” to variables

Env at fun decl \texttt{frozen} inside \texttt{<fun>} value
- Frozen env used to evaluate application \texttt{f e}

Q: Why is this a good thing?
A: Function behavior frozen at declaration!
Immutability: The Colbert Principle

“A function behaves the same way on Wednesday as it behaved on Monday, no matter what happened on Tuesday!”
Q: Why is this a good thing?
A: Function behavior frozen at declaration

- **Nothing** evaluated later affects function
- **Same inputs** always produce **same outputs**
  - Localizes **debugging**
  - Localizes **reasoning** about the program
  - No “sharing” means no evil aliasing
No Sharing

No addresses/pointers $\Rightarrow$ No sharing/aliasing

- Each variable is bound to a different value

So tuples, lists, etc. are extremely inefficient?

No! Compiler’s job is to optimize code

- Efficiently implement the “no-sharing” semantics
- There is sharing and pointers, but hidden from programmer

Programmer’s job is to write...

- ... correct, clean, readable, extendable systems
- Made easier by simplified semantics
Q: What is the value of res?

```
let f x = 1;;
let f x = if x<2 then 1 else (x * f(x-1));;
let res = f 5;;
```

(a) 120
(b) 60
(c) 20
(d) 5
(e) 1
Function Bindings

Functions are values, can bind using `let`

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

**Problem:** Can’t define recursive functions!
- `fname` is bound after computing value on right-hand side
- No “old” binding for occurrences of `fname` inside `e`

---

```plaintext
let rec fname x = e
```

Occurrences of `fname` inside `e` bound to “this” definition

```plaintext
let rec fac x =
  if x<=1 then 1 else x * fac (x-1)
```
So far: bindings remain “global” until a re-binding

Local, “temporary” variables are useful inside functions
- Avoid repeating computations
- Make functions more readable

```plaintext
let x = e1 in e2
```
Q: What is the value of res?

```ocaml
let y = let x = 10 in
       x + x ;;
let res = (x, y) ;;
```

(a) Syntax Error
(b) (10, 20)
(c) (10, 10)
(d) Type Error
Local Bindings

Evaluating let-in in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$
2. Use extended $E' = E [x \mapsto v_1 : t_1]$ to evaluate $e_2$
Local Bindings

Evaluating `let-in` in env $E$:
1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$
2. Use extended $E' = E \ [x \ mapsto v_1 : t_1]$ to evaluate $e_2$

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

$E$

$E'$
Local Bindings

Evaluating \texttt{let-in} in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$

2. Use extended $E' = E [x \mid \rightarrow v_1 : t_1]$ to evaluate $e_2$

\begin{verbatim}
let x = 10 in x * x ;;
\end{verbatim}
Local Bindings

Evaluating let-in in env E:
1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$
2. Use extended $E' = E [x \mapsto v_1 : t_1]$ to evaluate $e_2$

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

$\begin{array}{c}
\vdots \\
E \\
\vdots \\
E'
\end{array}
\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array}
\begin{array}{c}
\vdots \\
x \\
10 : \text{int}
\end{array}$
Local Bindings

Evaluating `let-in` in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$

2. Use extended $E' = E[x \mapsto v_1 : t_1]$ to evaluate $e_2$
Nested Bindings

Evaluating `let-in` in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$
2. Use extended $E' = E [x \mapsto v_1 : t_1]$ to evaluate $e_2$
Evaluating `let-in` in env \( E \):

1. Evaluate expr \( e_1 \) in env \( E \) to get value \( v_1 : t_1 \)
2. Use extended \( E' = E \[ x \mid \rightarrow v_1 : t_1 \] \) to evaluate \( e_2 \)

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

Evaluating `let-in` in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v_1 : t_1$
2. Use extended $E' = E [x \mapsto v_1 : t_1]$ to evaluate $e_2$
Nested Bindings

BAD Formatting

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

BAD Formatting

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

GOOD Formatting

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

(\textit{except for HW 2 \texttrademark{)}}}
Example

let rec filter f xs =
  match xs with
  | []         -> []
  | x::xs'     -> let ys  = if f x then [x] else [] in
                 let ys' = filter f xs
                 in
                 ys @ ys'

Recap: Environments

Variables are names for values

- Environment is dictionary/phonebook
- Most recent binding used
- Entries never change
- New entries added
Recap: Environments

Variables are names for values

Big expressions with local bindings

- `let-in` expression
- Variable “in scope” in `in`-expression
- Outside, variable not “out of scope”
Recap: Environments

Variables are names for values

Big expressions with local bindings

Env frozen at function definition

- Re-binding vars cannot change function
- Identical I/O behavior at every call
- Predictable code, localizes debugging
Recap: Environments

How is environment “frozen into” `<fun>`?
Q: What’s the value of a function?
Two ways of writing function expressions:

1. Anonymous functions:

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

2. Named functions:

   ```
   let fname x = e
   ```
Functions

Expressions

\( x \) is “in scope” in \( e \)

\[
\text{fun } x \rightarrow e == \text{fun } x \rightarrow (e)
\]

\[
\text{fun } x \rightarrow \text{fun } y \rightarrow e == \text{fun } x \rightarrow (\text{fun } y \rightarrow (e))
\]
Function Application (or “Call”)

“Apply” function value $e_1$ to argument $e_2$

$e_1 \ e_2$

Calls associate to the left:

$e_1 \ e_2 \ e_3 \ === \ ((e_1 \ e_2) \ e_3)$

$e_1 \ (e_2 \ e_3) \ === \ (e_1 \ (e_2 \ e_3))$
Every function has a type of the form:

- \( T1 \): the type of the “input”
- \( T2 \): the type of the “output”
Functions

Types

Every function has a type of the form:
• \( T_1 \) : the type of the “input”
• \( T_2 \) : the type of the “output”

\( T_1 \rightarrow T_2 \)

\( T_1, T_2 \) can be any types, including function types!

Whats an example of?
• \( \text{int} \rightarrow \text{int} \)
• \( \text{int} \ast \text{int} \rightarrow \text{bool} \)
• \( (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \)
Every function has a type of the form:
- $T_1$: the type of the “input”
- $T_2$: the type of the “output”

Function types associate to the right:

\[
T_1 \to T_2 \to T_3 \equiv (T_1 \to (T_2 \to T_3))
\]

\[
(T_1 \to T_2) \to T_3 \equiv ((T_1 \to T_2) \to T_3)
\]

\[
(T_1 \to T_2) \to T_3 \to T_4 \equiv ((T_1 \to T_2) \to (T_3 \to T_4))
\]
Think of function types as trees

\[ T_1 \rightarrow T_2 \rightarrow T_3 \]

\[ (T_1 \rightarrow T_2) \rightarrow T_3 \rightarrow T_4 \]
(e₁ e₂)

“Apply” function $e₁$ to argument $e₂$
or “Call” function $e₁$ with argument $e₂$
or “Pass” argument $e₂$ to function $e₁$

$e₁ : T₁ \rightarrow T₂$
$e₂ : T₁$
$(e₁ e₂) : T₂$

Argument must have same type as “input” $T₁$
Result has the same type as “output” $T₂$
Two questions:

What is the value:

1. ... of a function ?

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

What is the value:

1. ... of a function?

Closure = Code of Fun. (\texttt{formal x + body e}) + Environment at Fun. Definition
Values of Functions = “Closures”

- “Closure” = <code + environment at definition>
- Type checking when function is defined
- Body not evaluated until application

```ocaml
# 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 _ : int = 4
```

<table>
<thead>
<tr>
<th>Binding used to evaluate f</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

"closes over" all previous bindings

Binding for subsequent uses of x
Q: Which vars in closure of f?

```ml
let x = 2 + 2 ;;
let f y = x + y ;;
let z = x + 1 ;;
```

(a) x  
(b) y  
(c) z  
(d) x y z  
(e) None
Q: Which vars in closure of \( f \) ?

\[
\begin{align*}
\text{let } &a = 20; ; \\
\text{let } &f \ x = \\
&\quad \text{let } y = 1 \text{ in} \\
&\quad \text{let } g \ z = y + z \text{ in} \\
&\quad \quad a + (g \ x) \\
\end{align*}
\]

(a) a y  
(b) a  
(c) y  
(d) z  
(e) y z
Functions

Two questions:

What is the value:

1. ... of a function?

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

fun x -> e

(e1 e2)
Free vs. Bound Variables

Environment *frozen* inside function definition

Used to evaluate function application \((e_1 \ e_2)\)

Which vars are needed from frozen env?

```ml
let a = 20;;

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

```ml
f 0;;
```
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, g are “bound” inside f

A “free” occurrence:
• An unbound variable
a is “free” inside f

Frozen Environment
needed for values of free vars
let a = 20;;

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

f 0;;

**Bound** values determined when function is evaluated ("called")
- Arguments
- Local variable bindings
Values of Function Application

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

1. Evaluate \(e1\) in current-env to a closure

   \[= \text{code (formal } x + \text{ body } e) + \text{ closure-env } E\]

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 \( \text{res} \)?

\[
\text{let } f \ g = \\
\quad \text{let } x = 0 \text{ in } \\
\quad g \ 2 \\
; \\
\text{let } x = 100; ; \\
\text{let } h \ y = x + y ; ; \\
\text{let } \text{res} = f \ h ; ;
\]

(a) Syntax Error
(b) 102
(c) Type Error
(d) 2
(e) 100
Example

```ocaml
let f g =
  let x = 0 in
  g 2
;;

let x = 100;;

let h y = x + y;;

f h;;
```
“A function behaves the same way on Wednesday as it behaved on Monday, no matter what happened on Tuesday!”
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