CSE 130 [Winter 2014]
Programming Languages

Higher-Order Functions
(Continued)

Ravi Chugh
Jan 28
Announcements

• KQS (Keep/Quit/Start) Survey
  • Drop off responses on table now or end of class

• Clicker Survey on Piazza

• HW #3 due Fri Feb 07

• HW #4 will be due Fri Feb 21
  • After midterm, but may include midterm material
Moral of the (Other) Day

HOFs Allow "Factoring" Into:

General Pattern
+
Specific Operation
Factor Into Generic + Specific

Specific Operations

```ocaml
let evens xs =
  filter (fun x -> x mod 2 = 0) xs

let bigrams xs =
  filter (fun x -> length x = 2) xs
```

Generic “filter” pattern

```ocaml
let rec filter f xs =
  match xs with
  | [] -> []
  | x::xs' -> if f x
    then x::(filter f xs')
    else (filter f xs')
```
Factor Into Generic + Specific

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

```
let listSquare = map (fun x -> x * x)

let listUpper = map uppercase
```

Specific Op

Generic “iteration” pattern
Generic “fold” Pattern

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

let len =
  foldr (fun x n -> n+1) 0

let sum =
  foldr (fun x n -> x+n) 0

let concat =
  foldr (fun x n -> x^n) ""
What does this evaluate to?

foldr (fun x n -> x::n) [] [1;2;3]

(a) [1; 2; 3]
(b) [3; 2; 1]
(c) []
(d) [[3]; [2]; [1]]
(e) [[[1]; [2]; [3]]

Let rec foldr f b xs =
  match xs with
  | [] -> b
  | x::xs' -> f x (foldr f b xs')
let rec foldr f b xs =
    match xs with
    | []       -> b
    | x::xs'    -> f x (foldr f b xs')

foldr f b [x1;x2;x3]
====> f x1 (foldr f b [x2;x3])
====> f x1 (f x2 (foldr f b [x3]))
====> f x1 (f x2 (f x3 (foldr f b [])))
====> f x1 (f x2 (f x3 (b)))
Tail Recursion

A function is “tail recursive” if:

- all recursive calls are immediately followed by return
- that is, each recursive call is in “tail position”
- so cannot do anything between call and return

Why do we care?

- Compiler can transform recursion into a loop
- You write readable code
- Compiler optimizes into fast code!
“fold-right” pattern

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

foldr f b [x1;x2;x3]
====> f x1 (foldr f b [x2;x3])
  !===> f x1 (f x2 (foldr f b [x3]))
  !===> f x1 (f x2 (f x3 (foldr f b [])))
  !===> f x1 (f x2 (f x3 (b)))

Not Tail Recursive!
Write: Tail-recursive concat

```ocaml
let concat xs = ...
```

```
concat []
    ===> ""

concat ["carne"; "asada"; "torta"]
    ===> "carneasadatorta"
```
let concat xs =
  let rec helper res = function
  | [] -> res
  | x::xs' -> helper (res^x) xs'
  in helper "" xs

helper "" [""carne""; ""asada""; ""torta"]
====> helper ""carne"" [""asada""; ""torta"]
====> helper ""carneasada"" [""torta"]
====> helper ""carneasadatorta"" []
====> ""carneasadatorta""
Write: Tail-recursive concat

let concat xs =
  let rec helper res = function
  | []    -> res
  | x::xs' -> helper (res^x) xs'
  in helper "" xs

helper "" [""carne""; ""asada""; ""torta"" ]
  ===> helper ""carneasada"" [""torta""
  ===> helper ""carneasadatorta"" []
  ===> ""carneasadatorta""
Write: Tail-recursive sum

```haskell
let sum xs = ...
```

```haskell
sum []        ===> 0
sum [10;20;30] ===> 60
```
Write: Tail-recursive sum

```ocaml
let sum xs =
  let rec helper res = function
    | [] -> res
    | x::xs' -> helper (res+x) xs'
  in helper 0 xs

helper 0 [10; 100; 1000]
====> helper 10 [100; 1000]
====> helper 110 [1000]
====> helper 1110 []
====> 1110
```
“Accumulation” Pattern

```ocaml
let foldl f b xs = let rec helper res = function
| []   -> res
| x::xs' -> helper (f res x) xs'
in helper b xs

let sum xs = let rec helper res = function
| []   -> res
| x::xs'-> helper (res + x) xs'
in helper 0 xs

let sum = foldl (fun res x -> res + x) 0

let concat xs = let rec helper res = function
| []   -> res
| x::xs'-> helper (res ^ x) xs'
in helper "" xs

let concat = foldl (fun res x -> res ^ x) ""
```
```
let foldl f b xs =
    let rec helper res = function
        | [] -> res
        | x::xs' -> helper (f res x) xs'
    in helper b xs

let sum = foldl (fun res x -> res + x) 0
let sum = foldl (fun res x -> res ^ x) ""
```

“Accumulation” Pattern
What does this evaluate to?

\[
\text{foldl (fun res x -> x::res) [] [1;2;3]}
\]

(a) [1; 2; 3]
(b) [3; 2; 1]
(c) []
(d) [[3]; [2]; [1]]
(e) [[1]; [2]; [3]]
Another fun function: “pipe”

```ocaml
let pipe x f = f x

let (|>) x f = f x
```

Compute sum of squares of numbers in a list:

```ocaml
let sumOfSquares xs =
  xs |> map (fun x -> x * x)
  |> foldl (+) 0
```

Tail Rec ?
Higher-Order Functions

Identify common computation “patterns”

- **Filter** values in a set, list, tree ...
- **Iterate/Map** a function over a set, list, tree ...
- **Accumulate/Fold** some value over a collection

Pull out (factor) boilerplate code:

- **Computation Patterns**
- **Re-use** in many different situations
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?
  - Data structure (list, tree, etc.) library provides meta-functions (map, fold, filter, etc.) to traverse in a generic way
  - Data structure client uses meta-functions with application-specific details
  - “MapReduce” = “MapFold”
Environments & Closures
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)

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

“Bind value of expression \( e \) to variable \( x \)”

\[
\begin{align*}
\# & \text{ let } x = 2 + 2 ; ; \\
& \text{val } x : \text{int} = 4 \\
\# & \text{ let } y = x \times x \times x ; ; \\
& \text{val } y : \text{int} = 64 \\
\# & \text{ let } z = [x; y; x+y] ; ; \\
& \text{val } z : \text{int list} = [4;64;68]
\end{align*}
\]

• Later on, **most recently** “bound” value used to evaluate \( x \)

• Sounds like C/Java? **NO!**
How ML deals with variables:

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>$v_4$</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”? 

New binding:  
- No change or mutation  
- **Old binding** frozen in \( f \)
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><code>x</code></td>
<td><code>4 : int</code></td>
</tr>
<tr>
<td><code>f</code></td>
<td><code>&lt;fun&gt; : int -&gt; int</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>8 : int</code></td>
</tr>
</tbody>
</table>

**Bindings used to eval** `(f ...)`

**Bindings not used** to eval `(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?

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`

```ml
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`

```ml
let rec fname x = e
```

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

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

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 $E' = E [x \mid \rightarrow \ v_1 : t_1]$ to evaluate $e_2$
Local Bindings

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

1. Evaluate expr `e1` in env `E` to get value `v1 : t1`
2. Use extended `E’ = E [x |-> v1 : t1]` to evaluate `e2`

```
let x = 10 in x * x;;
```
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 $E' = E \left[ x \mapsto v_1 : t_1 \right]$ 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 $E' = E [x |\rightarrow v_1 : t_1]$ to evaluate $e_2$

let $y =$
let $x =$
10
in
$x * x$
;;
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 $e_2$

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

$E$:

$E'$:

$E[x:=100]$:
Evaluating `let-in` in env $E$:

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

\[
\text{let } x = 10 \text{ in } (\text{let } y = 20 \text{ in } x * y) + x ;;
\]
Nested Bindings

**GOOD Formatting**

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

**BAD Formatting**

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

**BAD Formatting** (except for HW 2 😊)

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

```plaintext
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?
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`
Functions

Expressions

\( x \) is “in scope” in \( e \)

\[
\text{fun } x \to e \equiv \text{fun } x \to (e)
\]

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

Expresssions

“Apply” function value \( e_1 \) to argument \( e_2 \)

\[ e_1 \ e_2 \]

Calls associate to the left:

\[ e_1 \ e_2 \ e_3 \equiv (e_1 \ e_2) \ e_3 \]
\[ e_1 \ (e_2 \ e_3) \equiv (e_1 \ (e_2 \ e_3)) \]
Functions

**Types**

Every function has a type of the form:

- \( T_1 \) : the type of the “input”
- \( T_2 \) : the type of the “output”

\[
\begin{align*}
T_1 &\rightarrow T_2 \\
T_1 &\rightarrow T_2 \\
T_1 &\rightarrow T_2 \\
\end{align*}
\]

\[
\text{let } \text{fname} = \text{fun } x \rightarrow e
\]

\[
\text{let } \text{fname} x = e
\]
Every function has a type of the form:

- $T_1$: the type of the “input”
- $T_2$: the type of the “output”

$T_1, T_2$ can be any types, including function types!

What's an example of?

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

- $T_1 \rightarrow T_2 \rightarrow T_3 \equiv (T_1 \rightarrow (T_2 \rightarrow T_3))$
- $(T_1 \rightarrow T_2) \rightarrow T_3 \equiv ((T_1 \rightarrow T_2) \rightarrow T_3)$
- $(T_1 \rightarrow T_2) \rightarrow T_3 \rightarrow T_4 \equiv ((T_1 \rightarrow T_2) \rightarrow (T_3 \rightarrow T_4))$
Think of function types as trees

\[ T_1 \to T_2 \to T_3 \]

\[ (T_1 \to T_2) \to T_3 \to T_4 \]

\[ \to \]

\[ T_1 \]

\[ \to \]

\[ T_2 \]

\[ \to \]

\[ T_3 \]

\[ \to \]

\[ T_1 \]

\[ \to \]

\[ T_2 \]

\[ \to \]

\[ T_3 \]

\[ \to \]

\[ T_1 \]

\[ \to \]

\[ T_2 \]

\[ \to \]

\[ T_3 \]

\[ \to \]

\[ T_4 \]
Type of Function Application

\[(e_1 \ e_2)\]

“Apply” function \(e_1\) to argument \(e_2\)

or  “Call” function \(e_1\) with argument \(e_2\)

or  “Pass” argument \(e_2\) to function \(e_1\)

\[
e_1 : T_1 \rightarrow T_2 \quad e_2 : T_1 \\
(e_1 \ e_2) : T_2
\]

Argument must have same type as “input” \(T_1\)

Result has the same type as “output” \(T_2\)
Functions

Values

Two questions:

What is the \textbf{value}:

1. ... of a function \? \hspace{2cm} \textcolor{red}{\textbf{fun}} \hspace{0.2cm} x \hspace{0.2cm} \rightarrow \hspace{0.2cm} e

2. ... of a function "application" (call) \? \hspace{2cm} (e1 \hspace{0.2cm} e2)
Two questions:

What is the value:

1. ... of a function?

Closure =

Code of Fun. \((\text{formal } x + \text{ body } e)\)
+ Environment at Fun. Definition
• “Closure” = `<code + environment at definition>`

• Type checking when function is defined

• **Body not evaluated** until application

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

**Values of Functions = “Closures”**

- **Binding used to evaluate f**
  - `<fun>` “closes over” all previous bindings

- **Binding for subsequent uses of x**
Q: Which vars in closure of \( f \)?

```plaintext
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 \( f \)?

```plaintext
let \( a = 20 \);;

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

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

Values

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

```
let a = 20;;

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

Environment *frozen* inside function definition

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

```
f 0;;
```

Which vars are needed from frozen env?
Free vs. Bound Variables

```ocaml
let a = 20;;
let f x =
  let y = 1 in
  let g z = y + z in
  a + (g x);
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:
- An unbound variable
- `a` is “free” inside `f`

Frozen Environment
needed for values of free vars
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;;
```

Bound values determined when function is evaluated ("called")

- Arguments
- Local variable bindings
Value of a function “application” (call) \((e_1 \; e_2)\)

1. Evaluate \(e_1\) in current-env to a closure
   \[= \text{code (formal } x + \text{ body } e) + \text{ closure-env } E\]

2. Evaluate \(e_2\) in current-env to get (argument) \(v_2\)

3. Evaluate body \(e\) in env \(E\) extended with \(x := v_2\)
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) Syntax Error  
(b) 102  
(c) Type Error  
(d) 2  
(e) 100
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!***
• 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