Recap: Functions as “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Compose Functions:
Flexible way to build Complex functions from primitives.

News

- PA 3 due THIS Friday (5/2)
- Midterm NEXT Thursday (5/8)
Higher-order functions enable modular code
- Each part only needs local information

**Funcs taking/returning funcs**

Data Structure
Client
Uses list

Uses meta-functions:
- map, fold, filter
With locally-dependent funs
- (lt h), square etc.
Without requiring implementing
details of data structure

**“Map-Reduce” et al.**

Data Structure
Library
list

Provides meta-functions:
- map, fold, filter
to traverse, accumulate over
lists, trees etc.
Meta-functions don’t need client
info (tester ? accumulator ?)

Map-Reduce
Client

Web Analytics “Queries”
Clustering, Page Rank, etc
as map/reduce + ops

Map-Reduce
Library

Provides:
- map, reduce
to traverse, accumulate
over WWW (“Big Data”)
Distributed across “cloud”

---

Higher Order Functions Are Awesome...

..but how do they work
Let's start with the humble variable...

"Bind value of expr $e$ to variable $x$"

Later expressions can use $x$

- Most recent "bound" value used for evaluation

Sounds like C/Java?

NO!

Variables and Bindings

Q: How to use variables in ML?

Q: How to "assign" to a variable?

```
let x = e;;
```

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

let x = e;;
```

Environments ("Phone Book")

How ML deals with variables

- Variables = "names"
- Values = "phone number"

```
<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>64 : int</td>
</tr>
<tr>
<td>z</td>
<td>[4;64;68] : int list</td>
</tr>
</tbody>
</table>
```

...
Environments and Evaluation

ML begins in a “top-level” environment
• Some names bound (e.g. +,-, print_string...)

```
let x = e;;
```

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. Evaluate expr e in current env to get value v : t
2. Extend env to bind x to v : t
(Repeat with next binding)

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”

Q: What is the value of res?

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

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!
Q: What is the value of res?

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

Environments

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

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

Binding used to eval (f …)

Binding for subsequent x

New binding:
- No change or mutation
- Old binding frozen in f

let x = 0 ;;
let y = x + 1 ;;
let z a = (x, y) ;;
let x = 100 ;;

let res = z [] ;;
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 \((f \ e)\)

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!”
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`

```ml
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`

```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)
```

Q: What is the value of `res`?

(a) 120  
(b) 60  
(c) 20  
(d) 5  
(d) 1

```

```
```

Q: What is the value of `res`?

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

```
```
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!

Evaluating let-in in env $E$:
1. Evaluate expr $e_1$ in env $E$ to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ (only) to evaluate $e_2$

Let binding examples:

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

Nested bindings

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

Nested binding examples:

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

let z = 
  let y = 20 
  in 
    x * y + x 
  ;;
```
**Nested bindings**

```plaintext
define x = 10
in
  define y = 20
  in
    x * y

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 1:** Variables are names for values
- Environment: dictionary/phonebook
- Most recent binding used
- Entries never change
- New entries added

**Recap 2:** Big Exprs With Local Bindings
- let-in expression
- Variable “in-scope” in-expression
- Outside, variable not “in-scope”
Recap 3: Env Frozen at Func Definition

- Re-binding vars cannot change function
- Identical I/O behavior at every call
- Predictable code, localized debugging

Static/Lexical Scoping

- For each occurrence of a variable, a unique place where variable was defined!
  - Most recent binding in environment

- Static/Lexical: Determined from 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

Next: Functions

Q: What’s the value of a function?

Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, no matter what happened on Tuesday!”
Two ways of writing function expressions:

1. Anonymous functions:

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

2. Named functions:

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

Application: fancy word for “call”

\[(e_1 e_2)\]

- Function value \(e_1\)
- Argument \(e_2\)
- “apply” argument \(e_2\) to function value \(e_1\)

The type of any function is:

- \(T_1\) : 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 functions!

- \(\text{int} \rightarrow \text{int}\)
- \(\text{int} * \text{int} \rightarrow \text{bool}\)
- \((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})\)
Type of function application

Application: fancy word for “call”

\[(e_1 \ e_2)\]

- “apply” argument \(e_2\) to function value \(e_1\)

\[\begin{align*}
  e_1 & : T_1 \rightarrow T_2 \\
  e_2 & : T_1 \\
  (e_1 \ e_2) & : T_2
\end{align*}\]

- Argument must have same type as “input” \(T_1\)
- Result has the same type as “output” \(T_2\)

Functions

Two questions about function values:

What is the value:

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

Values of function = “Closure”

Two questions about function values:

What is the value:

1. ... of a function?

Values of function = “Closure”

Two questions about function values:

What is the value:

1. ... of a function?

Closure =

- Code of Fun. \((\text{formal } x + \text{ body } e)\)
- Environment at Fun. Definition
Q: Which vars in closure of \( f \)?

(a) \( x \)
(b) \( y \)
(c) \( x \ y \)
(d) \( x \ y \ z \)
(e) None

---

Values of functions: Closures

- **Function value** = “Closure”
  - <code + environment at definition>

- **Body not evaluated** until application
  - But type-checking when function is defined

---

Q: Which vars in closure of \( f \)?

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

---

Two questions about function values:

What is the value:

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

---

Functions

```ml
let a = 20;;
let f x =
  let y   = 1 in
  let g z = y + z in
  a + (g x)
;;
```

Values

```
# let x = 2+2;;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + 1;;
val x : int = 8
# f 0;;
val it : int = 4
```

```ml
let a = 20;;
let f x =
  let y   = 1 in
  let g z = y + z in
  a + (g x)
;;
```

```ml
fun x -> e
```

```ml
(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)
;;
f 0;;

Free vs. Bound Variables

let a = 20;;
let f x =
  let y = 1 in
  let g z = a + z in
    a + (g x)
;;
f 0;;

Q: Which vars are free in f?

(a) a
(b) x
(c) y
(d) z
(e) None

Free vs. Bound Variables

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:
• Non-bound occurrence
  a is “free” inside f

Frozen Environment
needed for values of free vars
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;

Values of function application

Two questions about function values:

What is the value:

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

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

Values of function application

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

1. Find 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 \( \texttt{res} \)?

(a) 4  (b) 5  (c) 6  (d) 11  (e) 12

Example

```
let x = 1;;
let y = 10;;
let \( f \) y = x + y;;
let x = 2;;
let y = 3;;
let \( \texttt{res} \) = \( f \) (x + y);;
```

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

```
let x = 1;;
let y = 10;;
let \( f \) y = x + y;;
let x = 2;;
let y = 3;;
let \( \texttt{res} \) = \( f \) (x + y);;
```

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

(a) 4  (b) 5  (c) 6  (d) 11  (e) 12

Example

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

Q: Closure value of \( \texttt{g} \)?

```
let x = 0 in
\( \texttt{g} \) 2
;;
let x = 100;;
let h y = x + y;;
```

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

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

Application: \( f \) (\( x + y \))
Eval \( \texttt{body} \) in \( \texttt{env} \) extended with \( \texttt{formal} \)\( \rightarrow \) 5
Eval \( \texttt{x+y} \) in \([\texttt{x} \rightarrow 1, \texttt{y} \rightarrow 5]\) \( \rightarrow \) 6
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!”