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.
Higher-order functions enable modular code
- Each part only needs local information

**Funcs taking/returning funcs**

- Data Structure
  - Client
  - Uses 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” et al.**

- Data Structure
  - Library
  - list

- Map-Reduce
  - Client
  - List

- Client
  - Uses: `map`, `fold`, `filter`
  - To traverse, accumulate over WWW (“Big Data”)

**Higher Order Functions Are Awesome...**

**Higher Order Functions.. but how do they work**
Variables and Bindings

Q: How to use variables in ML?
Q: How to “assign” to a variable?

`let x = e;;`

“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!
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”

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

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

New binding!

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

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 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 = fn

# let x = x + x ;
val x : int = 8;
# f 0;
val it : int = 4

Tuesday, October 23, 2012

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

Tuesday, October 23, 2012
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

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 ;;
```

Occurences 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!

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

Nested bindings

Evaluating let-in in env $E$:
1. Evaluate $e_1$ in env $E$ to get value $v : t$
2. Use extended $E [x |-> v : t]$ to evaluate $e_2$
Nested bindings

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

GOOD Formatting

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

BAD Formatting

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 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, there is a unique place where the 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?
Functions

Two ways of writing function expressions:

1. Anonymous functions:

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

2. Named functions:

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

Function Application

Application: fancy word for “call”

\[
(e1 \ e2)
\]

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

Functions

The type of any function is:

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

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

Type

\( T1 \rightarrow T2 \)

Functions

The type of any function is:

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

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

Type

\( T1 \rightarrow T2 \)

\( T1, T2 \) can be any types, including functions!

What's an example of?

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

Application: fancy word for “call”

\[(e1 \ e2)\]

- “apply” argument \(e2\) to function value \(e1\)

\[e1 : T1 \rightarrow T2 \quad e2 : T1\]
\[(e1 \ e2) : T2\]

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

---

**Values** of function = “Closure”

Two questions about function values:

What is the value:

1. ... of a function ? \(\text{fun } x \rightarrow e\)

**Closure** =

Code of Fun. \((\text{formal } x + \text{body } e)\)

+ Environment at Fun. Definition

---

**Functions**

Two questions about function values:

What is the value:

1. ... of a function ? \(\text{fun } x \rightarrow e\)

2. ... of a function “application” (call) ? \((e1 \ e2)\)
Values of functions: Closures

- Function value = "Closure"
  - <code + environment at definition>

- Body not evaluated until application
  - But type-checking 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
```

Tuesday, October 23, 2012

Functions

Two questions about function values:

What is the value:

1. ... of a function ?

2. ... of a function "application" (call) ?

```
fun x -> e
```

Tuesday, October 23, 2012

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;;
```

Environment frozen with function

Used to evaluate fun application

Which vars needed in frozen env?

Frozen Environment

Inside a function:

A "bound" occurrence:
1. Formal variable
2. Variable bound in let-in

A "free" occurrence:
- Non-bound occurrence

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

Tuesday, October 23, 2012

Tuesday, October 23, 2012
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:
Not bound occurrence
nothing is “free” inside f

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

Where do bound-vars values come from?

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. 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) \((e_1 \ e_2)\)

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

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

3. Evaluate body \(e\) in env \(E\) extended with \(x := v_2\)

Example

```ocaml
let x = 1;;
let 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 \(g\)?

```
Q: Closure value of g?

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

Example 3

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

Tuesday, October 23, 2012
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!”