Environments & Closures

Ranjit Jhala
UC San Diego
Recap: Functions as “first-class” values

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

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions → Using, (Taking) Functions

Iterator, Accumul, Reuse computation pattern w/o exposing local info
Functions are “first-class” values

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

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

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

Iterator, Accumul, Reuse computation pattern w/o exposing local info
Higher-order funcs enable **modular** code

- Each part only needs **local** information

---

**Funcs taking/returning funcs**

Uses meta-functions: `map`, `fold`, `filter`

With locally-dependent funs

`(lt h), square` etc.

Without requiring Implement. details of data structure

---

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.

Higher-order funcs enable **modular** code

- Each part only needs **local** information

---

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…
Higher Order Functions

..but how do they work
Next: Environments & Functions

Let's start with the humble variable...
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”
Variables and Bindings

```ocaml
# 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 expressions can use `x`

- **Most recent** “bound” value used for evaluation

Sounds like C/Java?

NO!
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>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>
Environments and Evaluation

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

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

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
```

New binding!
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”?

```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 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”?

```ocaml
# 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
```

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, : int-&gt;int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, : int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>
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”? 

```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 it : int = 4
```

<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Binding used to eval (f ...)

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 \( f \ e \)

Q: Why is this a good thing?

```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 it : int = 4
```

<table>
<thead>
<tr>
<th>Binding used to eval ( f \ e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
</tr>
</tbody>
</table>
| \( f \) | fn <code,
\[ \vdash \] >: 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
Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, no matter what happened on Tuesday!”
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
Q: What is the value of res?

\[
\text{let } f = \text{fun } x \rightarrow 1;;
\]
\[
\text{let } f = \text{fun } x \rightarrow \text{if } x<2 \text{ then } 1 \text{ else } (x \ast f(x-1));;
\]
\[
\text{let } res = f 5;;
\]

(a) 120
(b) 60
(c) 20
(d) 5
(e) 1
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 ;;
```

Occurences 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`?

```ocaml
let y = let x = 10 in
      x + x ;;

let res = (x, y);;
```

(a) Unbound Var 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 x = e1 in
  e2
;;
```
Local 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 $e_2$
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 $E [x \mapsto v : t]$ to evaluate $e_2$

```
let y =
  let
    x = 10
  in
    x * x
;;
```
Evaluating let-in in env $E$:
1. Evaluate expr $e_1$ in env $E$ to get value $v : t$
2. Use extended $E \left[ x \mapsto v : t \right]$ to evaluate $e_2$

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

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

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

GOOD Formatting

BAD Formatting
Example

```ocaml
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
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:

   `let fname = fun x -> e`

2. Named functions:

   `let 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:

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

The type of any function is:
- $T_1$: the type of the “input”
- $T_2$: the type of the “output”

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

What's an example of?
- `int -> int`
- `int * int -> bool`
- `(int -> int) -> (int -> int)`
Type of function application

Application: fancy word for “call”

\((e_1, e_2)\)

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

\[
\begin{align*}
\text{e}_1 &: T_1 \rightarrow T_2 \\
\text{e}_2 &: T_1 \\
(\text{e}_1 \text{ e}_2) &: T_2
\end{align*}
\]

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

```
fun x -> e
```

Closure =

Code of Fun. \( (\text{formal } x + \text{ body } e) \) + Environment at Fun. Definition
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 env. of f?

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

(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

```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 it : int = 4
```

<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Binding used to eval (f ...)

Binding for subsequent x
Q: Vars in closure-env of f?

```latex
let a = 20;;

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

;;
```

(a) a y
(b) a
(c) y
(d) z
(e) y z
Free vs. Bound Variables

```haskell
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?
Free vs. Bound Variables

Inside a function:

A “bound” occurrence:
1. Formal variable
2. Variable bound in `let-in`

A “free” occurrence:
- Non-bound occurrence

Frozen Environment

needed for values of free vars
Q: Which vars are free in f?

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

(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 \texttt{let-in-end}

\texttt{x, a, z} are “bound” inside \texttt{f}

A “free” occurrence:
Not bound occurrence
\texttt{nothing} is “free” inside \texttt{f}

\begin{verbatim}
let a = 20;;

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

f 0;
\end{verbatim}
Where do bound-vars values come from?

let \( a = 20; \);

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

\( f \ 0; \)

Bound values determined when function is evaluated ("called")
- Arguments
- Local variable bindings
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) \((e_1 \ e_2)\)

1. Find **closure** of \(e_1\)
2. Execute body of **closure** with param \(e_2\)

**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 \(x := v_2\)
Q: What is the value of \texttt{res}?

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

(a) 4  (b) 5  (c) 6  (d) 11  (e) 12
Q: What is the value of \texttt{res}?

\[
\begin{align*}
\text{let } x &= 1 ;; \\
\text{let } y &= 10 ;; \\
\text{let } f \ y &= x + y ;; \\
\text{let } x &= 2 ;; \\
\text{let } y &= 3 ;; \\
\text{let } \text{res} &= f \ (x + y) ;;
\end{align*}
\]

\[
\begin{align*}
f &\mapsto \text{formal} := y \\
\text{body} &:= x + y \\
\text{env} &:= [x|\mapsto 1] \\
x &\mapsto 2 \\
y &\mapsto 3 \\
x + y &\mapsto 5
\end{align*}
\]

Application: \( f \ (x + y) \)

 Eval \texttt{body} in \texttt{env} extended with \texttt{formal} \mapsto 5

Eval \( x+y \) in \([x|\mapsto 1, \ y|\mapsto 5]\) \mapsto 6
Example

```ocaml
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 of g?

- **formal** `z`
- **body** `x + y + z`
- **env** `[(x|->2, y|->4)]`

Eval **body** in **env** extended with **formal|-> 1**
Eval `x+y+z` in `[(x|->2, y|->4, z|->1)]` ===> 7
Q: What is the value of \( \text{res} \) ?

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

(a) Syntax Error
(b) 102
(c) Type Error
(d) 2
(e) 100
Example 3

```haskell
let \( f \ g = \
  \begin{align*}
  &\text{let } x = 0 \text{ in} \\
  &g \ 2 \\
  &
  \\
  \end{align*}

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!”