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:
• 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 this 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
```

**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 ($f \ldots$)

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

Tuesday, October 11, 2011

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 occurences 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)
```

Tuesday, October 11, 2011

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 x = e1 in
e2
```

Let-in is an expression!

Evaluating let-in in env E:
1. Evaluate expr e1 in env E to get value v : t
2. Use extended E [x := v : t] (only) to evaluate e2

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

Evaluating let-in in env E:
1. Evaluate expr e1 in env E to get value v : t
2. Use extended E [x := v : t] to evaluate e2

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

Tuesday, October 11, 2011
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 $E [x := v : t]$ to evaluate $e_2$

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

Nested bindings

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

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

Example

```ml
let rec filter (f,l) =
  if l = [] then []
else
  let h = hd l in
  let t = filter (f, tl l) in
    if (f h) then h::t else t
```

Correct Formatting
Nested function bindings

```
let a = 20;;
let f x =
  let y = 10 in
  let g z = y + z in
    a + (g x)
  ;;

f 0;
```

Recap

- Variables are **names for values**
  - Environment: dictionary/phonebook
  - Most recent binding used
  - Entries never changed, new entries added

- Environment **frozen at fun definition**
  - Re-binding variables cannot change a function
  - Same I/O behavior at every call

Recap

- Build complex expressions with **local bindings**
  - **let-in** expression
    - The **let**-binding is visible (in scope) inside **in**-expression
    - Elsewhere the binding is not visible

Static/Lexical Scoping

- For each occurrence of a variable, there is a **unique** place in program text where the variable was 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
Next: Functions

Recap: Environments

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

1. **Evaluate:**
   Find and use most recent value of variable

2. **Extend:** `let x = e ;;`
   Add new binding at end of “phone book”

Recap

- Variables are **names** for **values**
  - Environment: dictionary/phonebook
  - Most recent binding used
  - Entries never changed, new entries added

- Environment **frozen at fun definition**
  - Re-binding variables cannot change a function
  - Same I/O behavior at every call

Q: What’s the value of a function?
Functions Expressions

Two ways of writing function expressions:

1. Anonymous functions:

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

2. Named functions:

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

Function Application Expressions

Application: fancy word for “call”

\[
(e_1 e_2)
\]

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

Functions Type

The type of any function is:

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

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

\(T_1 \rightarrow T_2\)

Functions Type

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!

Whats an example of ?

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

Application: fancy word for “call”

\[(e_1 \ e_2)\]

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

\[
\begin{array}{c}
  e_1 : T_1 \rightarrow T_2 \\
  e_2 : T_1 \\
  (e_1 \ e_2) : T_2
\end{array}
\]

- 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

Two questions about function values:

What is the value:

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

Values of functions: Closures

- Function value = “Closure”
  - <code + environment at definition>
- Body not evaluated until application
  - but type-checking takes place 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
```

Binding used to eval \((f \ ...)\)

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

Binding for subsequent \(x\)
Free (vs. Bound) Variables

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:
- Not bound occurrence
- $a$ is “free” inside $f$

Environment frozen in closure used for values of free vars

Nested function bindings

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 $f$

Nested function bindings

$Q$: Where do values of bound vars come from?

Bound variable values determined when fun evaluated (“executed”)
- Arguments
- Local variable bindings

Values of function application

Application: fancy word for “call”

$e_1 e_2$

- “apply” the argument $e_2$ to the (function) $e_1$

Application Value:
1. Evaluate $e_1$ in current env to get (function) $v_1$
   - $v_1$ is code + env
   - code is (formal $x$ + body $e$), env is $E$
2. Evaluate $e_2$ in current env to get (argument) $v_2$
3. Evaluate body $e$ in env $E$ extended by binding $x$ to $v_2$
Example 1

```ml
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```

Example 2

```ml
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 (Code + Env)?

Code:
- Formal: `z`
- Body expr: `x + y + z`

Frozen Environment:
- `x` bound to `2`
- `y` bound to `4`

Example 3

```ml
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