Environments and Evaluation

ML begins in a “top-level” environment
• Some names bound

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

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. **Evaluate** expr 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

\[
- \text{val } x = 2+2;
\text{val } x = 4 : \text{int}
\]

\[
- \text{val } y = x \times x \times x;
\text{val } y = 64 : \text{int}
\]

\[
- \text{val } z = [x, y, x+y];
\text{val } z = [4, 64, 68] : \text{int list}
\]

\[
- \text{val } x = x + x;
\text{val } x = 8;
\]

New binding!

---

Environments

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

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

\[
- \text{val } x = 2+2;
\text{val } x = 4 : \text{int}
\]

\[
- \text{val } f = \text{fn } y \Rightarrow x + y;
\text{val } f = \text{fn} : \text{int} \Rightarrow \text{int}
\]

\[
- \text{val } x = x + x;
\text{val } x = 8 : \text{int}
\]

\[
- \text{val } it = 4 : \text{int}
\]

New binding:
• No change or mutation
• Old binding frozen in \( \varepsilon \)
**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”? 

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

**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`) 

**Q** Why is this a good thing?

A: Function behavior frozen at declaration
- Nothing entered afterwards affects function
- Same inputs always produce the same outputs
  - Localizes debugging
  - Localizes reasoning about the program
  - No “sharing” means no evil aliasing effects

**Local bindings**

Evaluating let-in-end 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`

```ml
let
  val x = 10
in
  let
    val y = 20
  in
    x * y
  end
end
```

**Recap: Variables, bindings**

- Environment: frozen at fun definition
  - Re-binding variables cannot change a function
  - Same I/O behavior at every call
  - Localized type-checking, reasoning, debugging
- Build complex expressions with local bindings
  - [let-in-end expression](#)
    - The last-binding is visible (in scope) inside let-in expression
    - Elsewhere the binding is not visible
    - Person writing function ensures that users don’t depend upon, use (or, heaven forbid, change!) the local variables
- Static/Lexical Scoping
  - Program text dictates where each variable gets its value

**Nested bindings**

Evaluating let-in-end 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`

```ml
let
  val x = 10
in
  let
    val y = 20
  in
    x * y
  end
end
```

**Cannot change the world**

Env. declaration frozen inside fun “value”
- Frozen env used to evaluate application (`f`) 

Q: Why is this a good thing?

A: Function behavior frozen at declaration
- Nothing entered afterwards affects function
- Same inputs always produce the same outputs
  - Localizes debugging
  - Localizes reasoning about the program
  - No “sharing” means no evil aliasing effects
Benefits of Static Scoping

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

- Static: Can be 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

Functions

Two ways of writing function expressions:

1. Anonymous functions:
   ```
   val fname = fn x => e
   ```

2. Named functions:
   ```
   fun 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”

```
val fname = fn x =>
```

```
fun fname =
```

T1 \rightarrow T2

T1, T2 can be any types, including functions!

What’s an example of ?

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

Application: fancy word for “call”

\[(e_1 \ e_2)\]

- “apply” argument \(e_2\) to function value \(e_1\)
  \[
  \begin{align*}
  e_1 \colon & \text{T1} \\
  e_2 \colon & \text{T2}
  \end{align*}
  \]
- Argument must have same type as “input” \(T_1\)
- Result has the same type as “output” \(T_2\)

Values of functions

- “Body” expression NOT evaluated until application
  - But type-checking takes place at compile time
  - I.e. when function is defined
- Remember: Function value is
  - \(<\text{code} + \text{environment at definition}>\)
  - “closure”

Values

- val \(x = 2\) + \(y\)
- val \(f = \text{fn} \ y \to \ y\)
- val \(x = x + x\)
- val \(z = 3\) : int;
- \(f 0\)
- val \(it = \text{it} : \text{int}\.

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

\[
\begin{align*}
\text{val} \ x &= \text{do} \ y \to \ y; \\
\text{val} \ f &= \text{fn} \ y \to \ y; \\
\text{val} \ x &= x + x; \\
\text{val} \ z &= 3; \\
\text{val} \ it &= \text{it};
\end{align*}
\]

Binding for subsequent \(x\)

Functions

Two questions about function values:

What is the value:

1. \(\ldots\) of a function?
2. \(\ldots\) of a function “application” (call)? \((e_1 \ e_2)\)

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
- is “free” inside \(f\)

Not bound occurrence:

Environment at definition, frozen inside “closure”, is used for values of free variables

Nested function bindings

\[
\begin{align*}
\text{val} \ a &= 20; \\
\text{fun} \ f &= \ \\
\text{let} & \text{val} \ a = 1 \\
\text{fun} \ g &= x + z \\
in & a = (g \ x) \\
\text{end}; \\
\text{end};
\end{align*}
\]

Free variable values determined

when function evaluated

“executed”

- From arguments
- Local var binding (evaluation)

Nested function bindings

\[
\begin{align*}
\text{val} \ a &= 20; \\
\text{fun} \ f &= \ \\
\text{let} & \text{val} \ a = 1 \\
\text{fun} \ g &= x + z \\
in & a = (g \ x) \\
\text{end}; \\
\text{end};
\end{align*}
\]
Values of function application

Application: fancy word for “call”

- “apply” the argument \( a_2 \) to the (function) \( a_1 \)

Application Value:
1. Evaluate \( a_1 \) in current env to get (function) \( v_1 \)
   - \( v_1 \) is code \( \Rightarrow \) env
   - code is (formal \( x \) \( \Rightarrow \) body \( \Rightarrow \) env \( \Rightarrow \) \( E \))
2. Evaluate \( a_2 \) in current env to get (argument) \( v_2 \)
3. Evaluate body \( e \) in env \( E \) extended by binding \( x \) to \( v_2 \)

Example 1

```plaintext
val x = 1;
fun f y = x + y;
val x = 2;
val y = 3;
f (x + y);
```

Example 2

```plaintext
val x = 1;
fun f y =
  let val x = y in
  fn z => x + y + z
end;
val x = 100;
val g = (f 4);
val 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

```plaintext
fun f g =
  let
    val x = 0
  in
    g 2
  end;
val x = 100;
fun h y = x + y;
f h;
```