The Substitution Model

- Recall the substitution model:
  - Bind variables at "let" constructs
  - Bind function arguments at function calls
  - Substitute bindings in the let body or function body

- Rules:
  \[ \text{let } \text{val } x = v \text{ in } e \text{ end } \rightarrow e[v/x] \]
  \[ (\text{fn } x \Rightarrow e)(v) \rightarrow e[v/x] \]

- Example:
  \[ \text{let } \text{val } x = 3 \text{ in } x * x \text{ end} \rightarrow 3 * 3 \rightarrow 9 \]

Problems

- Substitution model:
  - Useful for understanding program execution
  - Inefficient as an implementation

- Problem 1: We must traverse the code just to perform substitutions; the code will be traversed again when we execute it.

- Problem 2: Substitutions can lead to code blow-up
  \[ \text{let } \text{val } x = (1,2) \]
  \[ \text{val } y = (x,x) \]
  \[ \text{val } z = (y,y) \]
  \[ \text{in} \]
  \[ (z,z) \]
  \[ \text{end} \]

The Environment Model

- Solution: the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed

- Example:
  
<table>
<thead>
<tr>
<th>Program:</th>
<th>Environment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{let } \text{val } x = 2</td>
<td></td>
</tr>
<tr>
<td>\text{val } y = \text{&quot;hello&quot;}</td>
<td></td>
</tr>
<tr>
<td>\text{in}</td>
<td></td>
</tr>
<tr>
<td>\text{x + size(y)}</td>
<td></td>
</tr>
<tr>
<td>\text{end}</td>
<td></td>
</tr>
</tbody>
</table>

The Environment Model

- Solution: the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed

- Example:
  
<table>
<thead>
<tr>
<th>Evaluation:</th>
<th>Environment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\rightarrow \text{let } y = \text{&quot;hello&quot;}</td>
<td></td>
</tr>
<tr>
<td>\text{in}</td>
<td></td>
</tr>
<tr>
<td>\text{x + size(y)}</td>
<td></td>
</tr>
<tr>
<td>\text{end}</td>
<td></td>
</tr>
<tr>
<td>x = 2</td>
<td></td>
</tr>
</tbody>
</table>
### The Environment Model

- **Solution:** the environment model
  - Idea: use an environment to store bindings of variables
  - No substitutions
  - Environment is a map from variables to values
  - Values are looked up lazily, when needed

- **Example:**
  
  Evaluation:  
  \[ \Rightarrow x + \text{size}(y) \]
  Environment:  
  \[
  \begin{array}{ll}
  x & = 2 \\
  y & = "hello"
  \end{array}
  \]

### Environments

- Bindings added when entering a scope
- Bindings removed at end of scope
- Nested let blocks: how do we remove just the inner bindings?
- Idea: use a stack-like structure of bindings
  - Entering a scope: push new bindings, record the parent
  - Exiting a scope: move to the parent
  - Most recent binding = current environment
  - Least recent binding = TOP

### Variable Lookup

- To evaluate a variable, look it up in the environment
  - start with the last binding added to the environment and then explore the path towards TOP.
- Evaluating "x" in this environment yields 3:

  ![Variable Lookup Diagram]

### Boxed vs. Unboxed Values

- Values of primitive types are placed directly in the environment ("unboxed" values)
  - E.g., int, bool, real, char
- All other values are placed in the heap ("boxed" values)
  - Each heap cell drawn as a new box in the diagram
  - Examples: references, tuples, records, datatype constructors (hence lists), anonymous functions
  - The environment stores a pointer to the corresponding heap cell

  ```
  let val x = 1 
  val y = (2, 3, 4) 
  in ... 
  ```

### Let Expressions

To evaluate let val x = e1 in e2:

1. Evaluate e1 in the current environment
2. Extend the current environment with a binding that maps x to the value of e1
3. Evaluate e2 in the extended environment
4. Restore the old environment (i.e., remove the binding for x)
5. Return the value of e2
Let Example

\[
\text{let val } x = (1,2) \text{ in } (x,3) \text{ end}
\]

1. Evaluating \((1,2)\) yields a pointer to a heap cell.
2. Extend the environment with a binding for \(x\).

current env → TOP

result = \[
\begin{array}{c}
1 \\
2 \\
\end{array}
\]

current env → TOP

Let Example

\[
\text{let val } x = (1,2) \text{ in } (x,3) \text{ end}
\]

1. Evaluating \((1,2)\) yields a pointer to a heap cell.
2. Extend the environment with a binding for \(x\).
3. Evaluate the body of the let in the current env.
4. Restore the environment and return the result.

Multiple Declarations

- To evaluate:
  \[
  \begin{align*}
  \text{let val } x &= e1 \\
  \text{val y} &= e2 \\
  \text{in } e3 \\
  \text{end}
  \end{align*}
  \]

- Do the same the same thing as you would for:
  \[
  \begin{align*}
  \text{let val } x &= e1 \\
  \text{in } \text{let val y} &= e2 \\
  \text{in } e3 \\
  \text{end} \\
  \text{end}
  \end{align*}
  \]
Datatype Constructors

datatype list = Nil | Cons of int * list

- To evaluate Cons(e, e'):
  - evaluate e, e' to their values
  - allocate a new ref cell
  - place the values in the ref cell
  - return a pointer to the ref cell
- To evaluate Nil:
  - Treat it as an unboxed value because it does not carry data

References

- To evaluate ref e:
  - evaluate e to a value first
  - allocate a new ref cell
  - place the value in the ref cell
  - return a pointer to the ref cell
- Example: ref (3, 1, 2) evaluates to:

Ref Example

let val x = ref 1 in
val y = x
in
  x := 2; !y
end

Ref Example

let val x = ref 1 in
val y = x
in
  x := 2; !y
end

Ref Example

let val x = ref 1 in
val y = x
in
  x := 2; !y
end

Ref Example

let val x = ref 1 in
val y = x
in
  x := 2; !y
end
let val x = ref 1 in
  val y = x
  in
  x := 2; !y
end

Result = 2

x = 9

current env → y = 2

y = 2

Result = 2

Unreachable cells = “garbage”

x = 2

current env → y = 2

Result = 2

Garbage Collection

- Garbage cells are those heap cells not reachable from:
  - The current environment
  - Or from the result

- Garbage collection is the process of collecting the unreachable heap cells
  - Takes place as the program runs
  - Will discuss more about it later in the course
Functions

- Consider the following code:

```ml
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

- What value do we assign to f?
- Note: the body of f refers to variable x
  - What is the value of x?
- Solution: use a closure - (env, code) pair
  - env tells us about the values of unbound variables

Function Example

```ml
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

current env $\rightarrow$ TOP

1. Create function closure

Function Example

```ml
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

current env $\rightarrow$ x = 2

2. Bind f to the closure

Function Example

```ml
let val x = 2
    val f = fn z => z + x
in
    f 3
end
```

old env $\rightarrow$ x = 2

3. Use the closure env
Save the "old" env
let val x = 2
   val f = fn z => z + x
in
  f 3
end

x = 2
fn z=>z+x

4. Bind formal parameters

5. Evaluate function body

x = 3
z = 3

6. Restore env

7. Exit scope

---

**Function Calls**

To evaluate e1(e2):
1. Evaluate e1 - you must get a pointer to a closure.
2. Evaluate e2 to a value.
3. Save the current environment (and refer to it as the "old" environment).
4. Use the environment from the closure, extend it with binding for formal parameters.
5. Evaluate the body of the function within the extended environment; this is the result.
6. Restore the old environment (saved in step 3)
7. Return the result.

---

**Static vs. Dynamic Scoping**

- Consider this code:
  ```ml
  let val x = 2
     val f = fn z => z + x
  in
    f 3
  end
  ```
  - This is the case in ML, Java. Result = 5
- **Static scoping**: use the binding at the declaration (this is the environment saved in the closure)
- **Dynamic scoping**: use the binding at the call
  - Other languages (older LISP, Perl). Result = 4
let val r = ref (fn x=>x)
val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end

crt ← TOP

let val r = ref (fn x=>x)
val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end

crt ← TOP

let val r = ref (fn x=>x)
val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end

crt ← TOP

let val r = ref (fn x=>x)
val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end

old ← f

let val r = ref (fn x=>x)
val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end

old ← f

crt ← TOP

n = 2

crt ← TOP

old ← f

let val r = ref (fn x=>x)
val f = fn n=> if n<2 then 1 else n*(!r)(n-1)
in r := f; f 2
end

old ← f

n = 2

crt ← TOP

old ← f
Recursive Function Definitions

- To handle recursive function definition we need to extend the environment first, with an “incomplete” binding for the recursive function.
- Next, build the closure and make the environment in the closure point to the extended environment, that includes the function.
- Finally, bind the function symbol to the closure.
- We get a cycle:
  - the function symbol points to the closure
  - The environment in the closure points to the symbol

Recursion

```plaintext
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

```plaintext
fn n=> if n<2 then 1 else n*f(n-1)
```

Recursion

```plaintext
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

```plaintext
fn n=> if n<2 then 1 else n*f(n-1)
```

Recursion

```plaintext
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

```plaintext
fn n=> if n<2 then 1 else n*f(n-1)
```

Recursion

```plaintext
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

```plaintext
fn n=> if n<2 then 1 else n*f(n-1)
```

Recursion

```plaintext
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end
```

```plaintext
fn n=> if n<2 then 1 else n*f(n-1)
```
let fun f(n) = if n < 2 then 1 else n * f(n-1)
in f 2
end

Recursion

old n = 2
n = 1

top

fn n=> if n<2 then 1
else n*f(n-1)

Recursive call

Recursive call result

Result = 1

Result = 2

Result of first call

After the call

Comparison

current env

fn n=> if n<2 then 1
else n*f(n-1)

top

fn n=> if n<2 then 1
else n*f(n-1)

top