Substitution Model

- Represents computation as doing substitutions for bound variables at reduction of let, application:

  \[
  \text{let val } x = v \text{ in } e \Rightarrow e\{v/x\}
  \]

  \[
  (\text{fn}(x:t)=>e)(v) \Rightarrow e\{v/x\}
  \]

  \[
  \text{let val } x = \text{fn } z:'a=>z \text{ in } \\
  x(x(x)) \\
  \text{end} \\
  \Rightarrow (\text{fn } z=>z)((\text{fn } z=>z)(\text{fn } z=>z))
  \]
Problems

- Not a realistic implementation! Substitution is too slow.

\[(fn(x:t)=e)(v) \rightarrow e(v/x)\]

- Idea: use environment to record substitutions, do them only as needed

\[(fn(x:t)=e)(v) \rightarrow e\]

\[x = v\]

Could be many x’s in here

Environment Model

- No substitution, realistic cost model
- Environment is a finite map from variables to values
- Example:
  let val x = 2
  val y = “hello”
  val f = fn z:int=>x
  in f(x + size(y)) end

Evaluate:
\[f(x + size(y))\]
in environment:

\[x = 2\]
\[y = “hello”\]
\[f = fn z:int=>x?\]
Variables

• To evaluate a variable, look it up in the environment. To look it up, we start with the last binding added to the environment and then work towards the TOP.
• Evaluating “x” in this environment yields 3:

Let expressions

To evaluate \texttt{let val x = e1 in e2}:

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

let val x = (1,2) in #1 x end

1. Evaluating (1,2) yields a pointer to a tuple in memory.
Let Example

let val x = (1,2) in #1 x end

1. Evaluating (1,2) yields a pointer to a tuple in memory.

2. Extend the environment with a binding for x.

3. Evaluate the body of the let in the new environment. x evaluates to a pointer to the tuple, so #1 x evaluates to the first component, namely 1.
Let Example

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

1. Evaluating \((1,2)\) yields a pointer to a tuple in memory.
2. Extend the environment with a binding for \(x\).
3. Evaluate the body of the let in the new environment. \(x\) evaluates to a pointer to the tuple, so \(\#1 \ x\) evaluates to the first component, namely 1.
4. Restore the old environment.

5. Return the value we got: 1
Pictorial Overview:

- Primitive values like integers, reals, unit, or nil evaluate to themselves.

- A tuple value, such as (1,2,3) evaluates to a pointer to a box in memory containing the values of the sub-expressions:
  
  1 2 3

Multiple Declarations

To evaluate:

```plaintext
let val x = e1
  val y = e2
  val z = e3
in  e4
end
```

Do the same the same thing as you would for:

```plaintext
let val x = e1
in  let val y = e2
  in  let val z = e3
  in    e4
  end
end
```
Evaluation of Example

```ml
let val x = (3,4)
  val y = (x,x)
in
  #1(#2 y)
end
```

Evaluation of Example

```ml
let val x = (3,4)
  val y = (x,x)
in
  #1(#2 y)
end
```

current env → TOP

```ml
3 4
```

current env → TOP
Evaluation of Example

```plaintext
let val x = (3,4) in
  val y = (x,x)
in #1(#2 y) end
end

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

current env ⚪ x = 3 4

Evaluation of Example

```plaintext
let val x = (3,4) in
  val y = (x,x) in
    #1(#2 y) end
end
```

current env ⚪ x = 3 4

current env ⚪ TOP
Evaluation of Example

```haskell
let val x = (3,4) in
  val y = (x,x)
in
#1(#2 y)
end
```

Evaluation of Example

```haskell
let val x = (3,4) in
  val y = (x,x)
in
#1(#2 y)
end
```

```
3   4
x = current env
```

```
current env → y = current env
```

```
3   4
x = current env
```

```
 TOP
```

```
 TOP
```
Evaluation of Example

let val x = (3,4)
val y = (x,x)
in
#1(#2 y)
end

let val x = (3,4)
in
let val y = (x,x)
in
#1(#2 y)
end

Result = 3

Evaluation of Example

let val x = (3,4)
val y = (x,x)
in
#1(#2 y)
end

let val x = (3,4)
in
let val y = (x,x)
in
#1(#2 y)
end

Restore last env

Result is 3
**Evaluation of Example**

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

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

---

**Refs**

- To evaluate `ref e`, evaluate `e` to a value first, and then allocate a new ref cell, place the value in the ref cell, and return a pointer to the ref cell. For instance, `ref (1, 2, 3)` evaluates to:

  ![Ref Cell Diagram]

  ref cells = red boxes.
Ref Example

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

current env → TOP

Ref Example

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

current env → x = ✓ 2

TOP
Ref Example

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

Ref Example

```ocaml
let val x = ref 2 in
  val y = x
in
  x:=1; !y
end
```
Ref Example

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

Ref Example

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


**Functions**

```
let val x = 2
  val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

- How do we make sure the environment has the (correct) binding for `x`?
  - We must keep track of the environment at the point where the function was evaluated.
  - Function evaluation: `fn z:int => x`, not `f(size(x))`

- We create a **closure**
  - A pair of a function and its environment

**Static scope:**
ML, Java, Scheme, ...

**Dynamic scope:**
Perl, Python, BASIC

---

**Functions**

- To evaluate a function `(fn x => e)` create a **closure** out of the function and the current environment and return a pointer to the closure.
Creating closures

- To evaluate a function \((\text{fn } x \Rightarrow e)\) create a closure out of the function and the current environment and return a pointer to the closure.

I’ll draw closures using yellow.

---

Function Example

```haskell
let val x = 2
  val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

---
Function Example

let val x = 2
    val f = fn z:int => x
in
let val x = “bye”
in    f(size(x))
end

Function Example

let val x = 2
    val f = fn z:int => x
in
let val x = “bye”
in    f(size(x))
end
Function Example

```
let val x = 2
  val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

Function Calls

To evaluate e₁(e₂):
1. evaluate e₁ -- you should get a pointer to a closure.
2. evaluate e₂ to a value.
3. save the current environment -- we'll come back to it after the function call.
4. extend the environment of the closure, mapping the formal argument to the actual argument.
5. evaluate the body of the function within the extended environment -- this gives us our result value.
6. restore the old environment (saved in step 3)
7. return the result.
Function Call Example

```
let val x = 2
  val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

1. Evaluate e1, e2

```
1. Evaluate e1, e2
2. Save environ.
```
Function Call Example

```ocaml
let val x = 2
val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

1. Evaluate e1, e2
2. Save environ.
3. Extend env with actual

Diagram:

- Saved env: `fn z:int => x`
- Current env: `z = 3`, `x = 2`
- Result: 2

---

Function Call Example

```ocaml
let val x = 2
val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

1. Evaluate e1, e2
2. Save environ.
3. Extend env with actual
4. Evaluate body (result = 2)
### Function Call Example

```
let val x = 2
val f = fn z:int => x
in
let val x = "bye"
in
  f(size(x))
end
```

1. Evaluate e1, e2
2. Save environ.
3. Extend env with actual
4. Evaluate body
   (result= 2)
5. Restore env.
   (result= 2)

### Creating a cycle

```
let val x = ref (fn x:int => x)
val f = fn n:int =>
  if n <= 1 then 1 else n * (!x)(n-1)
in
  x := f;
  f(3)
end
```
Creating a cycle

let val x = ref (fn x => x)
    val f = fn n =>
      if n <= 1 then 1 else n * (!x)(n-1)
in
    x := f;
    f(3)
end
let val x = ref (fn x => x)
   val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
f(3)
end

Creating a cycle

let val x = ref (fn x => x)
   val f = fn n =>
       if n <= 1 then 1 else n * (!x)(n-1)
in
   x := f;
f(3)
end
let val x = ref (fn x => x)
  val f = fn n =>
    if n <= 1 then 1 else n * (!x)(n-1)
in
  x := f;
  f(3)
end

Creating a cycle

let val x = ref (fn x => x)
  val f = fn n =>
    if n <= 1 then 1 else n * (!x)(n-1)
in
  x := f;
  f(3)
end

Creating a cycle
Creating a cycle

```ml
let val x = ref (fn x => x)
    val f = fn n =>
        if n <= 1 then 1 else n * (!x)(n-1)
    in
    x := f;
    f(3)
end
```

Note: !x is the same as f
Creating a cycle

```haskell
let val x = ref (fn x => x)
val f = fn n =>
  if n <= 1 then 1 else n * (!x)(n-1)
in
  x := f;
f(3)
end
```

Result = 1
Creating a cycle

let val x = ref (fn x => x)
val f = fn n =>
    if n <= 1 then 1 else n * (!x)(n-1)
in
    x := f;
    f(3)
end

Result = 2*1

Creating a cycle

let val x = ref (fn x => x)
val f = fn n =>
    if n <= 1 then 1 else n * (!x)(n-1)
in
    x := f;
    f(3)
end

Result = 3*2*1
Creating a cycle

```ml
let val x = ref (fn x => x)
  val f = fn n =>
    if n <= 1 then 1 else n * (!x)(n-1)
in
  x := f;
  f(3)
end
```

Recursion

```ml
let fun f(n) =
  if n <= 1 then 1 else n * f(n-1)
in
  f(3)
end
```

1. create a new binding for f before creating the closure and extend the current environment with it (but don’t put in the value yet.)
2. now create a closure for f that uses the extended environment.
3. fix the binding to use the closure!
Recursion

let fun f(n) => if n <= 1 then 1 else n * f(n-1)
in
  f(3)
end

1. create a new binding for f before creating the closure and extend the current environment with it (but don’t put in the value yet.)
2. now create a closure for f that uses the extended environment.
3. fix the binding to use the closure!
Recursion

let fun \( f(n) \) => if \( n \leq 1 \) then 1 else \( n \times f(n-1) \) in
\( f(3) \)
end

1. create a new binding for \( f \) before creating the closure and extend the current environment with it (but don’t put in the value yet.)
2. now create a closure for \( f \) that uses the extended environment.
3. fix the binding to use the closure!

Cycle

let fun \( f(n) \) => if \( n \leq 1 \) then 1 else \( n \times f(n-1) \) in
\( f(3) \)
end

1. create a new binding for \( f \) before creating the closure and extend the current environment with it (but don’t put in the value yet.)
2. now create a closure for \( f \) that uses the extended environment.
3. fix the binding to use the closure!

- Closure points to environment
- Environment points to closure
Cycle

\[
\begin{align*}
\text{let} & \quad \text{fun } f(n) \Rightarrow \text{if } n \leq 1 \text{ then } 1 \text{ else } n \times f(n-1) \\
\text{in} & \quad f(3) \\
\end{align*}
\]

- Closure points to environment
- Environment points to closure

1. create a new binding for \( f \) before creating the closure and extend the current environment with it (but don’t put in the value yet.)
2. now create a closure for \( f \) that uses the extended environment.
3. fix the binding to use the closure!

Comparison

\[
\begin{align*}
\text{current env} & \quad f= \\
\text{TOP} & \quad \text{fn } n =\Rightarrow \text{if } n \leq 1 \text{ then } 1 \text{ else } n \times (!x)(n-1) \\
\text{current env} & \quad f= \\
\text{TOP} & \quad \text{fn } n =\Rightarrow \text{if } n \leq 1 \text{ then } 1 \text{ else } n \times f(n-1)
\end{align*}
\]