Substitution Model

- Represents computation as doing substitutions for bound variables at reduction of let, application:
  
  \[
  \begin{align*}
  \text{let } & \text{val } x = v \text{ in } e \implies e(v/x) \\
  \text{let } & \text{val } x : t \Rightarrow e(v) \implies e(v/x)
  \end{align*}
  \]

| let val x = \text{fn } z : \text{a} \Rightarrow z \text{ in } x(x(x)) |
| end |
| \implies (\text{fn } z : \Rightarrow z)(\text{fn } z : \Rightarrow z)(\text{fn } z : \Rightarrow z) |

Problems

- Not a realistic implementation: substitution is too slow

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

... could be many x's to substitute for...

- Doesn't handle refs: no aliasing

\[
\begin{align*}
\text{let } & \text{val } a = \text{ref } 2 \text{ in } \\
& (\text{fn } x,y \Rightarrow \ldots y \ldots ) (a,a) \\
\text{end }
\end{align*}
\]

\[
\implies (\text{fn } x,y \Rightarrow \ldots x = \ldots !y \ldots ) (\text{ref } 2, \text{ref } 2) ?
\]

Environment Model

Don't substitute for variables; look them up lazily in an environment!

- No substitution, realistic cost model
- The environment is a finite map from variables to values

- Example:

\[
\begin{align*}
\text{let } & \text{val } x = 2 \\
\text{val } & y = "hello" \\
\text{val } & f = \text{fn } z : \text{int} \Rightarrow x \\
\text{in } & f(x + \text{size}(y)) \text{ end}
\end{align*}
\]

Evaluate:

\[
\begin{align*}
f(x + \text{size}(y)) & \quad \text{in environment:} \\
\end{align*}
\]

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 nil.
- Evaluating "x" in this environment yields 3:

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

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

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:

\[
\begin{align*}
&\text{let } \text{val } x = e_1 \\
&\text{val } y = e_2 \\
&\text{val } z = e_3 \\
&\text{in } e_4 \\
&\text{end}
\end{align*}
\]

Do the same the same thing as you would for:

\[
\begin{align*}
&\text{let } \text{val } x = e_1 \\
&\text{in } \text{let } \text{val } y = e_2 \\
&\text{in } \text{let } \text{val } z = e_3 \\
&\text{in } e_4 \\
&\text{end} \\
&\text{end} \\
&\text{end}
\end{align*}
\]

**Example**

\[
\begin{align*}
&\text{let } \text{val } x = (3,4) \\
&\text{val } y = (x,x) \\
&\text{in } \#1(\#2 y) \\
&\text{end}
\end{align*}
\]

**Evaluation of Example**

\[
\begin{align*}
&\text{let } \text{val } x = (3,4) \\
&\text{val } y = (x,x) \\
&\text{in } \#1(\#2 y) \\
&\text{end}
\end{align*}
\]

current env \rightarrow \text{nil}
To evaluate \texttt{ref e}, evaluate \texttt{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, \texttt{ref (1, 2, 3)} evaluates to:

![Ref cells = red boxes.](image)
Ref Example

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: \(fn \ x : int => x\), not \(f(size(x))\)
- We create a closure
  - A pair of a function and its environment

Creating closures

- 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.

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 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

```plaintext
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

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

Function Call Example

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

Function Call Example

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

Function Call Example

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

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

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
```

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

```
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

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

Note: !x is the same as f

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

Recursion

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

No value for f yet!

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 <= 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!

Cycle

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

- Closure points to environment
- Environment points to closure

Comparison