

More Interpreters, Operational Semantics, and Lazy Evaluation

(you are not responsible for
this lecture or operational semantics
on the first midterm)

COS 326

David Walker

Princeton University

Assignment #4 Overview

2

Given: a *substitution-based* interpreter with more features (booleans, pairs, lists, match)

Goal: Build a new *environment-based* interpreter that does the same thing, but faster.

substitution: good for reasoning about programs

environments: more efficient

AN ENVIRONMENT MODEL FOR PROGRAM EXECUTION

Substitution

4

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =  
  let (b, x, y) = arg in  
  if b then  
    (fun n -> n + x)  
  else  
    (fun n -> n + y)  
  
choose (true, 1, 2)
```

Substitution

5

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =  
  let (b, x, y) = arg in  
  if b then  
    (fun n -> n + x)  
  else  
    (fun n -> n + y)  
  
choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
choose (true, 1, 2)
```

Substitution

6

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =  
  let (b, x, y) = arg in  
  if b then  
    (fun n -> n + x)  
  else  
    (fun n -> n + y)  
  
choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
choose (true, 1, 2)  
-->  
let (b, x, y) = (true, 1, 2) in  
if b then (fun n -> n + x)  
else (fun n -> n + y)
```

Substitution

7

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =  
  let (b, x, y) = arg in  
  if b then  
    (fun n -> n + x)  
  else  
    (fun n -> n + y)  
  
choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
choose (true, 1, 2)  
-->  
let (b, x, y) = (true, 1, 2) in  
if b then (fun n -> n + x)  
else (fun n -> n + y)  
-->  
if true then (fun n -> n + 1)  
else (fun n -> n + 2)
```

Substitution

8

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =  
  let (b, x, y) = arg in  
  if b then  
    (fun n -> n + x)  
  else  
    (fun n -> n + y)  
  
choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
choose (true, 1, 2)  
-->  
  let (b, x, y) = (true, 1, 2) in  
  if b then (fun n -> n + x)  
  else (fun n -> n + y)  
-->  
  if true then (fun n -> n + 1)  
  else (fun n -> n + 2)  
-->  
  (fun n -> n + 1)
```

Substitution

9

How much work does the interpreter have to do?

traverse the
entire function
body, making
a new copy with
substituted values

```
choose (true, 1, 2)
-->
let (b, x, y) = (true, 1, 2) in
  if b then (fun n -> n + x)
  else (fun n -> n + y)
-->
  if true then (fun n -> n + 1)
  else (fun n -> n + 2)
-->
  (fun n -> n + 1)
```



Substitution

10

How much work does the interpreter have to do?

```
choose (true, 1, 2)
-->
let (b, x, y) = (true, 1, 2) in
if b then (fun n -> n + x)
else (fun n -> n + y)
-->
if true then (fun n -> n + 1)
else (fun n -> n + 2)
-->
(fun n -> n + 1)
```

traverse the
entire function
body, making
a new copy with
substituted values

traverse the
entire function
body, making
a new copy with
substituted values

Substitution

11

How much work does the interpreter have to do?

traverse the
entire function
body, making
a new copy with
substituted values

traverse the
entire function
body, making
a new copy with
substituted values

```
choose (true, 1, 2)
-->
let (b, x, y) = (true, 1, 2) in
if b then (fun n -> n + x)
else (fun n -> n + y)
-->
if true then (fun n -> n + 1)
else (fun n -> n + 2)
-->
(fun n -> n + 1)
```

Every step takes time proportional
to the size of the program.

We had to traverse the “else” branch
of the if twice, even though we never executed it!

The Substitution Model is Expensive

12

The substitution model is a *specification*.

We can implement such a model in a different way

- provided our results coincide.

Efficient interpreters use *environments* rather than substitution.

You can think of an environment as *delaying* substitution until it is needed.

Environment Models

13

An *environment* is a key-value store where the keys are variables and the values are ... programming language values.

Example:

[x -> 1; b -> true; y -> 2]

this environment:

- binds 1 to x
- binds true to b
- binds 2 to y

You can really think of an environment as a *suspended substitution* -- a substitution waiting to be applied when needed.

Execution with Environment Models

14

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```



Form of the semantic relation:

$e1 \rightarrow e2$

Execution with Environment Models

15

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```

Form of the semantic relation:

$e1 \rightarrow e2$

Execution with environments:

```
([], let x = 3 in
  let b = true in
  if b then x else 0)
```

Form of the semantic relation:

$(env1, e1) \Rightarrow (env2, e2)$

Execution with Environment Models

16

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```

Execution with environments:

```
([], let x = 3 in
  let b = true in
  if b then x else 0)
==>
([x->3], let b = true in
  if b then x else 0)
```

Execution with Environment Models

17

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```

Execution with environments:

```
([], let x = 3 in
  let b = true in
  if b then x else 0)
==>
([x->3], let b = true in
  if b then x else 0)
==>
([x->3; b->true], if b then x else 0)
```

Execution with Environment Models

18

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```

Execution with environments:

```
([], let x = 3 in
  let b = true in
  if b then x else 0)
==>
([x->3], let b = true in
  if b then x else 0)
==>
([x->3; b->true], if b then x else 0)
==>
([x->3; b->true], if true then x else 0)
```

Execution with Environment Models

19

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```

Execution with environments:

```
([], let x = 3 in
  let b = true in
  if b then x else 0)
==>
([x->3], let b = true in
  if b then x else 0)
==>
([x->3; b->true], if b then x else 0)
==>
([x->3; b->true], if true then x else 0)
==>
([x->3; b->true], x)
```

Execution with Environment Models

20

Execution with substitution:

```
let x = 3 in
let b = true in
if b then x else 0
-->
let b = true in
if b then 3 else 0
-->
if true then 3 else 0
-->
3
```

Execution with environments:

```
([], let x = 3 in
  let b = true in
  if b then x else 0)
==>
([x->3], let b = true in
  if b then x else 0)
==>
([x->3; b->true], if b then x else 0)
==>
([x->3; b->true], if true then x else 0)
==>
([x->3; b->true], x)
==>
([x->3; b->true], 3)
```

Another Example (Substitution Model)

21

```
(fun x ->  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x) 10
```

-->

```
(let f = fun y -> y + x in  
  let x = 3 in  
  f x) [10/x]
```

-->

```
let f = fun y -> y + 10 in  
let x = 3 in  
f x
```

-->

```
let x = 3 in  
(fun y -> y + 10) x
```

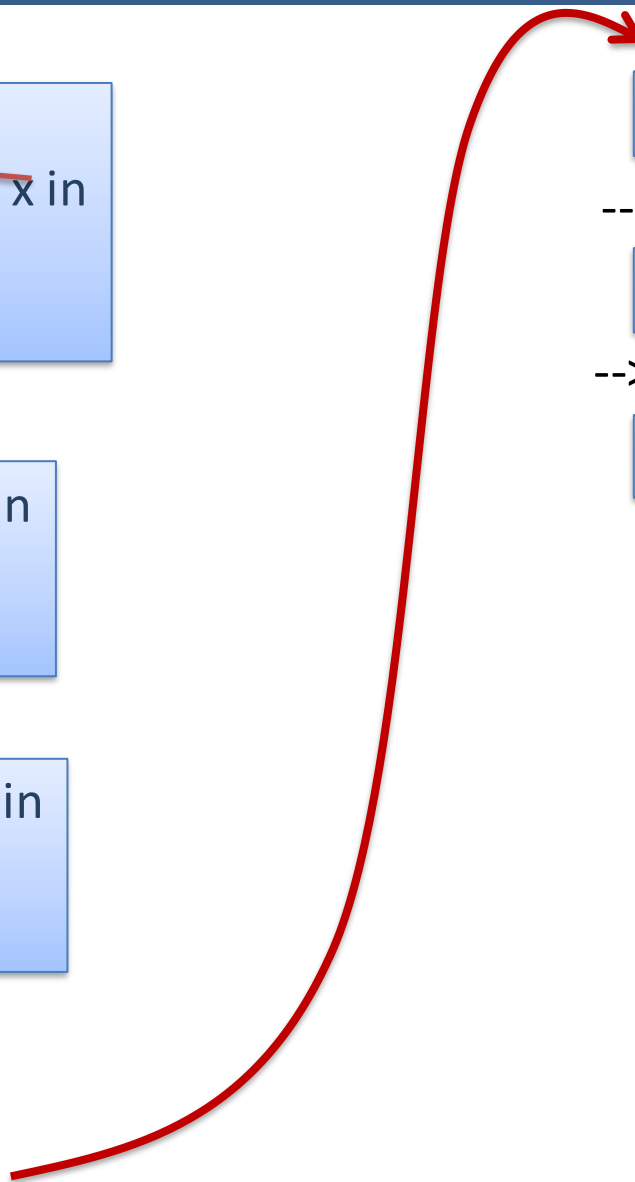
```
(fun y -> y + 10) 3
```

-->

```
3 + 10
```

-->

```
13
```



Another Example

22

```
([],  
 (fun x ->  
   let f = fun y -> y + x in  
   let x = 3 in  
   f x) 10 )
```

Another Example

23

```
([],  
 (fun x ->  
   let f = fun y -> y + x in  
   let x = 3 in  
   f x) 10 )
```

==>

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

Another Example

24

```
([],  
 (fun x ->  
   let f = fun y -> y + x in  
   let x = 3 in  
   f x) 10 )
```

==>

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

==>

```
([x -> 10; f -> fun y -> y + x],  
 let x = 3 in  
 f x )
```

Another Example

25

```
([],  
 (fun x ->  
   let f = fun y -> y + x in  
   let x = 3 in  
   f x) 10 )
```

==>

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

==>

```
([x -> 10; f -> fun y -> y + x],  
 let x = 3 in  
 f x )
```

what next? We need to put $x \rightarrow 3$ into the environment. But how?

Another Example

26

```
([],  
(fun x ->  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x) 10 )
```

==>

```
([x -> 10],  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x )
```

==>

```
([x -> 10; f -> fun y -> y + x],  
  let x = 3 in  
  f x )
```



what next?

Another Example

27

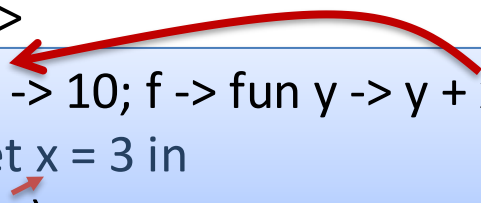
```
([],  
 (fun x ->  
   let f = fun y -> y + x in  
   let x = 3 in  
   f x) 10 )
```

==>

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

==>

```
([x -> 10; f -> fun y -> y + x],  
 let x = 3 in  
 f x )
```



==>

(1)

```
([x -> 3; f -> fun y -> y + x],  
 f x )
```

(1) **wrong**

- use of x in function now refers to wrong value (now 3)

(2) **wrong**

- use of x in f x now refers to wrong value (now 10)

(3) **ambiguous?**

- uncertain which value of x to choose!
- needs more book keeping!
- which x definition goes with which code

(2)

```
([x -> 10;  
 f -> fun y -> y + x],  
 f x )
```

(3)

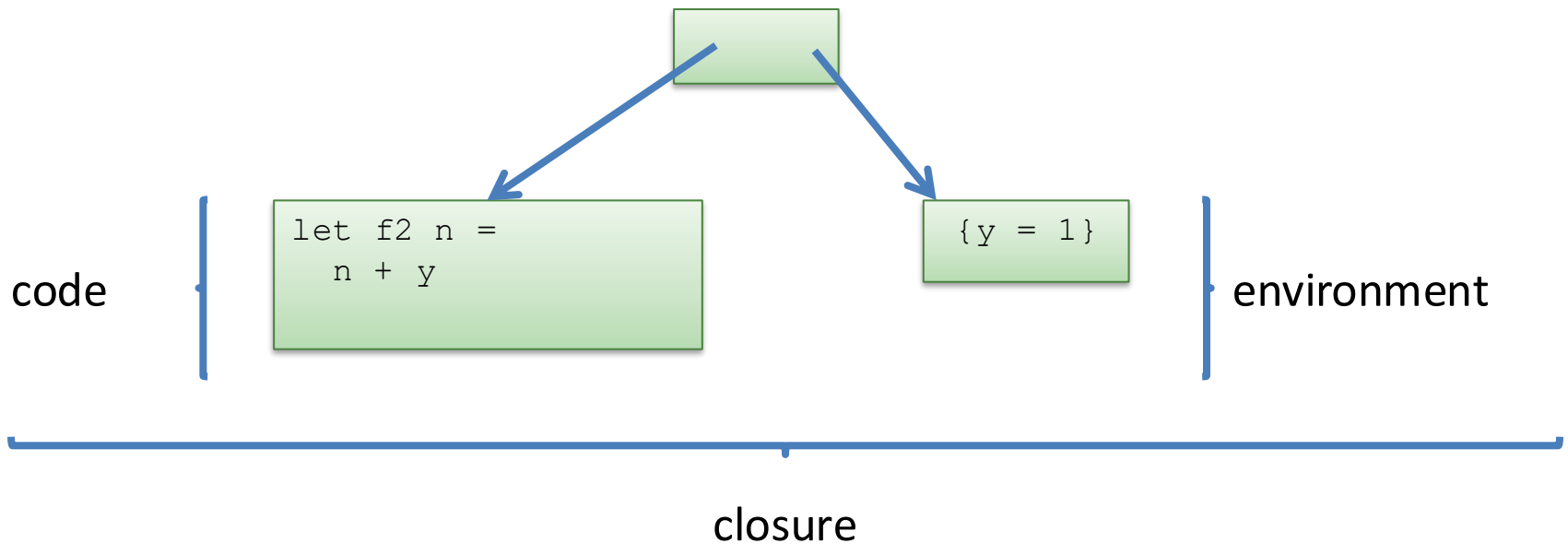
```
([x -> 10; x -> 3;  
 f -> fun y -> y + x],  
 f x )
```

Solution

34

Functions must carry with them the appropriate environment

A *closure* is a pair of code and environment

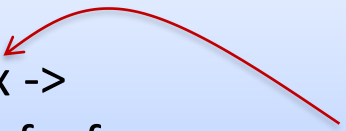


In the environment model, *function definitions* evaluate to *function closures*

Another Example

35

```
([],  
(fun x ->  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x) 10 )
```



Another Example

36

```
([],  
  (fun x ->  
    let f = fun y -> y + x in  
    let x = 3 in  
    f x) 10 )
```

closure created from function:

==>

```
([],  
  ((closure [] x -> let f = fun y -> y + x in let x = 3 in f x) 10 )
```

Another Example

37

```
([],  
  (fun x ->  
    let f = fun y -> y + x in  
    let x = 3 in  
    f x) 10 )
```

==>

```
([],  
  ((closure [] x -> let f = fun y -> y + x in let x = 3 in f x) 10 )
```

==>

```
([x = 10],  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x )
```

closure called, current environment replaced by closure environment
(but they are both empty so no observable change)

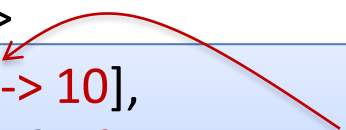
then environment extended with argument x and body executed

Another Example

38

...

$\Rightarrow$



```
([x -> 10],  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x )
```

Another Example

39

...

$\Rightarrow$

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

$\Rightarrow$

```
([x -> 10; f -> closure [x->10] y = y + x] ,  
 let x = 3 in  
 f x )
```

environment

parameter

body

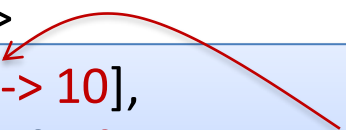
Another Example

40

...

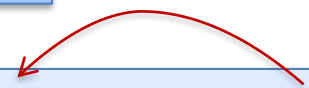
$\Rightarrow$

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```



$\Rightarrow$

```
([x -> 10; f -> closure [x->10] y = y + x],  
 let x = 3 in  
 f x )
```



$\Rightarrow$

```
([x -> 3; f -> closure [x->10] y = y + x],,  
 f x )
```



Another Example

41

...

$\Rightarrow$

```
([x -> 10],  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x )
```

$\Rightarrow$

```
([x -> 10; f -> closure [x->10] y = y + x],  
  let x = 3 in  
  f x )
```

$\Rightarrow$

```
([x -> 3; f -> closure [x->10] y = y + x],  
  f x )
```

```
([x -> 3; f -> closure [x->10] y = y + x],  
  (closure [x->10] y = y + x) x )
```

Another Example

42

...

==>

```
([x -> 10],  
  let f = fun y -> y + x in  
  let x = 3 in  
  f x )
```

==>

```
([x -> 10; f -> closure [x->10] y = y + x],  
  let x = 3 in  
  f x )
```

==>

```
([x -> 3; f -> closure [x->10] y = y + x],  
  f x )
```

```
([x -> 3; f -> closure [x->10] y = y + x],  
  (closure [x->10] y = y + x) x )
```

==>

```
([x -> 3; f -> closure [x->10] y = y + x],  
  (closure [x->10] y = y + x) 3 )
```

Another Example

43

...

==>

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

==>

```
([x -> 10; f -> closure [x->10] y = y + x],  
 let x = 3 in  
 f x )
```

==>

```
([x -> 3; f -> closure [x->10] y = y + x],  
 f x )
```

```
([x -> 3; f -> closure [x->10] y = y + x],  
 (closure [x->10] y = y + x) x )
```

==>

```
([x -> 3; f -> closure [x->10] y = y + x],  
 (closure [x->10] y = y + x) 3 )
```

==>

```
([x -> 10; y -> 3],  
 y + x )
```

When you call a closure,
replace the current
environment with the
closure's environment,
and bind the parameter
to the argument

Another Example

44

...

==>

```
([x -> 10],  
 let f = fun y -> y + x in  
 let x = 3 in  
 f x )
```

==>

```
([x -> 10; f -> closure [x->10] y = y + x],  
 let x = 3 in  
 f x )
```

==>

```
([x -> 3; f -> closure [x->10] y = y + x],  
 f x )
```

```
([x -> 3; f -> closure [x->10] y = y + x],  
 (closure [x->10] y = y + x) x )
```

==>

```
([x -> 3; f -> closure [x->10] y = y + x],  
 (closure [x->10] y = y + x) 3 )
```

==>

```
([x -> 10; y -> 3],  
 y + x )
```

==>

```
([x -> 10; y -> 3],  
 3 + 10 )
```

==>

```
([x -> 10; y -> 3],  
 13 )
```

FORMALIZING THE OPERATIONAL SEMANTICS

Environment-Based Operational Semantics

46

expressions $e ::= x \mid n \mid e_1 + e_2 \mid \text{let } x = e \text{ in } e$
 $\mid \text{fun } x = e \mid \text{closure } [\text{env}] x = e \mid e e$

values $v ::= n \mid \text{closure } [\text{env}] x = e$

environments $\text{env} ::= . \mid \text{env}, x = v$

Form of big step operational relation:

$(\text{env}, e) ==> v$

Environment-Based Operational Semantics

47

expressions $e ::= x \mid n \mid e_1 + e_2 \mid \text{let } x = e \text{ in } e$
 $\mid \text{fun } x = e \mid \text{closure } [\text{env}] x = e \mid e e$

values $v ::= n \mid \text{closure } [\text{env}] x = e$

environments $\text{env} ::= . \mid \text{env}, x = v$

Form of big-step operational relation:

$(\text{env}, e) ==> v$

Operational Rules

48

$$\frac{\text{env}(x) = v}{\text{-----}(\text{var})} (\text{env}, x) ==> v$$
$$\frac{}{\text{-----}(\text{val})} (\text{env}, v) ==> v$$
$$\frac{(\text{env}, e1) ==> v1 \quad (\text{env}[x=v1], e2) ==> v2}{\text{-----}(\text{let})} (\text{env}, \text{let } x = e1 \text{ in } e2) ==> v2$$
$$\frac{(\text{env}, e1) ==> v1 \quad (\text{env}, e2) ==> v2 \quad \text{eval_op}(v1, +, v2) = v}{\text{-----}(+)} (\text{env}, e1 + e2) ==> v$$

Example

49

$([], \text{let } x = (\text{let } y = 1 \text{ in } y+2) \text{ in } x) ==> ?$

Example

50

$([], \text{let } y = 1 \text{ in } y+2) ==> ?$

----- (let)

$([], \text{let } x = (\text{let } y = 1 \text{ in } y+2) \text{ in } x) ==> ?$

Example

51

([], 1) ==> ?

----- (let)
([], let y = 1 in y+2) ==> ?

----- (let)
([], let x = (let y = 1 in y+2) in x) ==> ?

Example

52

----- (val)

$([], 1) \Rightarrow 1$

----- (let)

$([], \text{let } y = 1 \text{ in } y+2) \Rightarrow ?$

----- (let)

$([], \text{let } x = (\text{let } y = 1 \text{ in } y+2) \text{ in } x) \Rightarrow ?$

Example

53

----- (val)

([], 1) ==> 1

([y=1]), **y+2**) ==> ?

----- (let)

([], let y = 1 in **y+2**) ==> 3

----- (let)

([], let x = (let y = 1 in y+2) in x) ==> 3

Example

54

$$\begin{array}{l} [y=1](y) = 1 \\ \text{----- (var)} \quad \text{----- (val)} \\ ([y=1]), y ==> 1 \quad ([y=1]), 2 ==> 2 \quad \text{eval_op}(1,+,2) = 3 \\ \text{----- (val)} \quad \text{----- (+)} \\ ([], 1) ==> 1 \quad ([y=1]), y+2 ==> 3 \\ \text{----- (let)} \\ ([], \text{let } y = 1 \text{ in } y+2) ==> ? \\ \text{----- (let)} \\ ([], \text{let } x = (\text{let } y = 1 \text{ in } y+2) \text{ in } x) ==> ? \end{array}$$

Example

55

... omitted ...

----- (val)	----- (+)
$([], 1) \Rightarrow 1$	$([y=1]), y+2 \Rightarrow 3$
-----	----- (let) ----- (var)
$([], \text{let } y = 1 \text{ in } y+2) \Rightarrow 3$	$[x=3](x) = 3$
-----	----- (let)
$([], \text{let } x = (\text{let } y = 1 \text{ in } y+2) \text{ in } x) \Rightarrow ?$	

Big Step Operational Rules

56

$$\frac{}{(\text{env}, \text{fun } x = e) \Rightarrow \text{closure } [\text{env}] x = e} \quad (\text{fun})$$
$$\frac{(\text{env}', e1) \Rightarrow \text{closure } [\text{env}] x = e \quad (\text{env}', e2) \Rightarrow v2 \quad ([\text{env}, x=v2], e) \Rightarrow v}{(\text{env}', e1 \ e2) \Rightarrow v} \quad (\text{app})$$

Closure Example

57

$$\frac{(\text{env}, e1) \Rightarrow \text{closure } [\text{env}] \ x = e}{(\text{env}, \text{fun } x = e) \Rightarrow \text{closure } [\text{env}] \ x = e} \quad (\text{fun})$$

$$\frac{(\text{env}', e1) \Rightarrow \text{closure } [\text{env}] \ x = e \quad (\text{env}', e2) \Rightarrow v2 \quad ([\text{env}, x=v2], e) \Rightarrow v}{(\text{env}', e1 \ e2) \Rightarrow v} \quad (\text{app})$$

The diagram illustrates the evaluation of the expression $([y=3], (\text{fun } x = x + y) \ 7) \Rightarrow 10$ through several steps:

- Top Row (Environment Frames):**
 - Left: $([y=3][x=7], x) \Rightarrow 7$ (labeled (var))
 - Middle: $([y=3][x=7], y) \Rightarrow 3$ (labeled (var))
 - Right: $\text{eval_op}(7, +, 3) = 10$
- Bottom Row (Environment Frames):**
 - Left: $([y=3], \text{fun } x = x + y) \Rightarrow \text{closure } [y=3] \ x = x + y$ (labeled (let))
 - Middle: $([y=3], 7) \Rightarrow 7$ (labeled (val))
 - Right: $([y=3][x=7], x+y) \Rightarrow 10$ (labeled (+))
- Final Result:** $([y=3], (\text{fun } x = x + y) \ 7) \Rightarrow 10$ (labeled (fun))

Red arrows indicate the flow of data:

- An arrow from the x result of the first (var) frame to the x parameter in the (let) frame.
- An arrow from the y result of the second (var) frame to the y parameter in the (let) frame.
- An arrow from the eval_op result to the $x+y$ expression in the (+) frame.

Summary: Environment Models

In environment-based interpreter, values are drawn from an environment. This is more efficient than using substitution.

To implement nested, higher-order functions, create *closures*, a pair of code and environment

Different operational models appropriate at different levels of abstraction. High-level: substitution; Lower-level: environment.

Obligations:

- Understand how to read and explain operational rules
- Be able to follow rules and generate example derivations
- Understand the correspondence between code and rules