

An OCaml definition of OCaml evaluation, or,  
**Implementing OCaml in OCaml**  
(Part II)

COS 326

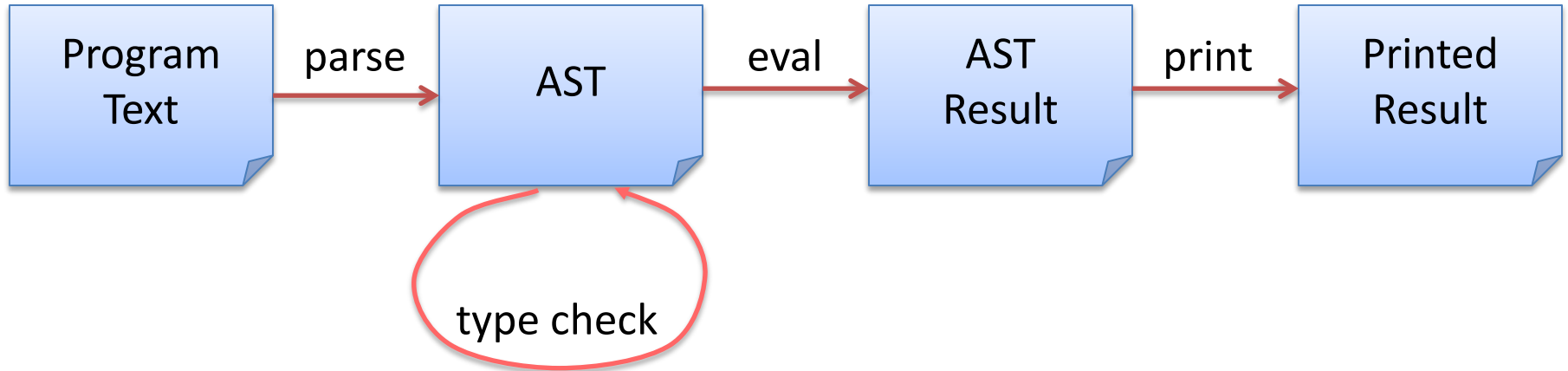
Andrew Appel

Princeton University

# Last Time

2

Implementing an interpreter:



Components:

- Evaluator for primitive operations
- Substitution
- Recursive evaluation function for expressions

# Last Time: Implementing Interpreters

```
type var = string
type op = Plus | Minus
type exp =
  | Int_e of int
  | Op_e   of exp * op * exp
  | Var_e of var
  | Let_e of var * exp * exp
```

Represent  
abstract  
syntax via  
data types

```
exception UnboundVariable of variable
```

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
```

Evaluate  
expressions

# A MATHEMATICAL DEFINITION\* OF OCAML EVALUATION

\* it's a partial definition and this is a big topic; for more, see COS 510

# From Code to Abstract Specification

5

OCaml code can give a language semantics

- **advantage**: it can be executed, so we can try it out
- **advantage**: it is amazingly concise
  - especially compared to what you would have written in Java
- **disadvantage**: it is a little ugly to operate over concrete ML datatypes like “`Op_e(e1,Plus,e2)`” as opposed to “`e1 + e2`”
- **big disadvantage**: When you use language X to define the semantics of language Y, you only get a precise definition of Y if you already fully understand the semantics of X. So, when you use OCaml to define the semantics of OCaml, you get a precise definition of OCaml only if you already know the precise definition of OCaml.

# From Code to Abstract Specification

6

PL researchers have developed their own standard notation for writing down how programs execute

- it has a mathematical “feel” that makes PL researchers feel special and gives us *goosebumps* inside
- it operates over abstract expression syntax like “ $e1 + e2$ ”
- it is useful to know this notation if you want to read specifications of programming language semantics
  - e.g.: Standard ML (of which OCaml is a descendent) has a formal definition given in this notation (and C, and Java; but not OCaml...)
  - e.g.: most papers in the conference POPL (ACM Principles of Prog. Lang.)

# Goal

7

Our goal is to explain how an expression  $e$  evaluates to a value  $v$ .

In other words, we want to define a mathematical *relation* between pairs of expressions and values.

# Formal Inference Rules

8

We define the “evaluates to” relation using a set of (inductive) rules that allow us to *prove* that a particular (expression, value) pair is part of the relation.

A rule looks like this:

$$\frac{\text{premise 1} \quad \text{premise 2} \quad \dots \quad \text{premise 3}}{\text{conclusion}}$$

You read a rule like this:

- “if *premise 1* can be proven and *premise 2* can be proven and ... and *premise n* can be proven then *conclusion* can be proven”

Some rules have no premises

- this means their conclusions are always true
- we call such rules “axioms” or “base cases”



# An example rule

9

As a rule:

$$\frac{e1 \Downarrow v1 \quad e2 \Downarrow v2 \quad \text{eval\_op}(v1, \text{op}, v2) == v'}{e1 \text{ op } e2 \Downarrow v'}$$

In English:

“If  $e1$  evaluates to  $v1$   
and  $e2$  evaluates to  $v2$   
and  $\text{eval\_op}(v1, \text{op}, v2)$  is equal to  $v'$   
then  
 $e1 \text{ op } e2$  evaluates to  $v'$ ”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Op_e(e1,op,e2) -> let v1 = eval e1 in  
                        let v2 = eval e2 in  
                        let v' = eval_op v1 op v2 in  
                        v'
```

# An example rule

10

As a rule:

$$\frac{i \in \mathbb{Z}}{i \Downarrow i}$$

asserts  $i$  is  
an integer

In English:

“If the expression is an integer value, **it evaluates to itself.**”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  ...
```



# An example rule concerning evaluation

12

As a rule:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}$$

typical “lambda” notation  
for a function with  
argument x, body e

In English:

“A function value evaluates to itself.”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  ...  
  | Fun_e (x,e) -> Fun_e (x,e)  
  ...
```

# An example rule concerning evaluation

13

As a rule:

$$\frac{e1 \Downarrow \lambda x.e \quad e2 \Downarrow v2 \quad e[v2/x] \Downarrow v}{e1 \ e2 \Downarrow v}$$

In English:

“if  $e1$  evaluates to a function with argument  $x$  and body  $e$   
and  $e2$  evaluates to a value  $v2$   
and  $e$  with  $v2$  substituted for  $x$  evaluates to  $v$   
then  $e1$  applied to  $e2$  evaluates to  $v$ ”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  ..  
  | FunCall_e (e1,e2) ->  
    (match eval e1 with  
     | Fun_e (x,e) -> eval (substitute (eval e2) x e)  
     | ...)  
  ...
```

# An example rule concerning evaluation

14

As a rule:

$$\frac{e1 \Downarrow \text{rec } f \ x = e \quad e2 \Downarrow v \quad e[(\text{rec } f \ x = e)/f][v/x] \Downarrow v2}{e1 \ e2 \Downarrow v2}$$

In English:

“uggh”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  ...  
  | (Rec_e (f,x,e)) as f_val ->  
    let v = eval e2 in  
    eval (substitute f_val f (substitute v x e))
```

# Comparison: Code vs. Rules

15

complete eval code:

complete set of rules:

```

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1
     | Fun_e (x,e) -> eval (Let_e (x,e2,e))
     | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
    (Rec_e (f,x,e)) as f_val ->
    let v = eval e2 in
    substitute f_val f (substitute v x e)
  
```

$$\begin{array}{c}
 \frac{i \in \mathbb{Z}}{i \Downarrow i} \\
 \\
 \frac{e1 \Downarrow v1 \quad e2 \Downarrow v2 \quad \text{eval\_op}(v1, \text{op}, v2) == v}{e1 \text{ op } e2 \Downarrow v} \\
 \\
 \frac{e1 \Downarrow v1 \quad e2 [v1/x] \Downarrow v2}{\text{let } x = e1 \text{ in } e2 \Downarrow v2} \\
 \\
 \frac{}{\lambda x. e \Downarrow \lambda x. e} \\
 \\
 \frac{e1 \Downarrow \lambda x. e \quad e2 \Downarrow v2 \quad e[v2/x] \Downarrow v}{e1 \text{ } e2 \Downarrow v} \\
 \\
 \frac{e1 \Downarrow \text{rec } f \text{ } x = e \quad e2 \Downarrow v2 \quad e[\text{rec } f \text{ } x = e/f][v2/x] \Downarrow v3}{e1 \text{ } e2 \Downarrow v3}
 \end{array}$$

*Almost* isomorphic:

- one rule per pattern-matching clause
- recursive call to eval whenever there is a  $\Downarrow$  premise in a rule
- what's the main difference?

# Comparison: Code vs. Rules

16

complete eval code:

complete set of rules:

```

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1
     | Fun_e (x,e) -> eval (Let_e (x,e2,e))
     | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
    (Rec_e (f,x,e)) as f_val ->
      let v = eval e2 in
      eval (substitute f_val f (substitute v x e))
  
```

$$\begin{array}{c}
 \frac{i \in \mathbb{Z}}{i \Downarrow i} \\
 \\
 \frac{e1 \Downarrow v1 \quad e2 \Downarrow v2 \quad \text{eval\_op}(v1, \text{op}, v2) == v}{e1 \text{ op } e2 \Downarrow v} \\
 \\
 \frac{e1 \Downarrow v1 \quad e2 [v1/x] \Downarrow v2}{\text{let } x = e1 \text{ in } e2 \Downarrow v2} \\
 \\
 \frac{}{\lambda x. e \Downarrow \lambda x. e} \\
 \\
 \frac{e1 \Downarrow \lambda x. e \quad e2 \Downarrow v2 \quad e[v2/x] \Downarrow v}{e1 \text{ e2 } \Downarrow v} \\
 \\
 \frac{e1 \Downarrow \text{rec } f \text{ x} = e \quad e2 \Downarrow v2 \quad e[\text{rec } f \text{ x} = e/f][v2/x] \Downarrow v3}{e1 \text{ e2 } \Downarrow v3}
 \end{array}$$

- There's no formal rule for handling free variables
- No rule for evaluating function calls when a non-function in the caller position
- In general, *no rule when further evaluation is impossible*
  - the rules express the *legal evaluations* and say nothing about what to do in error situations
  - the code handles the error situations by raising exceptions
  - type theorists prove that well-typed programs don't run into undefined cases



# Summary

17

We can reason about OCaml programs using a *substitution model*.

- integers, bools, strings, chars, and *functions* are values
- value rule: values evaluate to themselves
- let rule: “let  $x = e_1$  in  $e_2$ ” : substitute  $e_1$ ’s value for  $x$  into  $e_2$
- fun call rule: “(fun  $x \rightarrow e_2$ )  $e_1$ ”: substitute  $e_1$ ’s value for  $x$  into  $e_2$
- rec call rule: “(rec  $x = e_1$ )  $e_2$ ” : like fun call rule, but also substitute recursive function for name of function
  - To unwind: substitute (rec  $x = e_1$ ) for  $x$  in  $e_1$

We can make the evaluation model precise by building an interpreter and using that interpreter as a specification of the language semantics.

We can also specify the evaluation model using a set of *inference rules*

- more on this in COS 510

# Limitations

18

The substitution model is only a model.

- it does not accurately model all of OCaml's features
  - I/O, exceptions, mutation, concurrency, ...
  - we can build models of these things, but they aren't as simple.
  - even substitution is tricky to formalize!

# Limitations

19

The substitution model is only a model.

- it does not accurately model all of OCaml's features
  - I/O, exceptions, mutation, concurrency, ...
  - we can build models of these things, but they aren't as simple.
  - even substitution is tricky to formalize!

You can say that again!  
I got it wrong the first  
time I tried, in 1932.  
Fixed the bug by 1934,  
though.



Alonzo Church,  
1903-1995  
Princeton Professor,  
1929-1967

# Limitations

20

The substitution model is only a model.

- it does not accurately model all of OCaml's features
  - I/O, exceptions, mutation, concurrency, ...
  - we can build models of these things, but they aren't as simple.
  - even substitution is tricky to formalize!

It's useful for reasoning about correctness of algorithms.

- we can use it to formally prove that, for instance:
  - $\text{map } f (\text{map } g \text{ xs}) == \text{map } (\text{comp } f \text{ } g) \text{ xs}$
  - proof: by induction on the length of the list `xs`, using the definitions of the substitution model.
- we often model complicated systems (e.g., protocols) using a small functional language and substitution-based evaluation.

It is *not* useful for reasoning about execution time or space.

- more complex models needed there

# Reasoning about Nested Evaluation

21

Nested Evaluation, aka, “inlining” is a common compiler optimization.

It is also used in theorem provers to reason about equality of expressions.

# Reasoning about Nested Evaluation

22

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

# Reasoning about Nested Evaluation

23

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

```
g 10
```

# Reasoning about Nested Evaluation

24

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

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



# Reasoning about Nested Evaluation

25

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

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

# Reasoning about Nested Evaluation

26

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

```
g 10  
-->  
  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
```

# Reasoning about Nested Evaluation

27

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

```
g 10  
-->  
  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
```

# Reasoning about Nested Evaluation

28

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

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



Inline

# Reasoning about Nested Evaluation

29

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

Inline

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

Substitute

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

# Reasoning about Nested Evaluation

30

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

Inline

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

Substitute

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

Eval

```
let g x =  
  
  let x = 3 in  
  x + x
```

# Reasoning about Nested Evaluation

31

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



```
let g x =  
  let x = 3 in  
  x + x
```

# Reasoning about Nested Evaluation

32

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

Inline

```
let g x =  
  let x = 3 in  
  x + x
```

```
g 10 -->* 13
```



# Reasoning about Nested Evaluation

33

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

Inline

```
let g x =  
  let x = 3 in  
  x + x
```

```
g 10 -->* 13
```

```
g 10  
-->  
let x = 3 in  
x + x
```

# Reasoning about Nested Evaluation

34

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

Inline

```
let g x =  
  let x = 3 in  
  x + x
```

```
g 10 -->* 13
```

```
g 10  
-->  
let x = 3 in  
x + x  
-->  
3 + 3  
-->  
6
```

# Reasoning about Nested Evaluation

35

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



```
let g x =  
  let x = 3 in  
  x + x
```

```
g 10 -->* 13
```

Our goal in inlining is to make the computation more efficient but to get the same answer!

The transformation is incorrect.

```
g 10  
-->  
let x = 3 in  
x + x  
-->  
3 + 3  
-->  
6
```

# Reasoning about Nested Evaluation

36

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

Inline

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

Substitute **← WRONG!**

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

The x inside the function f was “captured” by the enclosing let. Substitution should be “capture-avoiding”

# Solution

37

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

Inline

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

alpha-convert  
to avoid capture

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

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

# Solution: More Generally

38

$$(\text{let } x = e_1 \text{ in } e_2) [e/y] = \text{let } x = e_1' \text{ in } e_2'$$

where

$$e_1' = e_1 [e/y]$$

$$e_2' = e_2 \quad \text{if } y = x$$

$$e_2' = e_2 [e/y] \quad \text{if the free variables of } e \text{ do not include } x \\ \text{and if } y \neq x$$

and otherwise, choose an unused variable  $z$  and  
alpha-convert  $\text{let } x = \dots \text{ in } \dots$  to  $\text{let } z = \dots \text{ in } \dots$

# Solution: More Generally

39

$$(\text{let } x = e_1 \text{ in } e_2) [e/y] = \text{let } x = e_1' \text{ in } e_2'$$

where

$$e_1' = e_1 [e/y]$$

$$e_2' = e_2 \quad \text{if } y = x$$

$$e_2' = e_2 [e/y] \quad \text{if the free variables of } e \text{ do not include } x \\ \text{and if } y \neq x$$

and otherwise, choose an unused variable  $z$  and  
alpha-convert  $\text{let } x = \dots \text{ in } \dots$  to  $\text{let } z = \dots \text{ in } \dots$

# **ASSIGNMENT #4**



## Part 1: Build your own interpreter

- More features: booleans, pairs, lists, match
- Different model: environment-based vs substitution-based
  - The abstract syntax tree `Fun_e(,)` *is no longer a value*
    - *a Fun\_e is not a result of a computation*
  - There is one more computation step to do:
    - creation of a *closure* from a Fun\_e expression

## Part 2: Prove facts about programs using equational reasoning

- we have already seen a bit of equational reasoning
  - if  $e1 \rightarrow e2$  then  $e1 == e2$
- more in precept and next week

# **AN ENVIRONMENT MODEL FOR PROGRAM EXECUTION**

# Substitution

43

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

44

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

45

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

46

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

47

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

48

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

49

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)
```

# Substitution

50

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)
```

# Substitution

51

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

The substitution model of evaluation is *just a model*. It says that we generate new code at each step of a computation. We don't do that in reality. Too expensive!

The substitution model is good for reasoning about the input-output behavior of a function but doesn't tell us much about the resources used along the way.

Efficient interpreters use *environments* rather than substitution.

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

# Environment Models

53

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

# Execution with Environment Models

54

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

55

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

56

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

57

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

58

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

59

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

60

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

61

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

# Another Example

62

```
([],  
 (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

63

```
([],  
 (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

64

```
([],  
 (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 )
```

-->

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



# Another Example

65

```
([],  
 (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 )
```

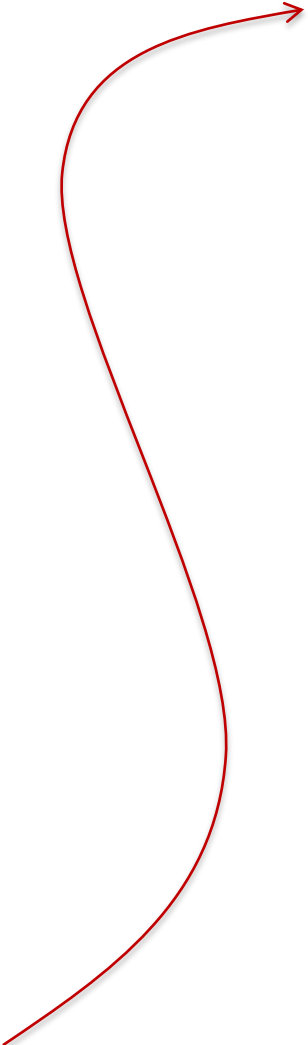
-->

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

-->

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

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



# Another Example

66

```
([],  
 (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 )
```

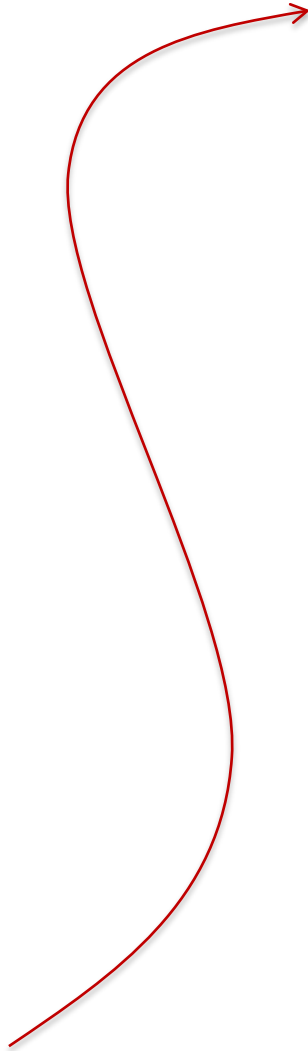
-->

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

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

-->

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



# Another Example

67

```
([],  
 (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 )
```

-->

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

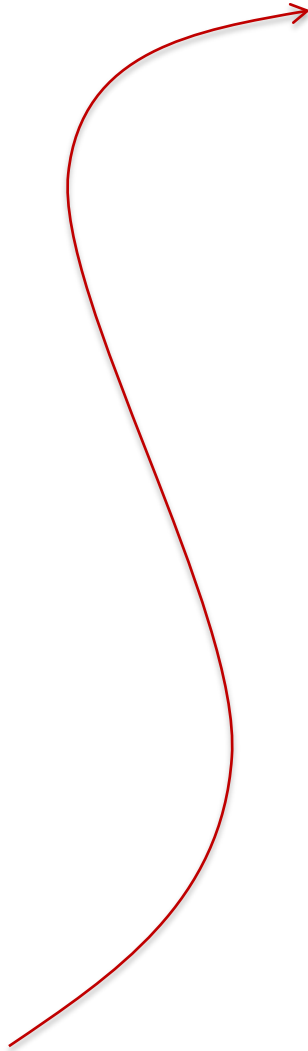
```
([x -> 3; f -> fun y -> y + x],  
 (fun y -> y + x) x )
```

-->

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

-->

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



# Another Example

68

```
([],  
 (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 )
```

-->

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

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

-->

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

-->

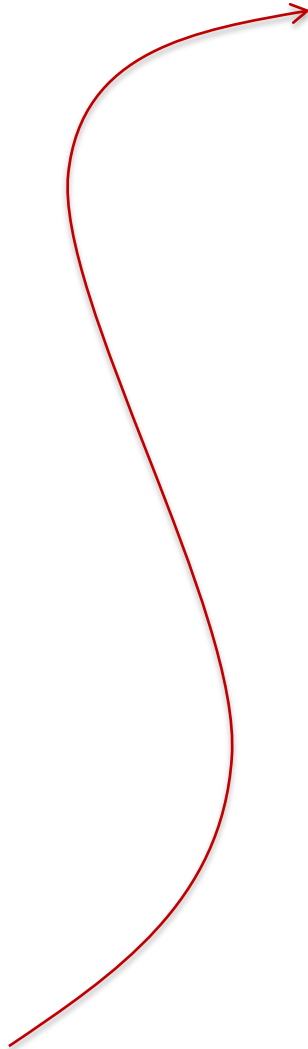
```
([x -> 3; f -> fun y -> y + x; y -> 3],  
 y + x )
```

-->

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

-->

```
([x -> 3; f -> fun y -> y + x; y -> 3],  
 6 )
```



# Recall our Problem with Inlining/Substitution

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

```
g 10 -->* 13
```

Incorrect  
Inlining

```
let g x =
  let x = 3 in
  x + x
```

```
g 10 -->* 6
```

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

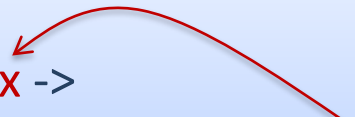
Incorrect  
Execution

```
(([], ...) -->*
 ([...], 6)
```

# Another Example

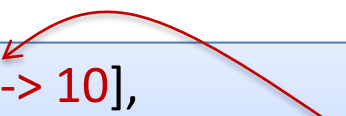
70

```
([],  
(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 )
```

-->

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



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

-->

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

-->

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

-->

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

-->

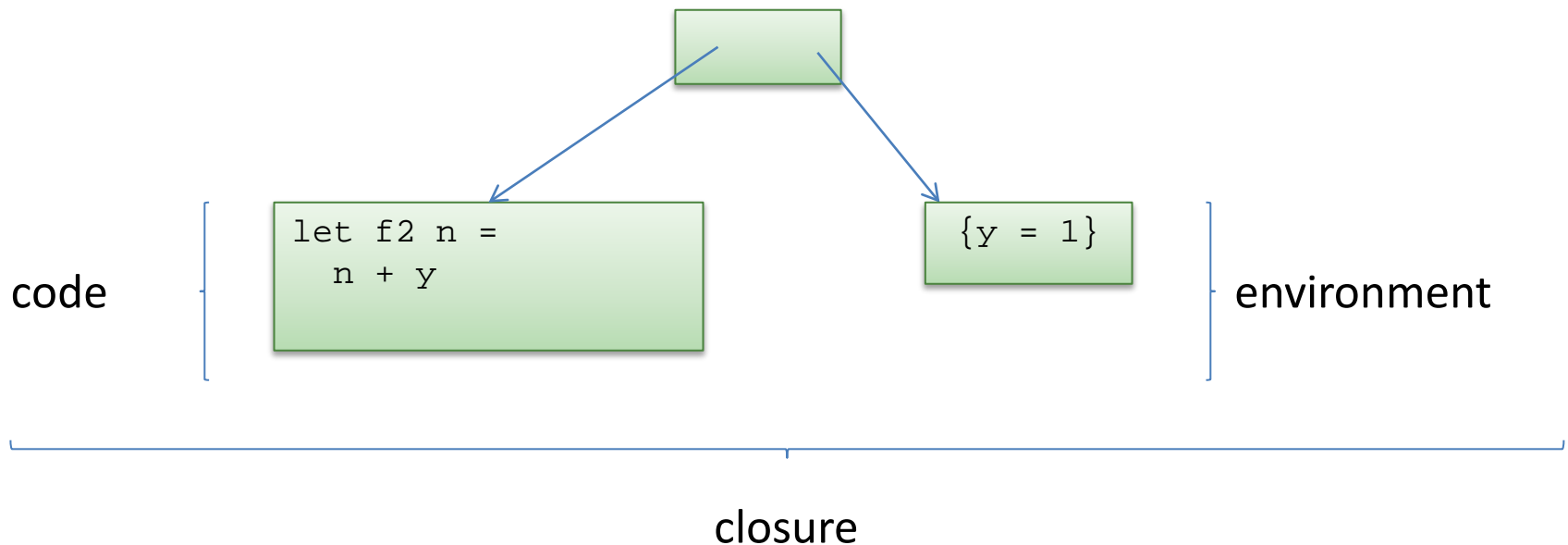
```
([x -> 3; f -> fun y -> y + x; y -> 3],  
6 )
```

# Solution

71

Functions must carry with them the appropriate environment

A *closure* is a pair of code and environment

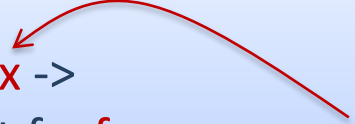


In the environment model, *function definitions* evaluate to *function closures*

# Another Example

72

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

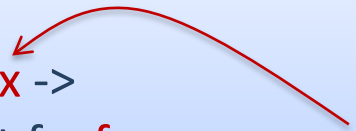




# Another Example

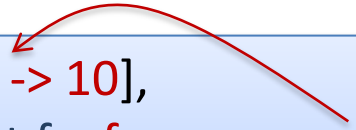
73

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



-->

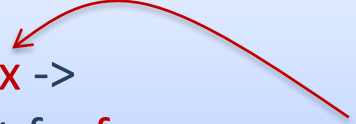
```
([x -> 10],  
let f = fun y -> y + x in  
let x = 3 in  
f x )
```



# Another Example

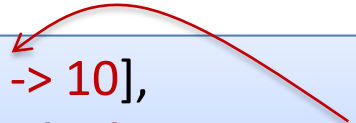
74

```
([],  
(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 )
```



-->

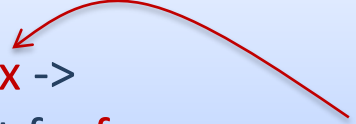
```
([x -> 10; f -> closure [x->10] y = y + x],  
  let x = 3 in  
  f x )
```



# Another Example

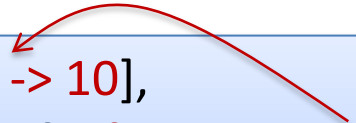
75

```
([],  
(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 )
```



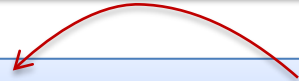
-->

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



-->

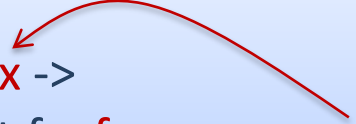
```
([x -> 3; f -> closure [x->10] fun y -> y = y + x],,  
  f x )
```



# Another Example

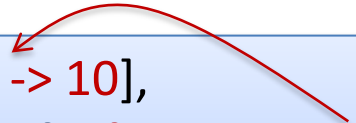
76

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




-->

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




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



# Another Example

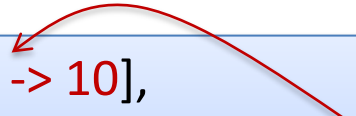
77

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



# Another Example

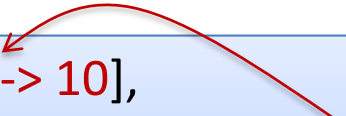
78

```
([],  
(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 -> 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

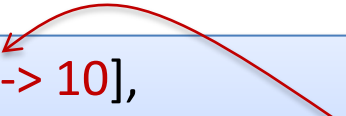
79

```
([],  
(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 -> 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 )
```

# Summary: Environment Models

80

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

To implement nested, higher-order functions, pair functions with the environment in play when the function is defined.

Pairs of function code & environment are called *closures*.

You have two weeks for assignment #4

- Recommendation: Don't wait until next week to start!