An OCaml definition of OCaml evaluation, or,

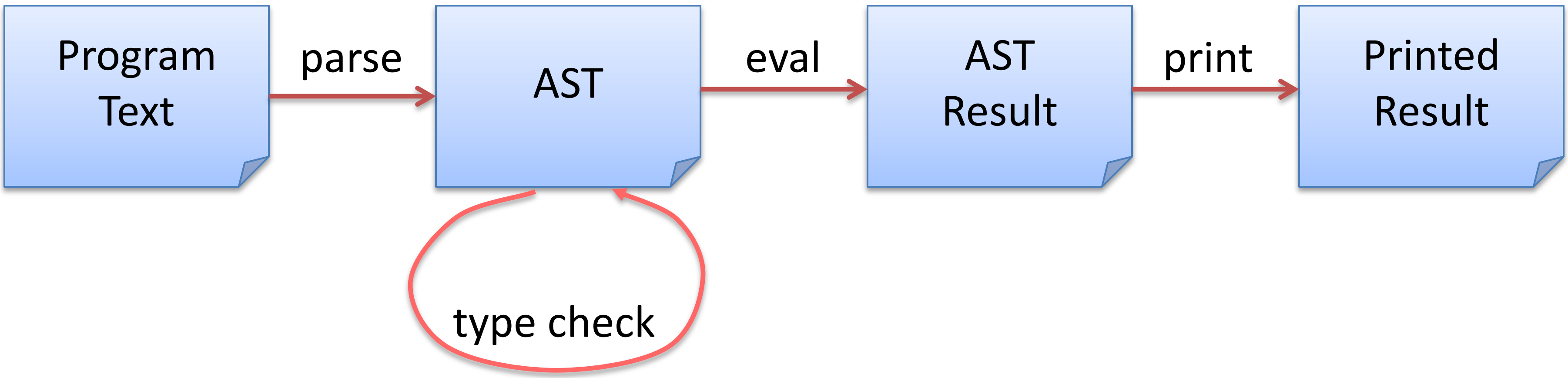
Implementing OCaml in OCaml

(Part II)

COS 326
Andrew Appel
Princeton University

Last Time

Implementing an interpreter:



Components:

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

Last Time: Implementing Interpreters

Represent abstract syntax via data types

```
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
```

Evaluate expressions

```
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)
```

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

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

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.)
  - Programming languages that have been formally defined this way:
    - Java, Javascript, Rust, C, ML, . . .

Goal

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

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:

$$\frac{\text{premise 1} \qquad \text{premise 2} \qquad \ldots \qquad \text{premise 3}}{\text{conclusion}}$$

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"

An example rule

As a rule:

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

In English:

"If e1 evaluates to v1
and e2 evaluates to v2
and eval_op (v1, op, v2) is equal to v'
then
e1 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

As a rule:

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

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

As a rule:

$$\frac{e1 \Downarrow v1 \qquad e2\,[v1/x] \Downarrow v2}{\text{let } x = e1 \text{ in } e2 \Downarrow v2}$$

In English:

> "If e1 evaluates to v1 (which is a *value*)
> and e2 with v1 substituted for x evaluates to v2
> then let x=e1 in e2 evaluates to v2."

In code:

```
let rec eval (e:exp) : exp =
  match e with
 | Let_e(x,e1,e2) -> let v1 = eval e1 in
                     eval (substitute v1 x e2)
 ...
```

An example rule concerning evaluation

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

As a rule:

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

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

As a rule:

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

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

complete eval code:

```
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)
```

complete set of rules:

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

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

$$\frac{e1 \Downarrow v1 \qquad e2\ [v1/x] \Downarrow v2}{\text{let } x = e1 \text{ in } e2 \Downarrow v2}$$

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

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

$$\frac{e1 \Downarrow \text{rec } f\ x = e \qquad e2 \Downarrow v2 \qquad e[\text{rec } f\ x = e/f][v2/x] \Downarrow v3}{e1\ e2 \Downarrow v3}$$

Almost isomorphic:

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

Comparison: Code vs. Rules

complete eval code:

```
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))
```

complete set of rules:

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

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

$$\frac{e1 \Downarrow v1 \qquad e2\ [v1/x] \Downarrow v2}{let\ x = e1\ in\ e2 \Downarrow v2}$$

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

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

$$\frac{e1 \Downarrow rec\ f\ x = e \qquad e2 \Downarrow v2 \qquad e[rec\ f\ x = e/f][v2/x] \Downarrow v3}{e1\ e2 \Downarrow v3}$$

- 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

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 = e1 in e2" : substitute e1's value for x into e2
- fun call rule: "(fun x -> e2) e1": substitute e1's value for x into e2
- rec call rule: "(rec x = e1) e2" : like fun call rule, but also substitute recursive function for name of function
  - To unwind: substitute (rec x = e1) for x in e1

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

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

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

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:
  - map f (map g xs) == map (comp f g) 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

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


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

Reasoning about Nested Evaluation


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

```
g 10
```

Reasoning about Nested Evaluation


```
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


```
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


```
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


```
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

```
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 ]
```

Reasoning about Nested Evaluation

```
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

```
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


```
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
```

Reasoning about Nested Evaluation

```
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


```
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

```
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

```
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
```

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

The transformation is incorrect.

Reasoning about Nested Evaluation


```
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


```
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

(let x = e1 in e2) [e/y] = let x = e1' in e2'

where

e1' = e1 [e/y]

e2' = e2 if y = x

e2' = e2 [e/y] if the free variables of e do not include x
and if y != x

and otherwise, choose an unused variable z and
alpha-convert let x = ... in ... to let z = ... in ...

Solution: More Generally

(let x = e1 in e2) [e/y] = let x = e1’ in e2’

where

e1’ = e1 [e/y]

e2’ = e2 if y = x

e2’ = e2 [e/y] if the free variables of e do not include x
and if y != x

and otherwise, choose an unused variable z and
alpha-convert let x = ... in ... to let z = ... in ...

ASSIGNMENT #4

Two Parts

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 --> e2 then e1 == e2
- more in precept and next week

AN ENVIRONMENT MODEL FOR PROGRAM EXECUTION

Substitution

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

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

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

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

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

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

Substitution

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

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

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

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

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

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 --> e2

Execution with Environment Models

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 --> 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) --> (env2, e2)

Execution with Environment Models

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

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

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

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

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


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

Another Example


```
([],
 (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


```
([],
 (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


```
([],
 (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


```
([],
 (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

```
([],
 (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


```
([],
 (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


```
([],
 (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
```

Incorrect Inlining

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

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

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

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

Incorrect Execution

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

Another Example

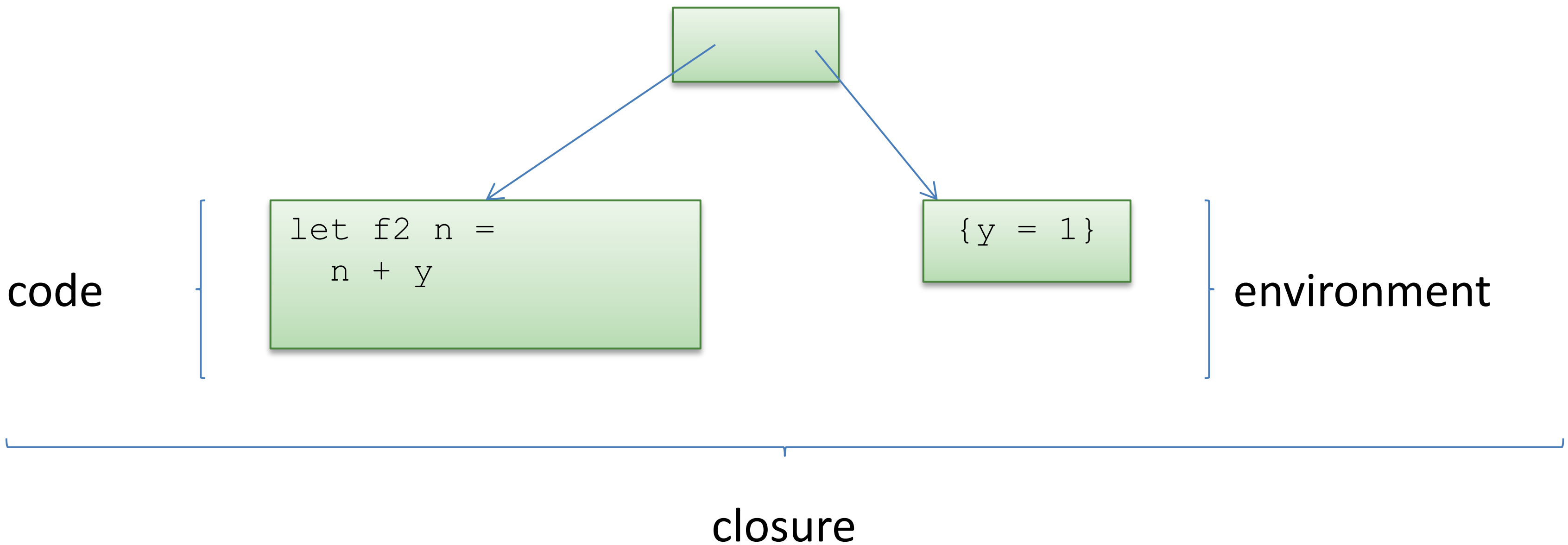

```
([],
 (fun x ->
    let f = fun y -> y + x in
    let x = 3 in
    f x) 10 )
-->
([x -> 10],
 let f = fun y -> y + x in
 let x = 3 in
 f x )
-->
([x -> 10; f -> fun y -> y + x],
 let x = 3 in
 f x )
-->
([x -> 3; f -> fun y -> y + x],
 f x )
```

```
([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

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


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

Another Example


```
([],
 (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


```
([],
 (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


```
([],
 (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


```
([],
 (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


```
([],
 (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


```
([],
 (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

```
([],
 (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

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!