Assignment #4

COS 326

David Walker

Princeton University

Assignment Overview

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

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)
```

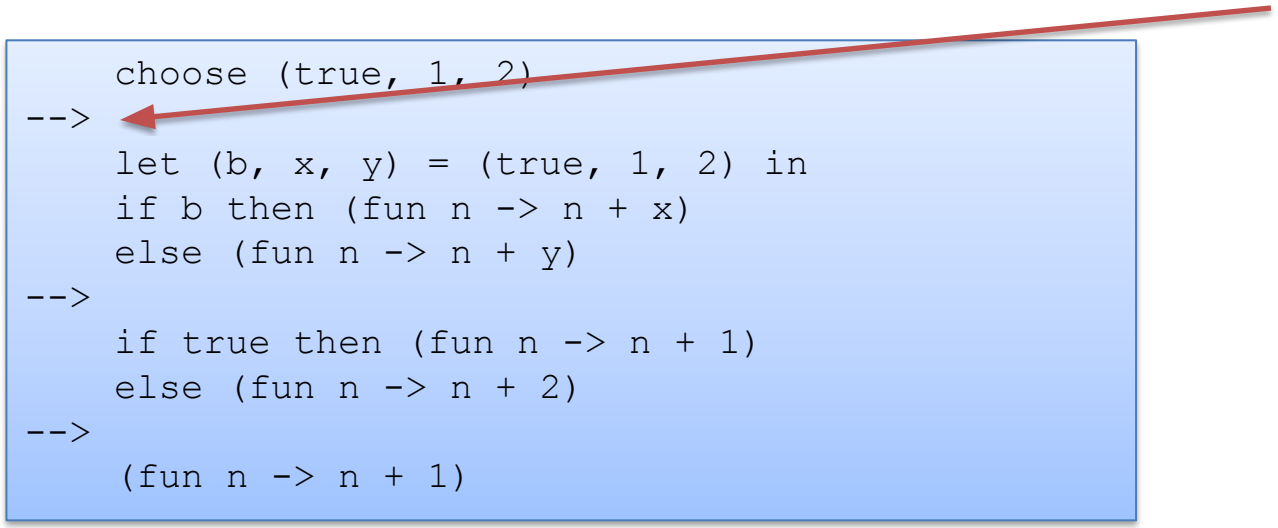
Its execution behavior according to the substitution model:

```
    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

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 )
```

-->

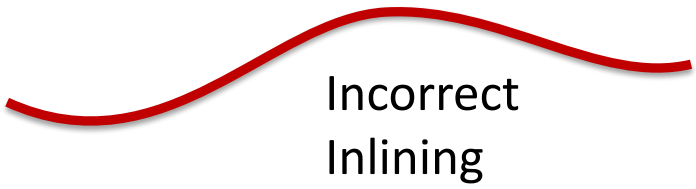
```
([x -> 3; f -> fun y -> y + x; y -> 3],
  3 + 3 )
```

-->

```
([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
```




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

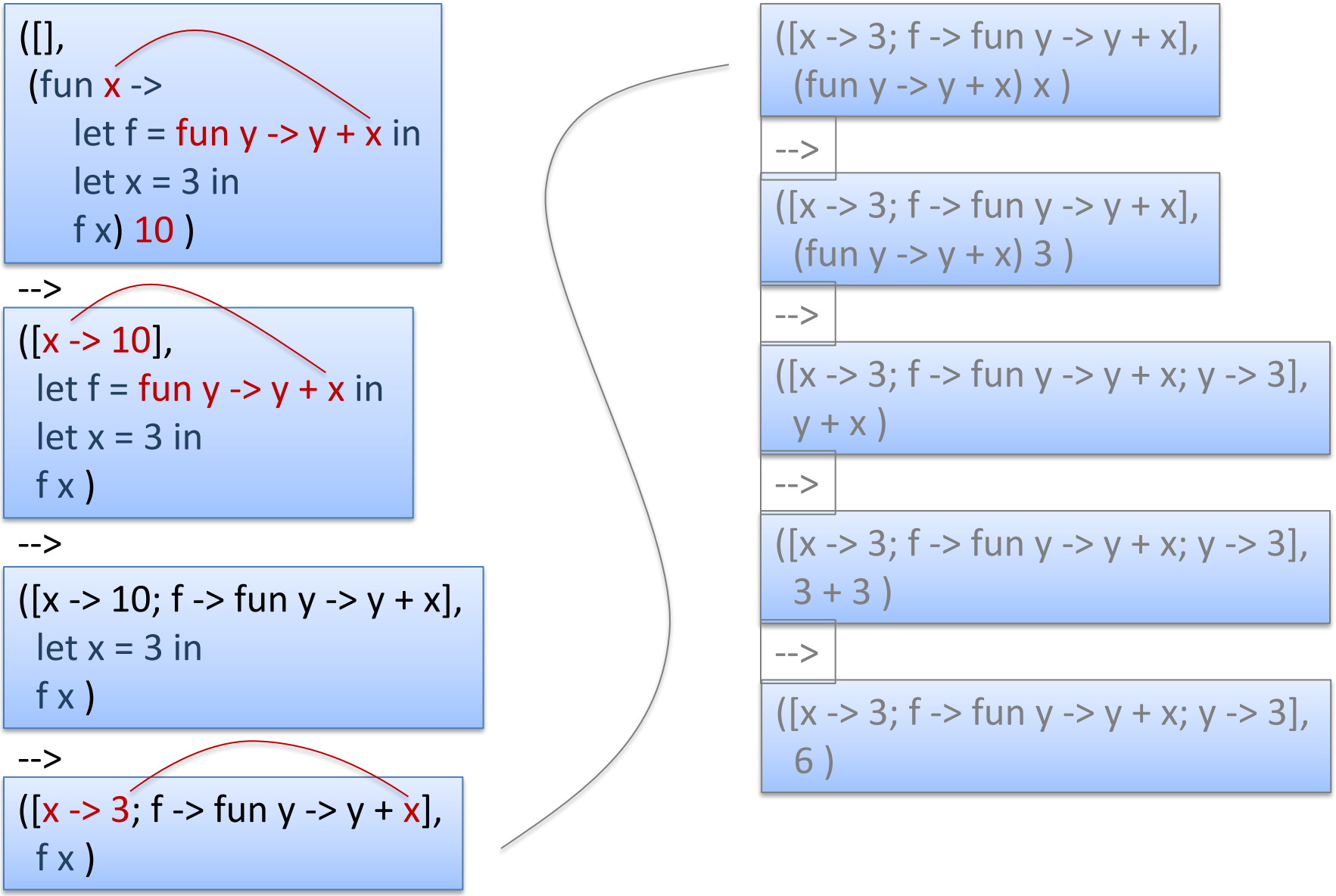
```
g 10 -->* 13
```

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

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


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

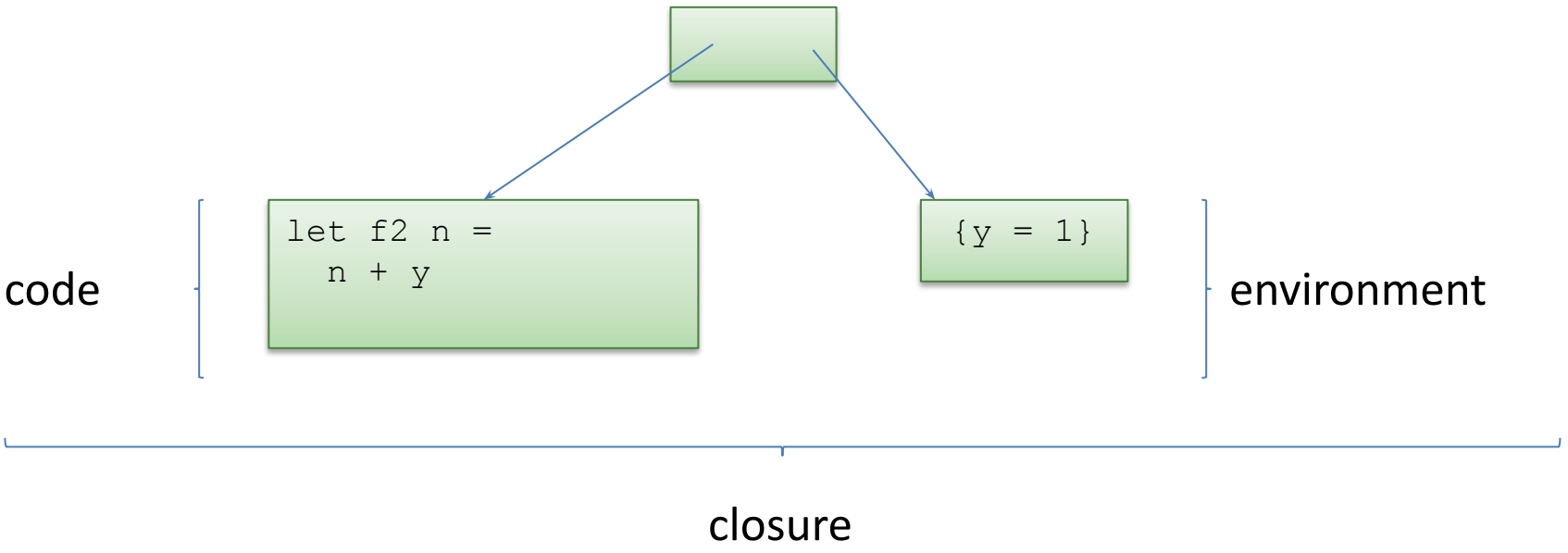
Another Example



Solution

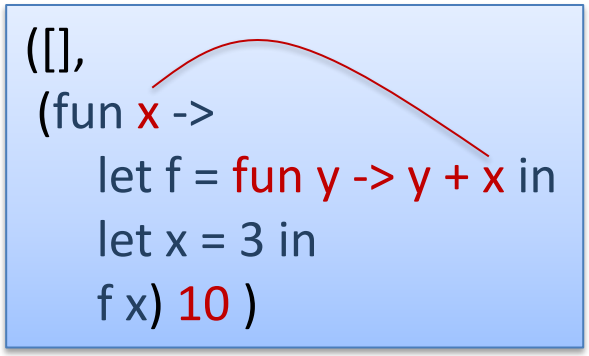
Functions must carry with them the appropriate environment

A *closure* is a pair of code and environment



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

Another Example



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 )
```

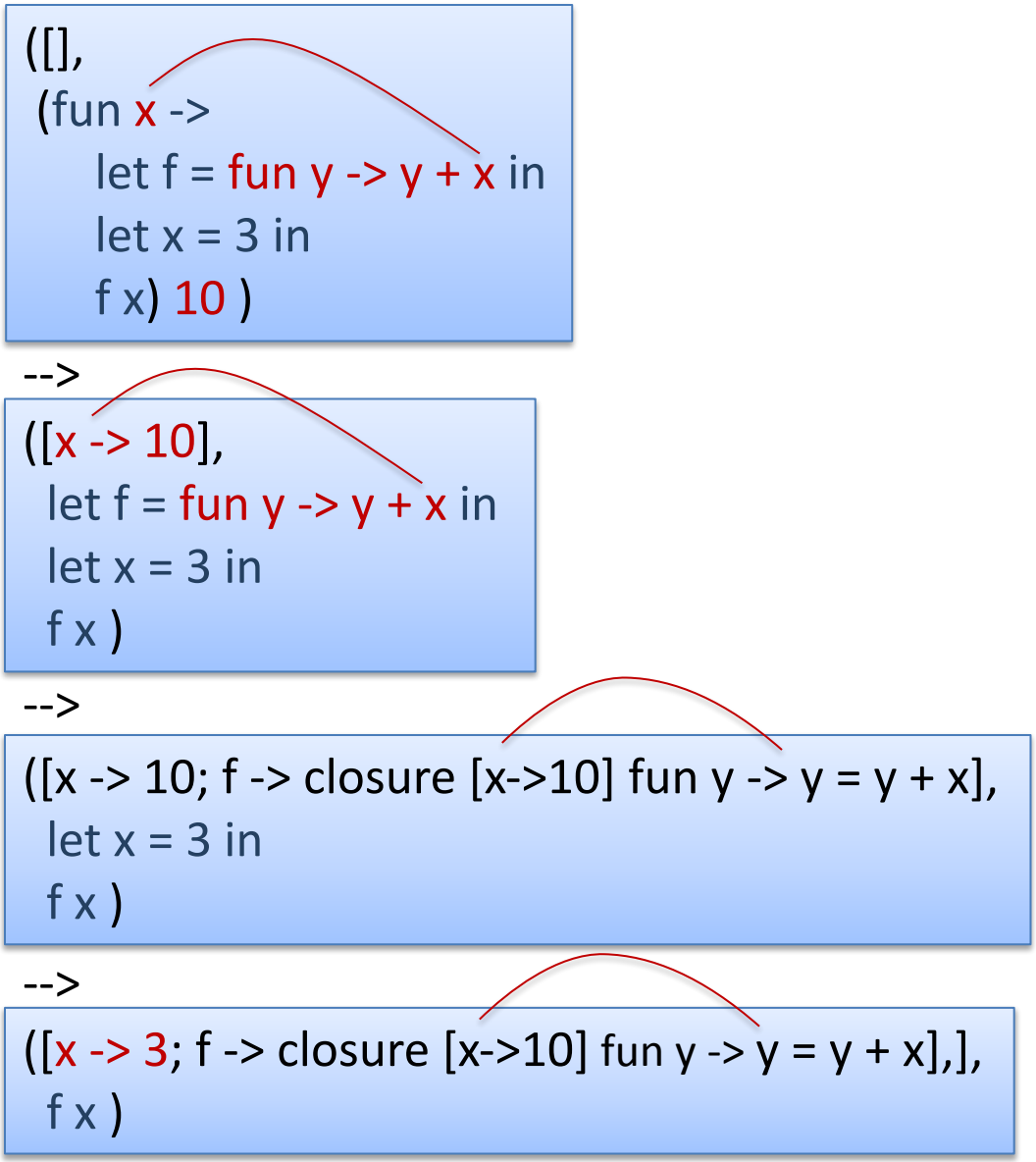
-->

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

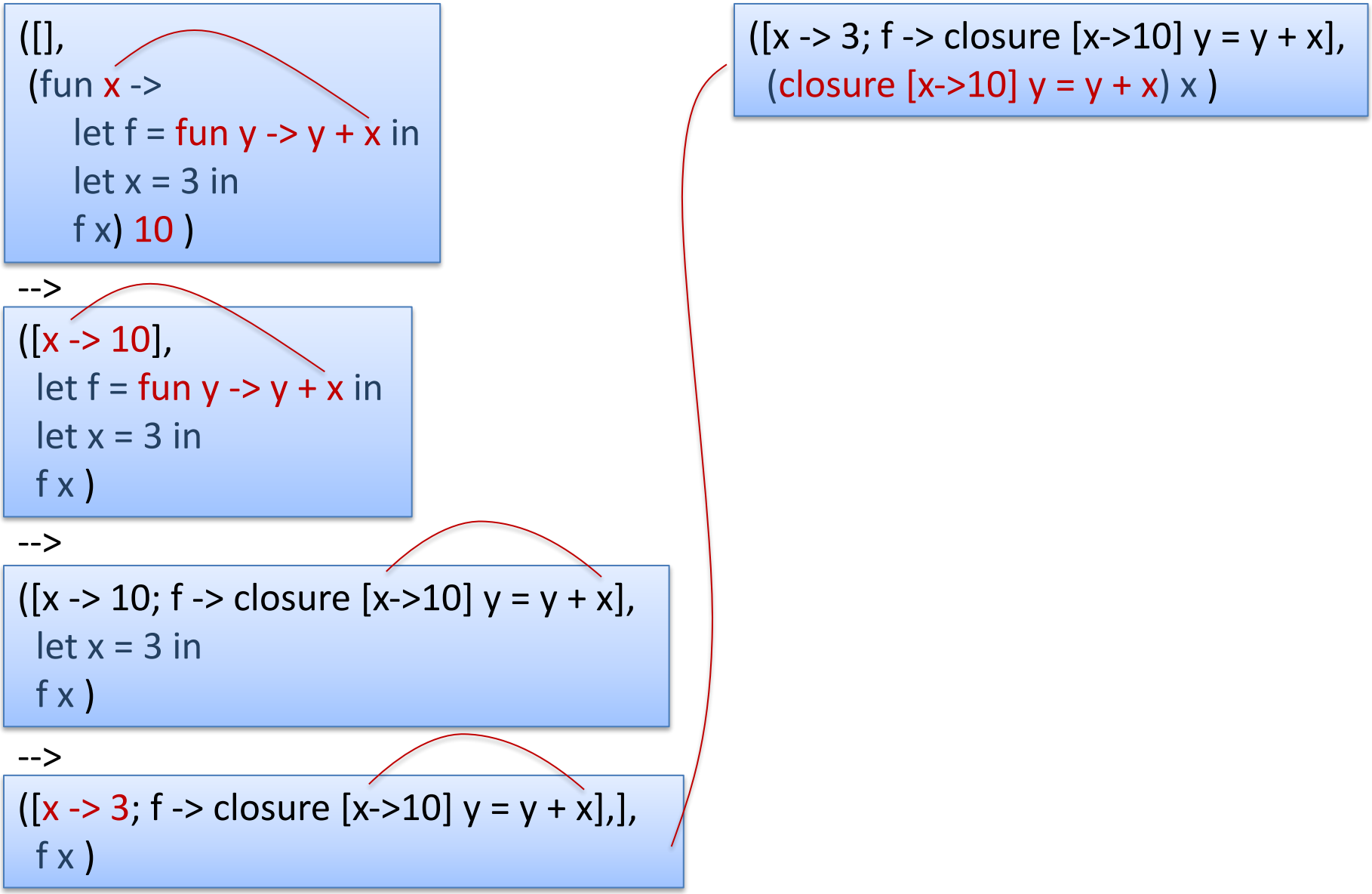
-->

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

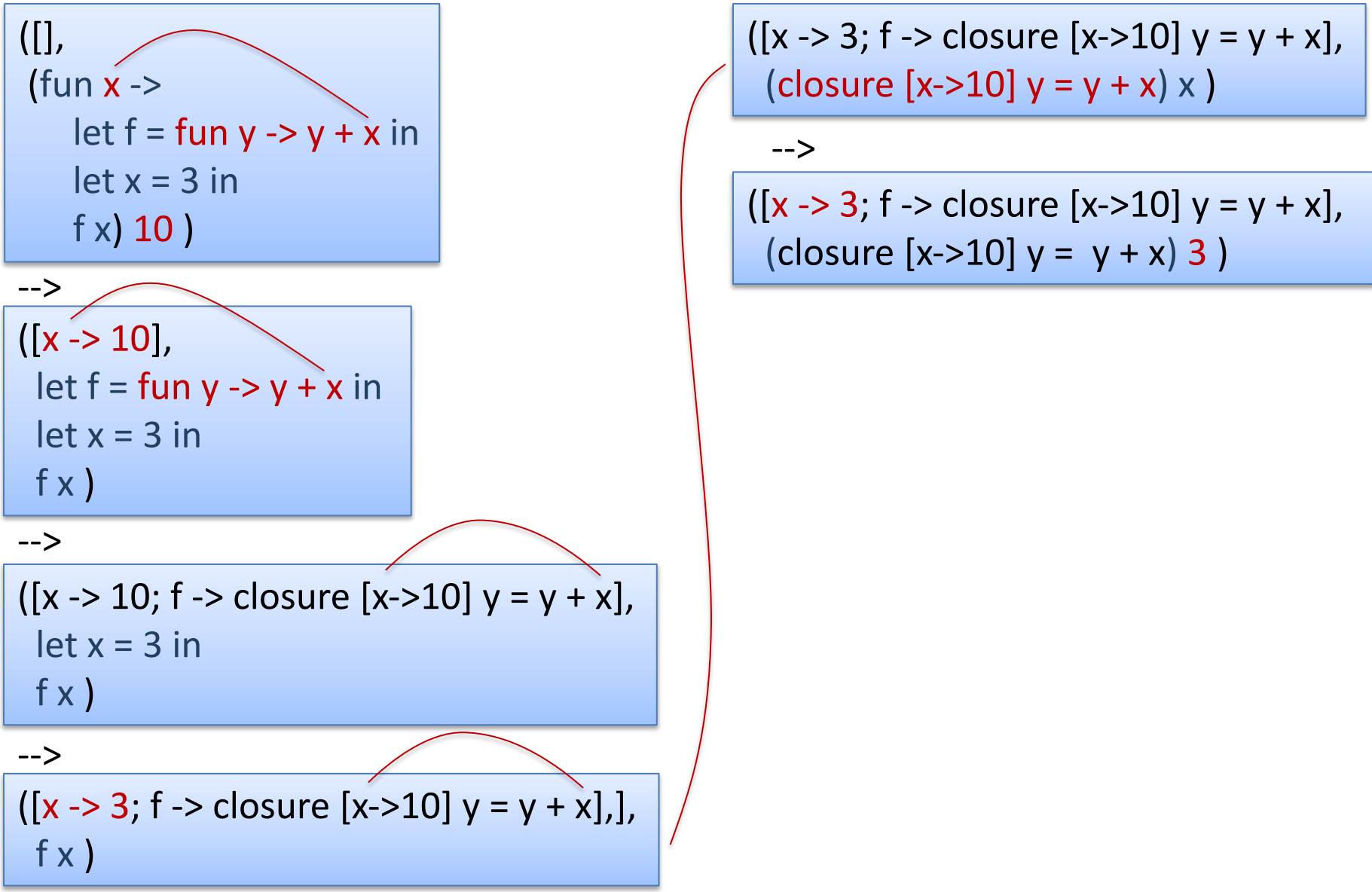
Another Example



Another Example



Another Example



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 )
```

-->

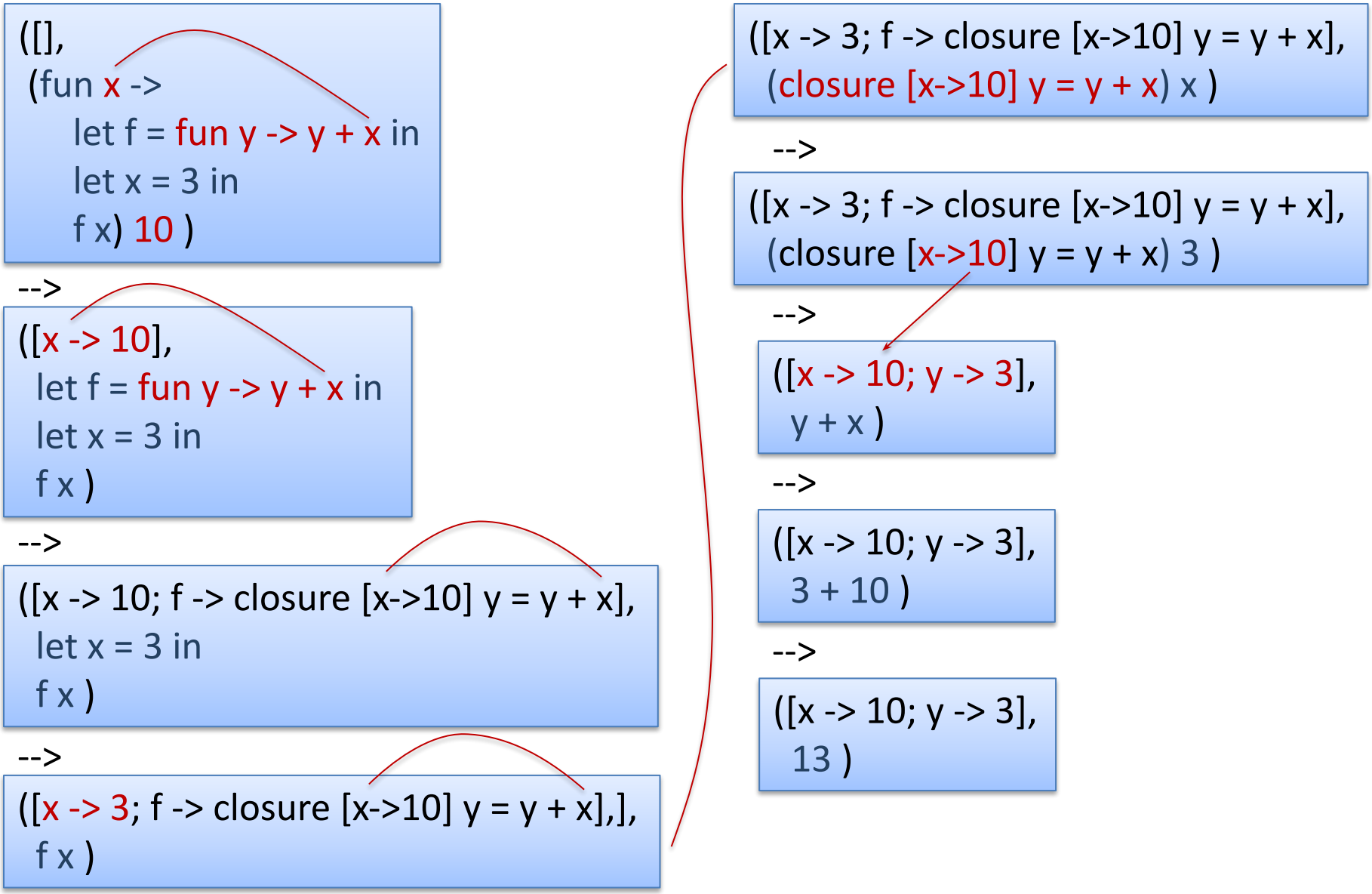
```
([x -> 3; f -> closure [x->10] y = y + x],
  (closure [x->10] y = y + x) 3 )
```

-->

```
([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



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*.