CS443: Compiler Construction

Lecture 11: Closure Conversion and Nested Environments

Stefan Muller

Based on material from Steve Chong, Steve Zdancewic, and Greg Morrisett

Do closure conversion and hoisting in one pass

```
compile_exp : ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * c_func list
```

Any extra statements needed to compute the value of the expression

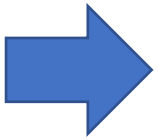
A C expression that computes the value of the ML expression (assuming the statements have run)

Function bodies collected while compiling the expression

Do closure conversion and hoisting in one pass

```
compile_exp : ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * cfunc list
```

```
let x = 1 in
let inc = fun y -> x + y in
inc 2
```



```
int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos(inc1__body, env);
env = __extend_env(env, “inc”, inc_1);
int temp_1 = inc_1.clos_fun(inc_1.clos_env, 2)
```

```
temp_1
```

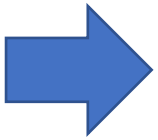
```
[int inc1__body(int y, env env) {
   env = __extend_env(env, “y”, y);
   return __lookup(env, “x”) + y;
 }]
```

Do closure conversion and hoisting in one pass

```
compile_exp : ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * cfunc list
```

```
int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos(inc1__body, env);
env = __extend_env(env, “inc”, inc_1);
```

```
let x = 1 in
let inc = fun y -> x + y in
inc 2
```



```
inc_1.clos_fun(inc_1.clos_env, 2)
```

```
[int inc1__body(int y, env env) {
   env = __extend_env(env, “y”, y);
   return __lookup(env, “x”) + y;
 }
```

A lambda just evaluates to a closure

```
compile_exp(fun (x: int) : int -> e)
```

```
closure temp_1 = __mk_clos(__fun, env);
```

```
temp_1
```

```
[int __fun(int x, env env) {
    env = __extend_env(env, “x”, x);
    … compilation of e …
 };
 plus any functions nested in e]
```

Funcception

```
let add = fun x -> fun y -> x + y

closure __fun1(int x, env env) {
  env = __extend_env(env, “x”, x);
  return __mk_clos(__fun2, env);
}

int __fun2(int y, env env) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + __lookup(env, “y”);
}
env = __extend_env(env, “add”, __mk_clos(__fun1, env));
```

Applications evaluate the two expressions, then apply

```
compile_exp(e1 e2)
```

```
statements for e1
statements for e2
```

```
exp_e1.clos_fun(exp_e2, exp_e1.clos_env)
```

```
(cfuncs from e1) @ (cfuncs from e2)
```

Let is pretty similar

```
compile_exp(let x = e1 in e2)
```

```
statements for e1
statements for e2
```

```

```

(cfuncs from e1) @ (cfuncs from e2)

Where do we push/pop?

Let is pretty similar

```
compile_exp(let x = e1 in e2)
```

```
statements for e1
env = __extend_env(env, “x”, e1_exp);
statements for e2
temp_1 = e2_exp;
env = __pop_env(env);
```

temp_1

(cfuncs from e1) @ (cfuncs from e2)

Think about what goes wrong if we did this

```
compile_exp(let x = e1 in e2)
```

```
statements for e1
env = __extend_env(env, “x”, e1_exp);
statements for e2
env = __pop_env(env);
```

```
e2_exp
```

```
(cfuncs from e1) @ (cfuncs from e2)
```

I guess we need to compile other things too

```
compile_exp(if e1 then (e2: int) else (e3: int))
```

```
statements for e1
int temp_1
if (exp_e1) { statements for e2; temp_1 = exp_e2; }
else { statements for e3; temp_1 = exp_e3; }
```

```
temp_1
```

```
(cfuncs from e1) @ (cfuncs from e2) @ (cfuncs from e3)
```

How to *actually* represent closures

```
struct __clos {
  env clos_env;
  int() clos_fun();
};
```

Stand-in function pointer type since C doesn't have parametric polymorphism. We'll need to cast it to whatever

Note that that means we need to cast when we apply

```
compile_exp(e1 e2)
```

```
statements for e1
statements for e2
```

```
((ret_ty(e2_ty))exp_e1.clos_fun)
  (exp_e2, exp_e1.clos_env)
```

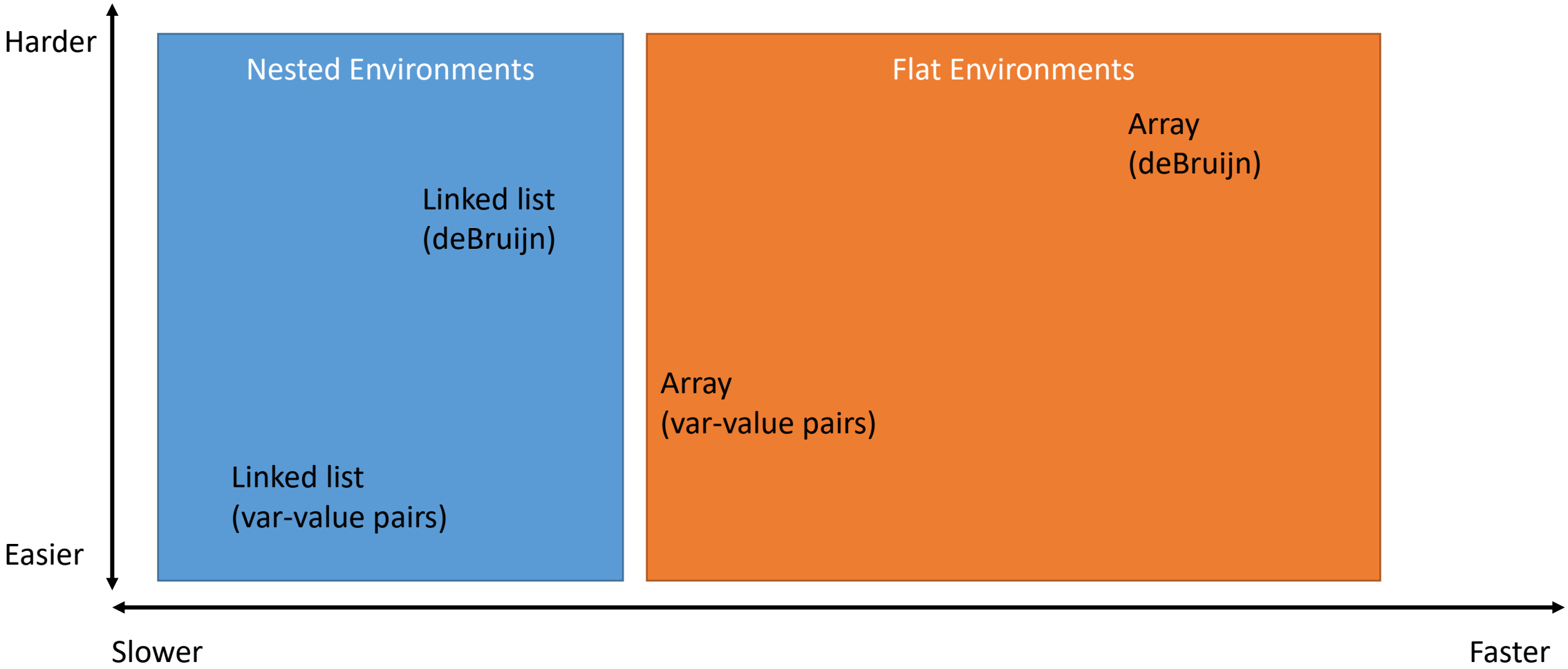
```
(closures from e1) @ (closures from e2)
```

How do we know what the return type and argument type are? X.einfo

How to represent environments

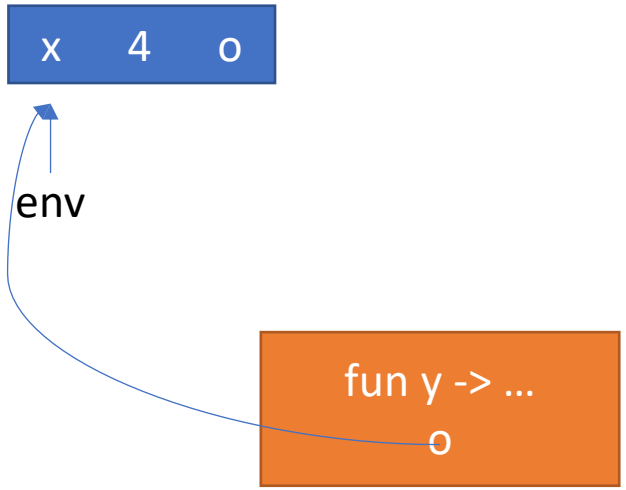
- Considerations:
  - Optimization: we don’t need to store all variables in the environment, just those that might “escape” (be used in nested functions)
  - Data structure: lookup should be fast (asymptotic and constant factors)

Data Structures for Environments



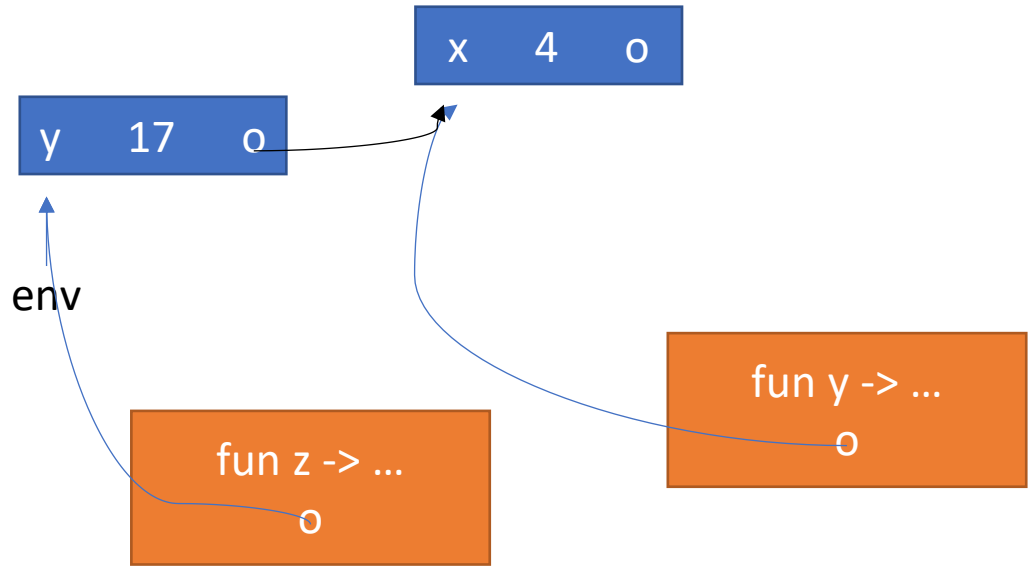
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



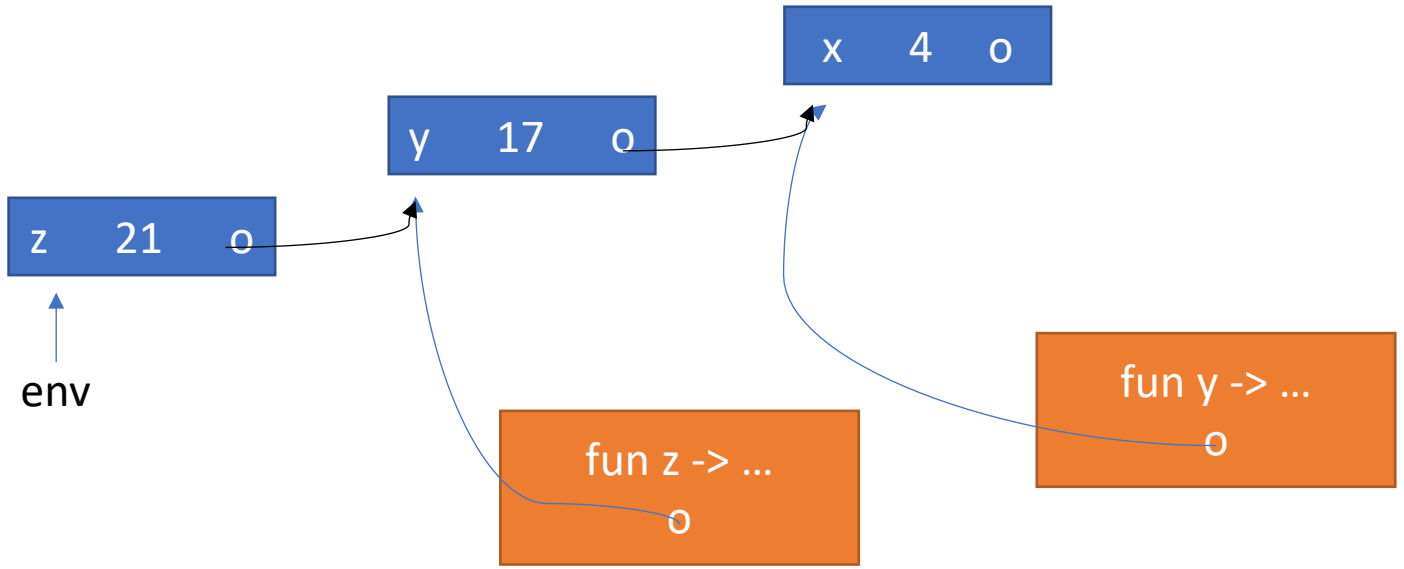
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



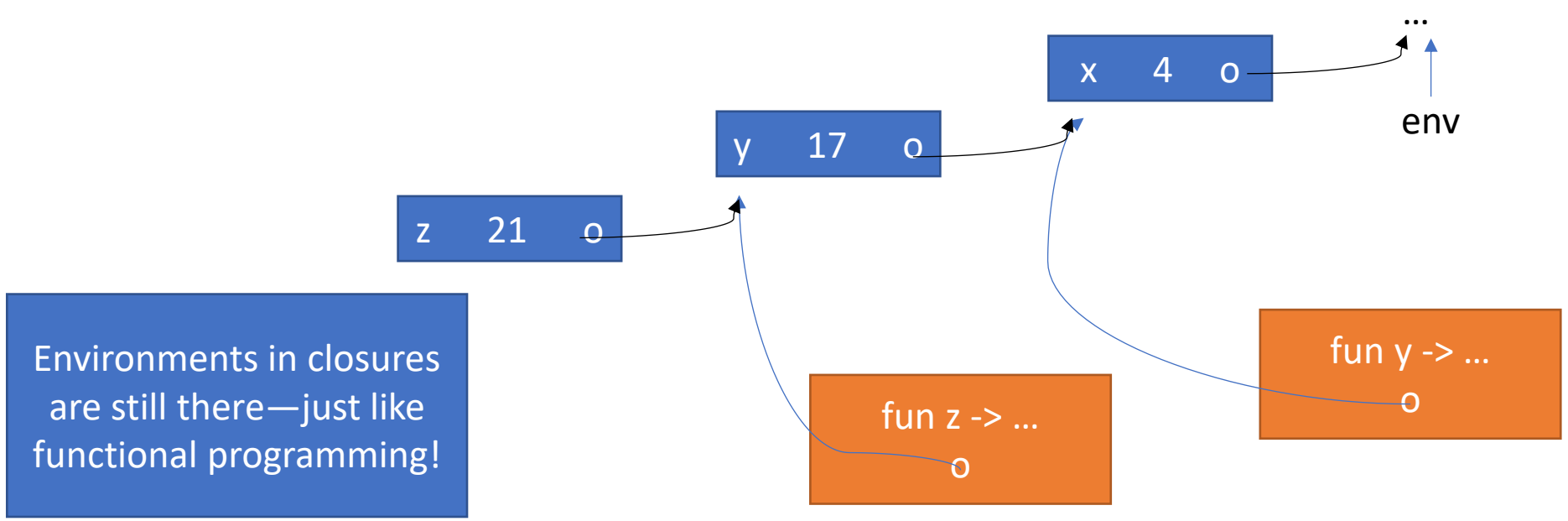
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Extend and Lookup for Nested Envs

```
__extend_env(env, var, val):
  env new_node = new env(var, val, env)
  return new_node

__lookup(env, var):
  while(env.var != var && env.next != NULL):
    env = env.next
  return env.val
```

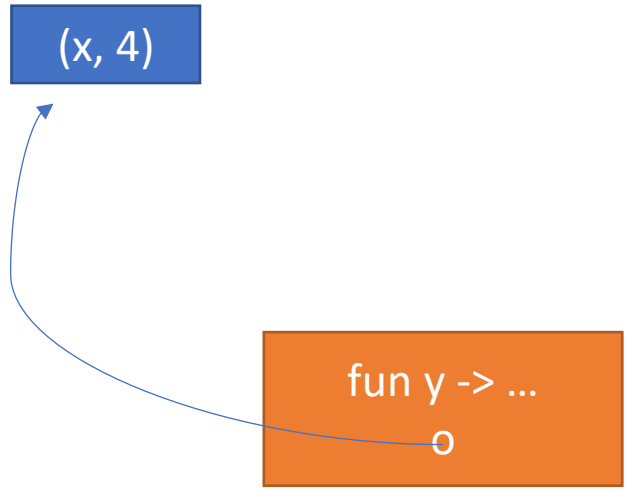
For recursive functions, can just make the closure and “backpatch” it later

```
let rec fact n = if n <= 1 then n else n * (fact (n – 1))


int fact__body(env env, int n) {
  env = __extend_env(env, “n”, n);
  …
}
closure fact_clos = __mk_clos(fact__body, env));
env = __extend_env(env, “fact”, fact_clos);
fact_clos.clos_env = env;
```

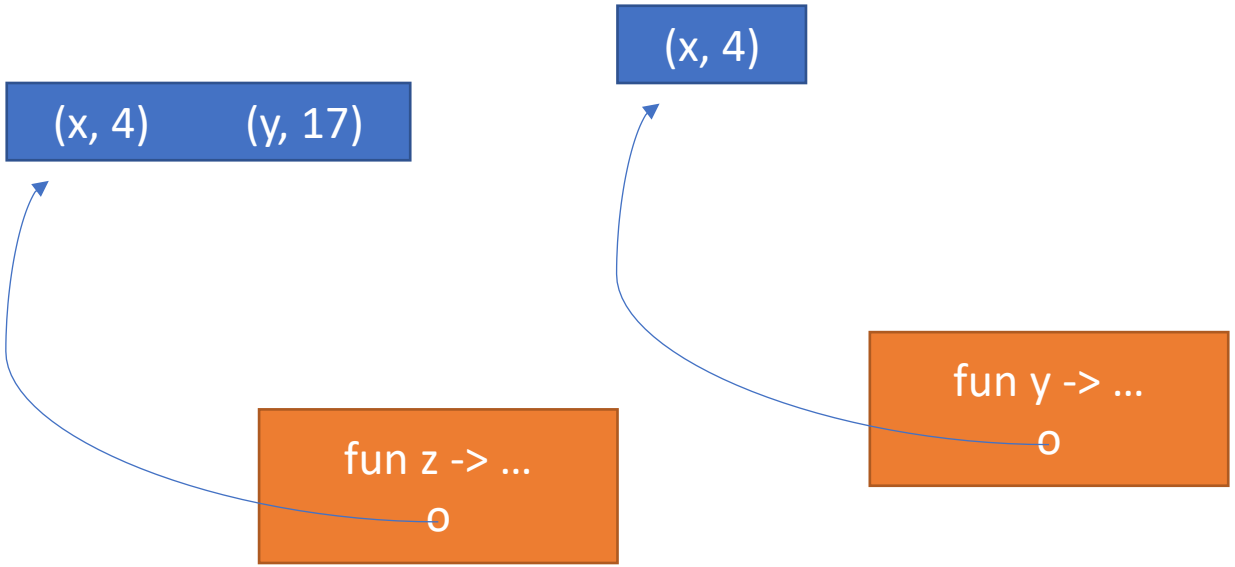
Flat Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Flat Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Pro: Faster lookup
Con: Slower construction

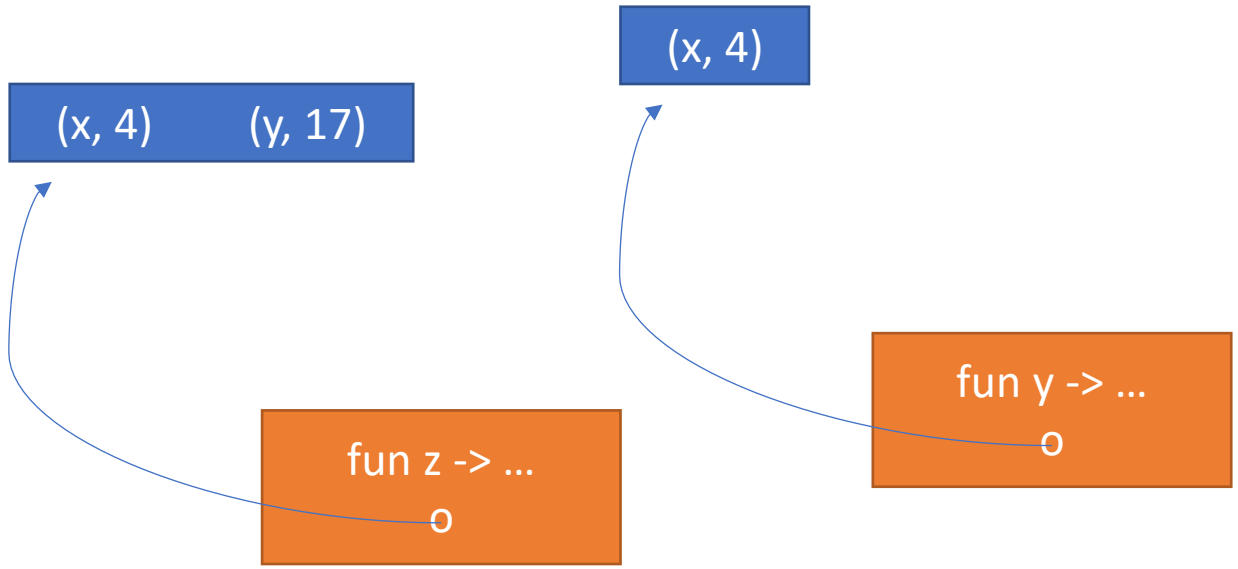
Extend and Lookup for Flat Envs

```
__extend_env(env, var, val):
  env new_env = new (env[env.length + 1])
  env[0] = (var, val)
  env[1:] = copy(env)
  return env

__lookup(env, var):
  i = 0
  while(env[i].var != var && i < env.length):
    i++
  return env[i].val
```

Optimization: We don’t need to add z to the environment!

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Optimizations for flat environments

- You can just produce the environment when you need it for a closure (assuming you know all the values you need to build it)
  - … and you can easily include only the free variables in the body.

Alternate way of thinking of flat environments: the closure *is* the environment

```
(((fun x -> (fun y -> x + (fun z -> y + z) 21) 17) 4
```

Side bonus: special case for recursive closures so we don't have to backpatch