CS443: Compiler Construction

Lecture 12: DeBruijn Indices and Compiling Values

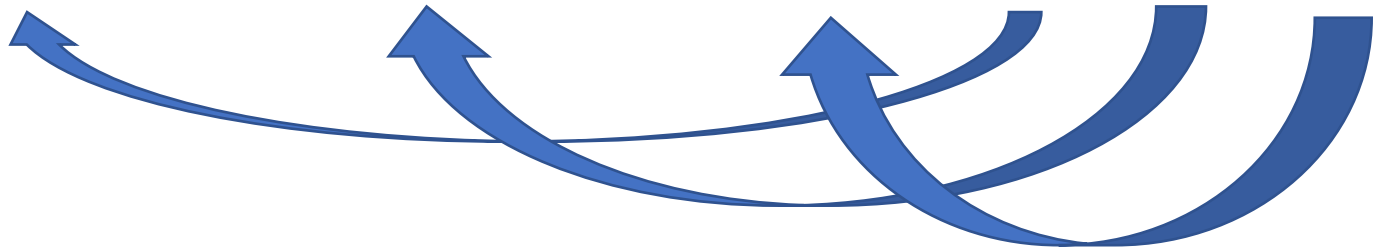
Stefan Muller

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

But getting back to the fact that lookup is still O(n) in the size of the environment…

deBruijn Indices Track Number of Binders

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



deBruijn Indices Track Number of Binders

```
(((fun   -> (fun   -> (fun   ->   +   +  ) 21) 17) 4
```

deBruijn Indices Track Number of Binders

```
(((fun -> (fun -> (fun -> 2 + 1 + 0) 21) 17) 4
```

deBruijn Indices: Example

```
let x = 1 in x +
            (let y = 2 in
               (let x = 3 in x + y)
               + y)

let = 1 in 0 +
          (let = 2 in
             (let = 3 in 0 + 1)
             + 0)
```

Note: Same binder can have different indices at different points in the program!

deBruijn Indices: Another Example

```
let x = 1 in
let add = fun y -> x + y in
let two = add 1 in
two

let = 1 in
let = fun -> 1 + 0 in
let = 0 1 in
0
```

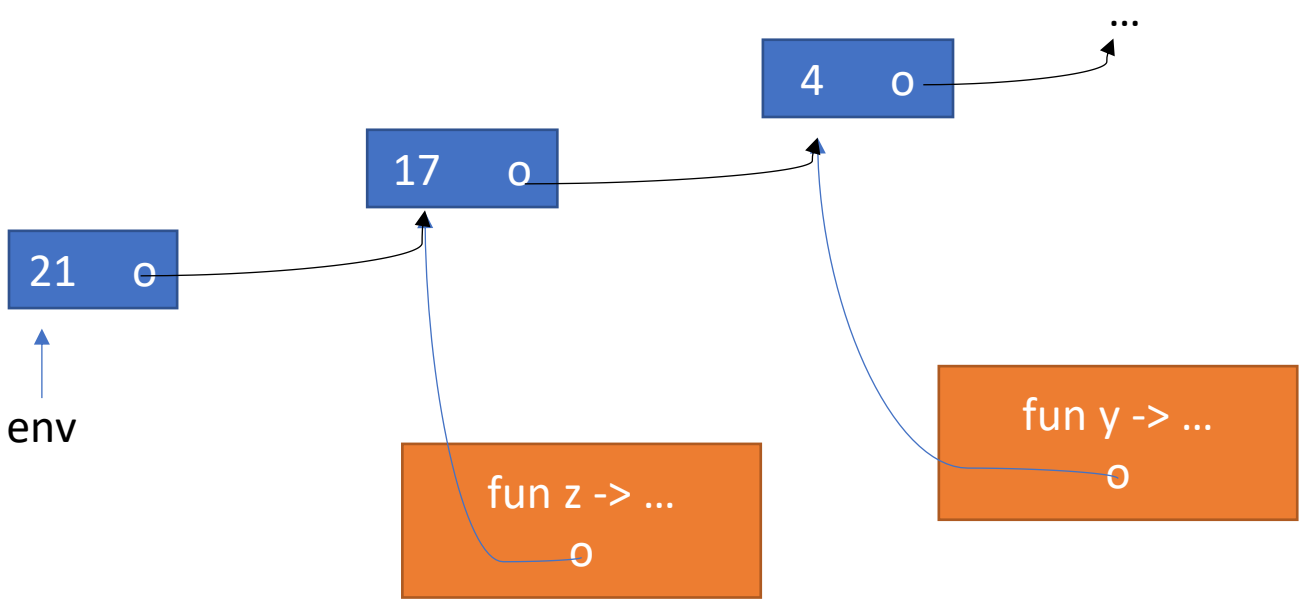
For recursive functions, consider “let rec” to bind the function name in the body

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

let rec = if 0 <= 1 then 0 else 0 * (1 (0 – 1))
```

Nested Environments with deBruijn Indices

```
(((fun -> (fun -> (fun -> 2 + 1 + 0) 21) 17) 4
```



Extend and Lookup for Nested Envs (deBruijn)

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

__lookup(env, ind):
  while(ind > 0):
    env = env.next
    ind--
  return env.val
```

Extend and Lookup for Flat Envs (deBruijn)

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

__lookup(env, ind):
  return env[ind]
```

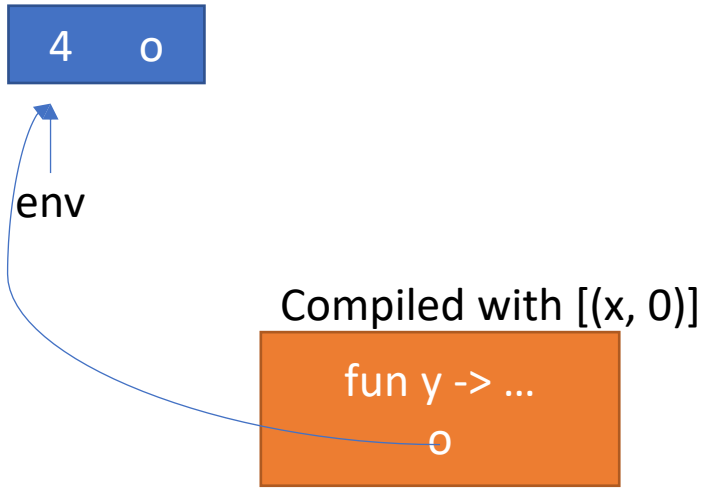
Compromise: Keep variable names, but remember their deBruijn index while compiling


```
compile_exp : (string * int) list -> ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * closure list
```

- Con: Have to keep environment record in sync with environment
- Pro: Way easier to debug

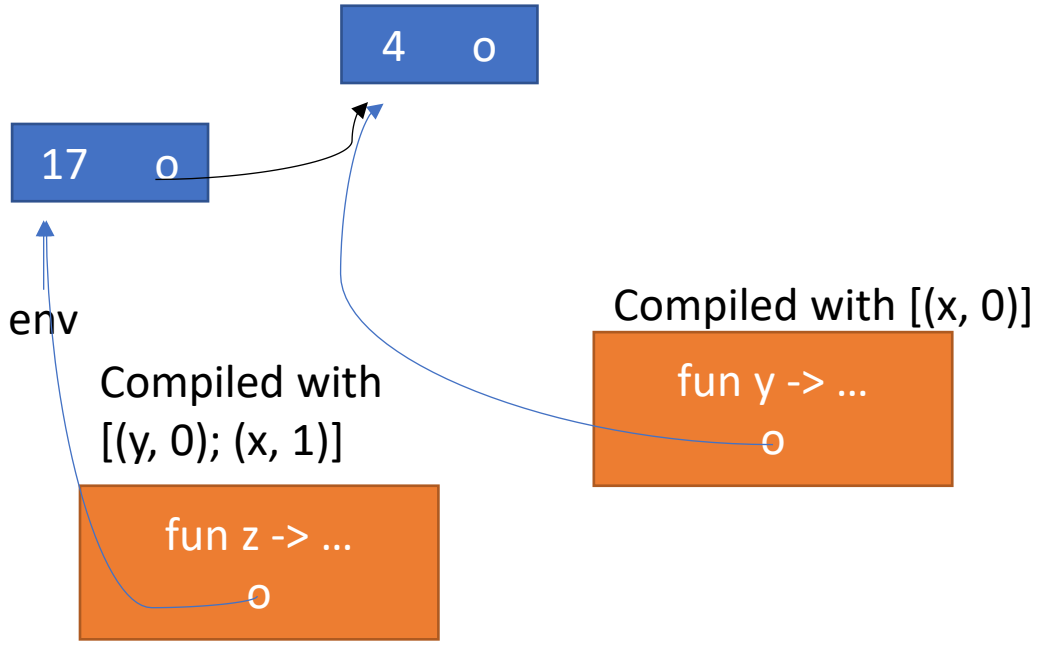
Nested Environments (Compromise)

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



Nested Environments (Compromise)

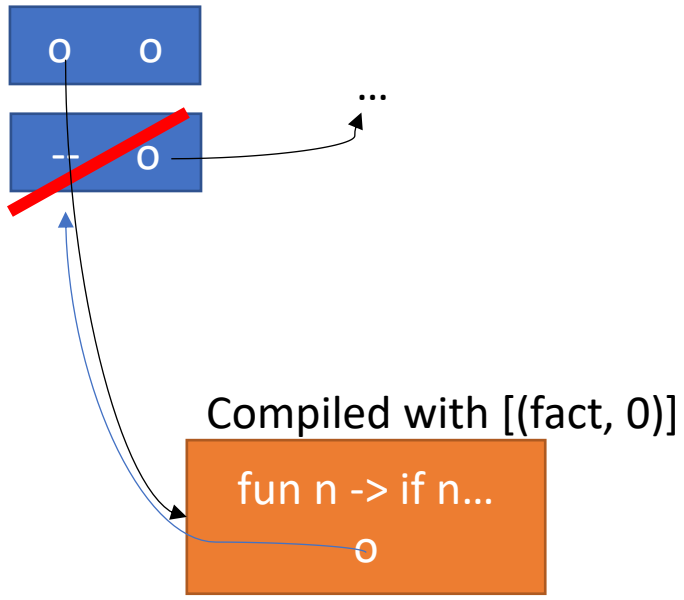
```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



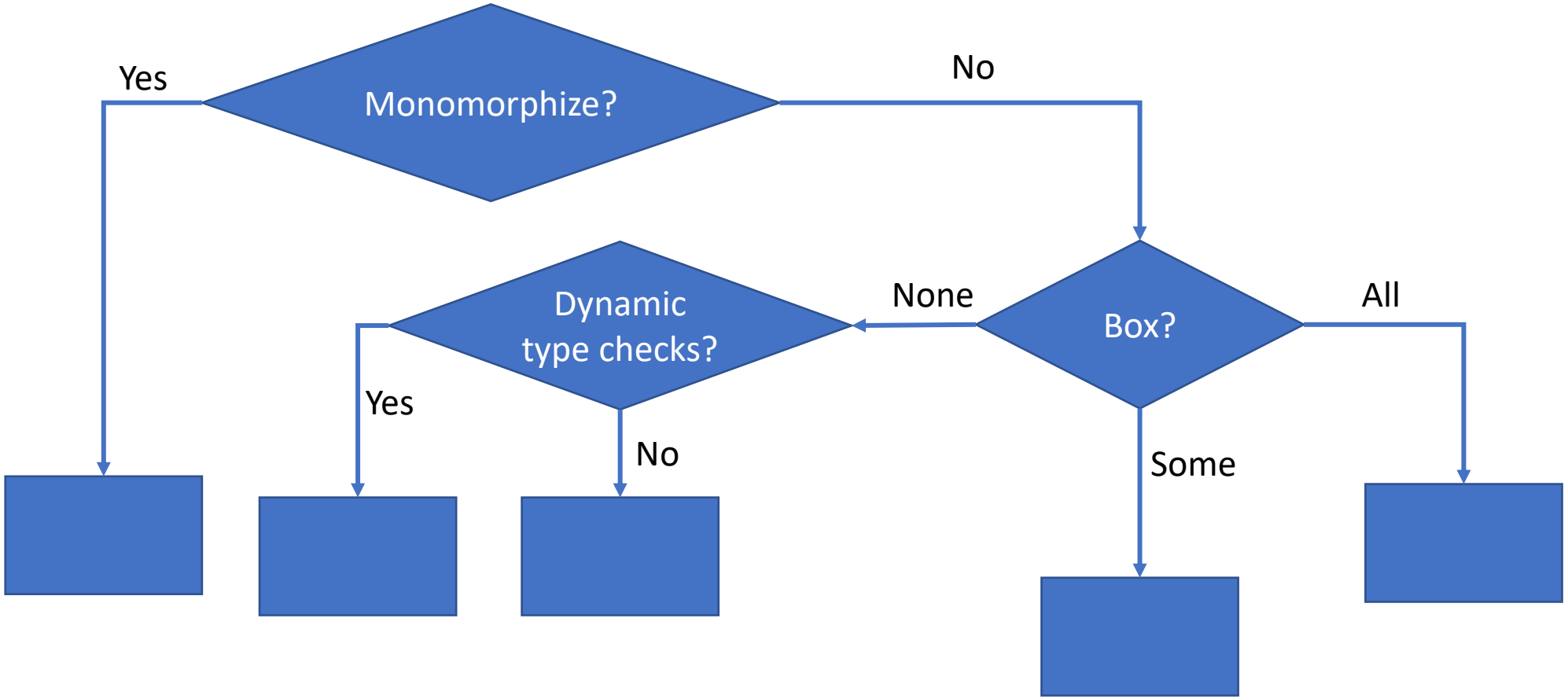
Recursive functions (compromise)

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

1. Extend environment, environment record with placeholder
2. Compile function with extended env. record
3. Make closure with placeholder-extended environment
4. Backpatch environment in closure to point back to closure



There are a lot of ways to compile values



We (probably) want a uniform representation of values

```
'a list
```

```
struct __list{
  value hd;
  __list tl;
};
```

Could pull the "pick a default type and cast as necessary" trick but still want values to all be the same size

First option: actually just have one type of values

```
enum Tag {INT, BOOLEAN, …} ;

struct Int { enum Tag t ; int value ; } ;

struct Boolean { enum Tag t ; unsigned int value ; } ;
…
union Value {
  enum Tag t ;
  struct Int z ;
  struct Boolean b ;
  …
} ;
```

Courtesy Matt Might: https://matt.might.net/articles/compiling-scheme-to-c/

Then we have to check the tag of an object when we use it…

```
Value neg(Value i) {
  switch (i.t) {
    case INT:
      Int ret;
      ret.t = INT;
      ret.value = -((Int) i).value;
      return ret;
    default:
      //Type Error!
      exit 1;
  }
}
```

Easy, Slow, Wasteful

…or do we?

- No (in a statically typed language without something like `instanceof`)

```
Value neg(Value i) {
  Int ret;
  ret.t = INT;
  ret.value = -((Int) i).value;
  return ret;
}
```

Easy, Fast, Wasteful

Second option: “Boxing” (use pointers for everything)

```
//Value is alias for void*
typedef void * Value

struct Int { int value; };
struct Boolean { bool value; };
struct List { Value hd; Value tl };
```

Key idea: we may not know a value’s value at compile time, but we know its type!

Second option: “Boxing” (use pointers for everything)

```
let l: int list = 1::[]
in (hd l) + 2
```

```
Value l = new(List));
Value i = new(Int);
((Int *) i)->value = 1;
((List *) l)->hd = i;
((List *) l)->tl = null;
Value i2 = new(Int);
((Int *) i2)->value = 2;
return ((Int *)((List *) l)->hd)->value + ((Int *) i2)->value
```

Harder, slower, still pretty wasteful

Compromise: “Unbox” ints, other small base types

```
let l: int list = 1::[]
in (hd l) + 2
```

```
Value l = malloc(sizeof(List));
((List *)l)->hd = ((Value) 1);
((List *)l)->tl = null;
return ((int) ((List *)l)->hd) + 2
```

Harder, relatively fast, space-efficient

Have structs for different types

```
struct __list {
  int list_hd;
  __list list_tl;
};

struct __pair {
  int pair_fst;
  int pair_snd;
};
```

We need to pick a default type for values.
May as well use int (no void* in MiniC)

By the way… compiling variables

```
x : t
```

```
[]
```

```
(t)__lookup(env, “x”)
```

```
[]
```

Actually, deBruijn index of x in the environment record

We still need dynamic tag checks for ADTs

```
type exp = EVar of string
         | EBinop of exp * exp
```

```
enum exp_tag { EVAR; EBINOP };
union exp;
struct EVar {
  exp_tag t;
  char[] arg1;
}
struct EBinop {
  exp_tag t;
  union exp *arg1;
  union exp *arg2;
}
union exp {
  struct EVar evar;
  struct EBinop ebinop;
}
```

A totally different option: get rid of polymorphism (“monomorphize”)

```
struct int_list{
  int hd;
  __list tl;
};

struct bool_list{
  bool hd;
  __list tl;
};
```

That means we need to make different versions of polymorphic functions

```
let pair (x: ‘a) : ‘a * ‘a = (x, x)
```

```
intpair pair_int(x: int) { … }
boolpair pair_bool(x: bool) { … }
```

…

(We’ll also need pair_intpair, pair_intboolpair, …)

To monomorphize functions, we need to know all the ways they can be used

- Check all call sites ➔ Whole program compilation

Much harder
Slow, non-modular compilation
Blindingly fast at runtime
Space-efficient

There are a lot of ways to compile values