CS443: Compiler Construction

Lecture 11: Environments and DeBruijn Indices
Stefan Muller

Based on material from Steve Chong, Steve Zdancewic, and Greg Morrisett
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**
  - Linked list (deBruijn)
  - Linked list (var-value pairs)

- **Flat Environments**
  - Array (deBruijn)
  - Array (var-value pairs)
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

`fun y -> ...`
Nested Environments

$$(((\text{fun } x \rightarrow (\text{fun } y \rightarrow (\text{fun } z \rightarrow x + y + z) \ 21) \ 17) \ 4)$$

Environments in closures are still there—just like functional programming!
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 != NULL):
        env = env.next
    return env.val
For recursive functions, can just make the closure and “backpatch” it later

```plaintext
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

(x, 4)

fun y -> ...

0
Flat Environments

\[((\text{fun } x \rightarrow (\text{fun } y \rightarrow (\text{fun } z \rightarrow x + y + z) \text{ 21}) \text{ 17}) \text{ 4} \rceil

\text{Pro: Faster lookup}
\text{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
But getting back to the fact that lookup is still $O(n)$ in the size of the environment...
deBruijn Indices Track Number of Binders

$$(((\text{fun } x \rightarrow (\text{fun } y \rightarrow (\text{fun } z \rightarrow x + y + z) 21) 17) 4$$
deBruijn Indices Track Number of Binders

\(((\text{fun} \rightarrow (\text{fun} \rightarrow (\text{fun} \rightarrow + + ) \ 21) \ 17) \ 4)\)
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

\[
\text{let rec fact } n = \text{ if } n \leq 1 \text{ then } n \text{ else } n \times (\text{fact } (n - 1))
\]

\[
\text{let rec } = \text{ if } 0 \leq 1 \text{ then } 0 \text{ else } 0 \times (1 (0 - 1))
\]
Nested Environments with deBruijn Indices

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

\[ (((\text{fun} \rightarrow (\text{fun} \rightarrow (\text{fun} \rightarrow 2 + 1 + 0) 21) 17) 4 \]]
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)

```python
__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)

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

Compiled with [(x, 0)]
Nested Environments (Compromise)

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

Compiled with [(x, 0)]

Compiled with [(y, 0); (x, 1)]
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