CS443: Compiler Construction

Lecture 10: Closure Conversion

Stefan Muller

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

• Suggests how to compile: closure now doesn’t depend on environment
  • Add code to build closures (closure conversion)
  • Lift code parts of closures into top-level functions (hoisting/lambda lifting)
Add the environment as an extra parameter to functions

fun (y: int) : int -> x + y

int __fun (env env, int y) {
    env = __extend_env(env, "y", y);
    return __lookup(env, "x") + y;
}
Can also just look \( y \) up in the environment

\[
\text{fun (y: int) : int -> x + y}
\]

\[
\text{int \_fun (env env, int y) \{
env = \_extend_env(env, "y", y);
return \_lookup(env, "x") + \_lookup(env, "y");
\}}
\]

**Pro**: uniform treatment of vars

**Con**: Less efficient
We need to make sure the environment keeps up with ML variable scope

```ml
let x = (let x = 1 in x + x) + 1 in x
```

```ml
int x_1 = 1
env = __extend_env(env, "x", x_1);
int temp_1 = x_1 + x_1;
env = __pop_env(env);
int x_2 = temp_1 + 1;
env = __extend_env(env, "x", x_2);
int temp_3 = x_2;
env = __pop_env(env);
```
As suggested by “extend” and “pop”, environment follows a stack

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

```plaintext
int x_1 = 1;
env = __extend_env(env, "x", x_1);
int y_1 = 2;
env = __extend_env(env, "y", y_1);
temp_1 = x_1 + y_1;
env = __pop_env(env);
temp_2 = x_1 + temp_1 + x_1
env = __pop_env(env);
```
A closure is a pair of the function code and the current environment

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(“fun y -> x + y” , env);
env = __extend_env(env, “inc”, inc_1);
int temp_1 = __call_closure(inc_1, 2);
(But the function code needs to be lifted to the top level)

```c
int inc1__body(env env, int y) {
    env = __extend_env(env, "y", y);
    return __lookup(env, "x") + y;
}

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 = __call_closure(inc_1, 2);
```
Call a closure by calling the function with the closure’s environment (NOT the current one)

```c
int inc1__body(env env, int y) {
    env = __extend_env(env, "y", y);
    return __lookup(env, "x") + y;
}
```

```c
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)
```
For recursive functions, the function itself needs to be in the environment

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

```plaintext
int fact__body(env env, int n) {
  env = __extend_env(env, “n”, n);
  if (n <= 1) { return n; }
  else {
    return n * __lookup(env, “fact”).clos_fun(
      __lookup(env, “fact”).clos_env, n - 1);
  }
}

env = __extend_env(env, “fact”, __mk_clos(fact__body, env))
```

Gets a little tricky depending on how we define environments—we’ll revisit this later
Do closure conversion and hoisting in one pass

\[ \text{compile\_exp} : \text{ML}\text{.Ast.t\_exp} \rightarrow \]
\[ \text{C}\text{.Ast.p\_stmt\_list} * \text{C}\text{.Ast.p\_exp} * \text{closure\ list} \]

- 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)
- Closures 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 * closure 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)

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

temp_1
Do closure conversion and hoisting in one pass

**compile_exp**: ML.Ast.t_exp ->  
C.Ast.p_stmt_list * C.Ast.p_exp * closure 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);

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

```
[int inc1__body(env env, int y) {
  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(env env, int x) {
    env = __extend_env(env, “x”, x);
    ... compilation of e ...
}]; plus any closures nested in e]
```
Funcception

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

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

int __fun2(env env, int y) {
    __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

\[ \text{compile\_exp}(e_1 \; e_2) \]

\[ \begin{align*}
\text{statements for } e_1 \\
\text{statements for } e_2
\end{align*} \]

\[ \text{exp}_e1.clos\_fun(\text{exp}_e1.clos\_env, \text{exp}_e2) \]

\[ (\text{closures from } e_1) \; @ \; (\text{closures from } e_2) \]
Applications evaluate the two expressions, then apply

\[ \text{compile\_exp(} \text{let } x = e_1 \text{ in } e_2) \]

- statements for \( e_1 \)
- statements for \( e_2 \)

\((\text{closures from } e_1) @ (\text{closures from } e_2)\)
Applications evaluate the two expressions, then apply

\[
\text{compile}\_\text{exp}(\text{let } x = e_1 \text{ in } e_2)
\]

**statements for** \(e_1\)
\[
\text{env} = \_\text{extend}\_\text{env}(\text{env}, "x", e_1\_\text{exp});
\]

**statements for** \(e_2\)
\[
\text{temp}_1 = e_2\_\text{exp};
\]
\[
\text{env} = \_\text{pop}\_\text{env}(\text{env});
\]

\[
\text{temp}_1
\]

\[
(\text{closures from } e_1) \odot (\text{closures from } e_2)
\]
Think about what goes wrong if we did this

\[
\text{compile_exp}\left(\text{let } x = e_1 \text{ in } e_2\right)
\]

\[
\text{statements for } e_1 \\
\text{env} = \text{extend_env(}\text{env, } "x", \text{ e}_1\text{)};
\]
\[
\text{statements for } e_2 \\
\text{env} = \text{pop_env(}\text{env)};
\]

\[
e_2\text{ exp}
\]

\[
(\text{closures from } e_1) @ (\text{closures from } e_2)
\]
I guess we need to compile other things too

\[
\text{compile}\_\text{exp}(\text{if } e_1 \text{ then } (e_2: \text{ int}) \text{ else } (e_3: \text{ int}))
\]

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

// closure @
(closures from e1) @ (closures from e2) @ (closures from e3)
```
How to *actually* represent closures

```c
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.
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)
- **Flat Environments**
  - Array (deBruijn)
  - Array (var-value pairs)

Slower -> Faster
Harder -> Easier
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

CS 443 - Fall 2022 - Lecture 10
Nested Environments

(((fun x -> (fun y -> (fun z -> 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
Flat Environments

(((\(x \rightarrow (\(y \rightarrow (\(z \rightarrow x + y + z) 21) 17) 4

\(x, 21\)

\(\text{fun y} \rightarrow \ldots\)

0
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
deBruijn Indices Track Number of Binders

(((fun x -> (fun y -> (fun z -> 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 $x +$

$(\text{let } y = 2 \text{ in }$

$(\text{let } x = 3 \text{ in } x + y) + y)$

let $= 1$ in $0 +$

$(\text{let } = 2 \text{ in }$

$(\text{let } = 3 \text{ in } 0 + 1) + 0)$

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

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

Compiled with [(x, 0)]
Nested Environments

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

17 o

21 o

env

Compiled with [(x, 0)]

fun y -> ...

o

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

fun z -> ...

o
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;
};
First option: actually just have one type of values

```c
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/](https://matt.might.net/articles/compiling-scheme-to-c/)
Then we have to check the tag of an object when we use it...

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

Easy, Slow, Wasteful
...or do we?

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

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

Easy, Fast, Wasteful
Second option: “Boxing” (use pointers for everything)

typedef void * Value

struct Int { int value; };
struct Boolean { bool value; };
struct List { Value hd; Value tl };
Second option: “Boxing” (use pointers for everything)

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

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

Value l = malloc(sizeof(List));
((List *)l)->hd = (Value 1);
((List *)l)->tl = null;
return ((Int *) l->tl)->value + ((Int *) i2)->value

Harder, relatively fast, space-efficient
Have structs for different types

```c
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)
We still need dynamic tag checks for ADTs

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

define 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")

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

struct bool_list{
    boolean 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

- Monomorphize?
  - Yes: MLton
  - No: Dynamic type checks?
    - Yes: Python
    - No: Matt Might’s Scheme compiler
- None: Box?
  - None: OCaml
  - Some: All