CS443: Compiler Construction

Lecture 10: FP and Closures

Stefan Muller

Based on material from Steve Zdancewic

Functional languages have first-class and nested functions

- Languages like ML, Haskell, Scheme, Python, C#, Java, Swift
  - Functions can be passed as arguments (e.g., map or fold)
  - Functions can be returned as values (e.g., compose)
  - Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?
  - in an interpreter? in a compiled language?

Let’s take a (very) small subset of OCaml

e ::= fun x -> e | e e | x

Operational semantics of the lambda calculus is by *substitution*

- e{v/x} : substitute v for all *free* instances of x in e

- We say that the variable `x` is *free* in `fun y → x + y`
  - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by "`fun y`" and its *scope* is the body "`x + y`" in the expression `fun y → x + y`
- Alternatively: free = not bound

- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

Free Variables, formally

fv(x)	=	{x}
fv(fun x → exp)	=	fv(exp) \ {x} *('x' is a bound in exp)*
$fv(exp_1\ exp_2)$	=	$fv(exp_1) \cup fv(exp_2)$

Substitution Definition + Examples

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming y ≠ x)*
(fun x → exp){v/x}	= (fun x → exp)	*(x is bound in exp)*
(fun y → exp){v/x}	= (fun y → exp{v/x})	*(assuming y ≠ x)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

(x y) {(fun z → z z)/y} = x (fun z → z z)

(fun x → x y){(fun z → z z)/y} = fun x → x (fun z → z z)

(fun x → x){(fun z → z z)/x} = fun x → x // x is not free!

This definition enables *partial application*

```
let add = fun x -> fun y -> x + y
let add1 = add 1 = (fun y -> x + y){1/x}
                 = fun y -> 1 + y
```

Result is a function!

Nobody implements interpreters for functional PLs using substitution

- Why?
  - Slow

More efficient implementation: first try

```
let add = fun (x, y) -> x + y
let three = add 1 2
```

Var	Value

More efficient implementation: first try

```
let add = fun (x, y) -> x + y
let three = add 1 2
```

Var	Value
add	fun (x, y) -> x + y

More efficient implementation: first try

```
let add = fun (x, y) -> x + y
let three = add 1 2
```

Var	Value
add	fun (x, y) -> x + y
x	1
y	2

More efficient implementation: first try

```
let add = fun (x, y) -> x + y
let three = add 1 2
```

Var	Value
add	fun (x, y) -> x + y
three	3

More efficient implementation: first try

```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```

Var	Value
add	fun x -> fun y -> x + y

More efficient implementation: first try

```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```

Var	Value
add	fun x -> fun y -> x + y
x	1

More efficient implementation: first try

```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```

Var	Value
add	fun x -> fun y -> x + y
add1	fun y -> x + y

Uh oh

More efficient implementation: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value

More efficient implementation: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1

More efficient implementation: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
f	fun y -> x + y

More efficient implementation: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	2
f	fun y -> x + y

More efficient implementation: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	2
f	fun y -> x + y
y	2

x should still be 1 in f!

Second try: use *closures*

- Closure: function code + environment
- This will be the value of a function

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

<table>
<tr><th>Var</th><th>Value</th></tr>
<tr><td>x</td><td>1</td></tr>
<tr><td>f</td><td>(fun y -> x + y,
<table><tr><th>Var</th><th>Value</th></tr><tr><td>x</td><td>1</td></tr></table>)</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

<table>
<tr><th>Var</th><th>Value</th></tr>
<tr><td>x</td><td>2</td></tr>
<tr><td>f</td><td>(fun y -> x + y,
<table><tr><th>Var</th><th>Value</th></tr><tr><td>x</td><td>1</td></tr></table>)</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

<table>
<tr><th>Var</th><th>Value</th></tr>
<tr><td>x</td><td>1</td></tr>
<tr><td>f</td><td>(fun y -> x + y,
<table><tr><th>Var</th><th>Value</th></tr><tr><td>x</td><td>1</td></tr></table>)</td></tr>
<tr><td>y</td><td>2</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

Call the function with the environment from the closure (+ arguments)

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

Environment now includes y also.

Environment loses y when y goes out of scope

Can also just look y up in the environment

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

```
int __fun (int y, env env) {
  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

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

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

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

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)

```
int inc1__body(int y, env env) {
  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)

```
int inc1__body(int y, env env) {
  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 = inc_1.clos_fun(2, inc_1.clos_env)
```

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

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

int fact__body(int n, env env) {
  env = __extend_env(env, "n", n);
  if (n <= 1) { return n; }
  else {
    return n * __lookup(env, "fact").clos_fun(n – 1,
                __lookup(env, "fact").clos_env);
  }
}
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