CS443: Compiler Construction

Lecture 9: 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 | (e)

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 $y \neq x$)*
$(\text{fun } x \rightarrow exp)\{v/x\}$	$= (\text{fun } x \rightarrow exp)$	*(x is bound in exp)*
$(\text{fun } y \rightarrow exp)\{v/x\}$	$= (\text{fun } y \rightarrow exp\{v/x\})$	*(assuming $y \neq x$)*
$(e_1\ e_2)\{v/x\}$	$= (e_1\{v/x\}\ e_2\{v/x\})$	*(substitute everywhere)*

- Examples:

$(x\ y)\ \{(\text{fun } z \rightarrow z\ z)/y\} = x\ (\text{fun } z \rightarrow z\ z)$

$(\text{fun } x \rightarrow x\ y)\{(\text{fun } z \rightarrow z\ z)/y\} = \text{fun } x \rightarrow x\ (\text{fun } z \rightarrow z\ z)$

$(\text{fun } x \rightarrow x)\{(\text{fun } z \rightarrow z\ z)/x\} = \text{fun } x \rightarrow x$ // 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!

If we naively substitute an open term, variables can be *captured*

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

Note: x is *free* in (fun z → x)

free x is *"captured"!!*

- Alpha equivalence to the rescue: names of bound vars don't matter!
  fun x -> (x y) = fun a -> (a y)

(fun a → (a y)){(fun z → x)/y}
 = fun a → (a (fun z -> x))

Alpha equivalence: real life application!

```
let rec qsort l =
  let (a, b) = split l in
  let asorted = qsort a in
  let bsorted = qsort b in
  merge asorted bsorted
```

```
let rec qsort l =
  let (alist, blist) = split l in
  let a_sort = qsort a_list in
  let b_sort = qsort b_list in
  merge a_sort b_sort
```

Substitution with open terms

$x\{e/x\} = e$ *(replace the free x by v)*

$y\{e/x\} = y$ *(assuming $y \neq x$)*

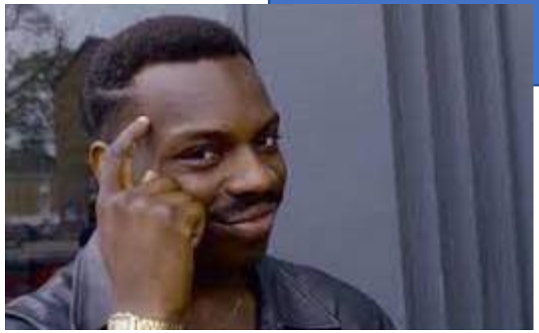
$(\text{fun } x \rightarrow e_1)\{e_2/x\} = (\text{fun } x \rightarrow e_1)$ *(x is bound in exp)*

$(\text{fun } y \rightarrow e_1)\{e_2/x\} = (\text{fun } y \rightarrow e_1\{e_2/x\})$ *(assuming $y \neq x$, $y \notin fv(e_2)$)*

$(e_1\ e_2)\{e/x\} = (e_1\{e/x\}\ e_2\{e/x\})$ *(substitute everywhere)*

Or just alpha convert everywhere right at the beginning so all the var names are different

If it is?
Alpha convert!



Example

(fun x → (x y)){(fun z → x)/y}

= (fun x’ → (x' (fun z → x))

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
```

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

<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>

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*)