# CS653: Functional Programming
# 2017-18 $II^{nd}$ Semester

# Supercombinators

Amey Karkare
karkare@cse.iitk.ac.in

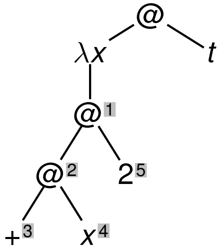
http://www.cse.iitk.ac.in/~karkare/cs653
Department of CSE, IIT Kanpur

# Agenda

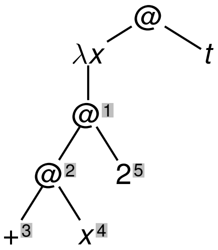
- Towards Compilation
- Supercombinators
- $\lambda$ Lifting
- Full Laziness

# Towards Compilation

- The instantitate process is an inefficient one
- Consider the redex $(\lambda x.x + 2)t$



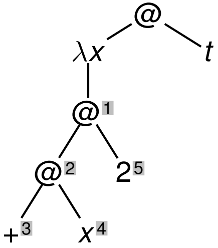

- While instantiating, the detailed steps involved are ...



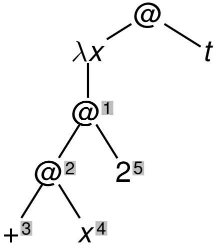

1. Check the type of node 1: @ node
2. Make a new @ node (@-2)
3. Check the type of node 2: @ node
4. Make a new node (@-4)
5. Check the type of node 3: Built-in +
6. Make the left child of @-4 point to 3
7. Check the type of node 4: Variable
8. Is this also the $\lambda$ variable?: Yes
9. Make the right child of @-4 point to the location of the argument $t$
10. Make the left child of @-2 point to @-4
11. Check the type of node 5: Constant
12. Make the right child of @-2 point to node 5

- Everytime $(\lambda x.x + 2)$ is encountered, this process is repeated
- How do we optimize it?
- Separate the *checking* parts from the *imperative* parts
- Checking part is done only once per compilation
- Code for performing Imperative part is generated by the compiler
  - Executed many times, without the penalty of the checks
  - The essence of compilation

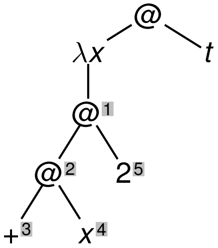
1. Check the type of node 1: @ node
2. Make a new @ node (@-2)
3. Check the type of node 2: @ node
4. Make a new node (@-4)
5. Check the type of node 3: Built-in +
6. Make the left child of @-4 point to 3
7. Check the type of node 4: Variable
8. Is this also the $\lambda$ variable?: Yes
9. Make the right child of @-4 point to the location of the argument *t*
10. Make the left child of @-2 point to @-4
11. Check the type of node 5: Constant
12. Make the right child of @-2 point to node 5



1. ~~Check the type of node 1: @ node~~
2. Make a new @ node (@-2)
3. ~~Check the type of node 2: @ node~~
4. Make a new node (@-4)
5. ~~Check the type of node 3: Built-in +~~
6. Make the left child of @-4 point to 3
7. ~~Check the type of node 4: Variable~~
8. ~~Is this also the $\lambda$ variable?: Yes~~
9. Make the right child of @-4 point to the location of the argument *t*
10. Make the left child of @-2 point to @-4
11. ~~Check the type of node 5: Constant~~
12. Make the right child of @-2 point to node 5



1. Make a new @ node (@-2)
2. Make a new node (@-4)
3. Make the left child of @-4 point to 3
4. Make the right child of @-4 point to the location of the argument *t*
5. Make the left child of @-2 point to @-4
6. Make the right child of @-2 point to node 5



# Any Issues?

- Instantiating the $\lambda$ body requires us to know the shape of the body
- Compile time decisions can not be taken if the body is *unknown*
- This could happen if the $\lambda$ body has free variables in it
- Example:

$$\lambda z.z + (\lambda y.y + z)\ t$$

# Any Issues

- Code can not be generated at compile time to instantitate

$$\lambda y.y + z$$

- The instantiation will depend on the value of $z$
- Different situation than making a pointer point to an argument

# Solution?

- Situation can be handled if there is a continuous run of $\lambda$s

$$\lambda z \lambda y . y + z$$

- We can do simultaneous instantiation of $z$ and $y$
- When both the arguments are avaliable
- We need to change the definition of WHNF to say that $\lambda z \lambda y . y + z$ applied to less than two arguments is in WHNF.
- What sort of $\lambda$ terms can be compiled? Supercombinators

# Supercombinators

- A supercombinator, \$S, of arity $n$ is a $\lambda$ expression of the form $\lambda x_1 \lambda x_2 \ldots \lambda x_n.E$ such that
  1. $E$ is not a $\lambda$ abstraction
  2. \$S has no free variables
  3. any $\lambda$ abstraction in $E$ is a supercombinator
  4. $n \geq 0$, i.e., there need be no $\lambda$s at all.

# Supercombinators

| | | | |
|---|---|---|---|
| 3 | ✓ | $\lambda x.y$ | $\times$ |
| (+ 2 5) | ✓ | +2 | ✓ |
| $\lambda y.x - y$ | $\times$ | $\lambda x.x$ | ✓ |
| $\lambda f.f(\lambda x.f\ x\ 3)$ | $\times$ | $\lambda x.x + 1$ | ✓ |
| $\lambda x \lambda y.x - y$ | ✓ | $\lambda f.f(\lambda x \lambda y.x - y)$ | ✓ |

# Supercombinators

- A Supercombinator of arity $n$ is not necessarily a function of $n$ arguments.
- It will be involved in instantiation involving $n$ arguments.
- Exercise: What is a *Combinator*? Examples of terms that are combinators but not supercombinators?

# Supercombinators and $\lambda$-lifting

- How to convert a lambda expression into SCs?
- The process we shall study is called $\lambda$-lifting
- We shall drive the process through examples

# $\lambda$-lifting Example

$$(\lambda x.x + (\lambda y.x + y))3$$

1. Take out the free variable of the inner lambda expression as a parameter
$$\lambda x \lambda y.x + y$$
2. Give it a name:
$$\$XY\ x\ y = x + y$$
3. Write the original lambda term in terms of $\$XY$
$$\lambda x.x + \$XY\ x$$
4. This is already in SC form. Give this a name:
$$\$X\ x = x + \$XY\ x$$

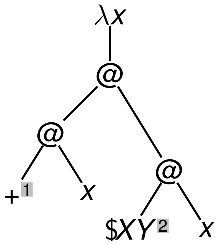
# $\lambda$-lifting Example

- Summing up, we have

$$\begin{aligned}
\$XY\ x\ y &= x + y \\
\$X\ x &= x + \$XY\ x \\
\$Prog &= \$X\ 3
\end{aligned}$$

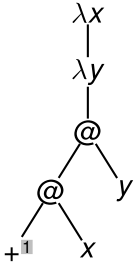
# The Graph and Instantiating code for $X

1. Make a new @ node (@-1)
2. Make a new @ node (@-2)
3. Make the left child of @-2 point to node 1
4. Make the right child of @-2 point to the arg
5. Make a new node (@-5)
6. Make the left child of @-5 point to the $*XY*
7. Make the right child of @-5 point to the arg
8. Make the left child of @-1 point to @-2
9. Make the right child of @-1 point to @-5



# The Graph and Instantiating code for $XY

1. Make a new @ node (@-1)
2. Make a new @ node (@-2)
3. Make the left child of @-2 point to node 1
4. Make the right child of @-2 point to the arg-1
5. Make the left child of @-1 point to @-2
6. Make the right child of @-1 point to the arg-2



# $\lambda$-lifting Example 2

$$(\lambda x.x + (\lambda y.(\lambda z.x + y * z)))\ 6$$

1. Take the innermost $\lambda$ abstraction and bring out free variables as parameters

$$\lambda y \lambda x \lambda z.x + y * z$$

2. Name this SC

$$\$YXZ\ y\ x\ z = x + y * z$$

3. Substitute it back

$$(\lambda x.x + (\lambda y.\$YXZ\ y\ x))\ 6$$

# $\lambda$-lifting Example 2

$$(\lambda x.x + (\lambda y.\$YXZ\ y\ x))\ 6$$

4. Do the same for the next (now innermost) abstraction with free variables
$$\$XY\ x\ y = \$YXZ\ y\ x$$

5. Substitute
$$(\lambda x.x + \$XY\ x)\ 6$$

6. Continuing, we get
$$\begin{array}{rcl} \$YXZ\ y\ x\ z & = & x + y + z \\ \$XY\ x\ y & = & \$YXZ\ y\ x \\ \$X\ x & = & x + \$XY\ x \\ \$Prog & = & \$X\ 6 \end{array}$$

# $\lambda$-lifting Example 3

$$(\lambda x.x + (\lambda y.(\lambda z.x + y * z))) \; 6$$

1. Take the innermost $\lambda$ abstraction and bring out free variables as parameters (in a different order)

$$\lambda x \lambda y \lambda z.x + y * z$$

2. Name this SC

$$\$XYZ \; x \; y \; z = x + y * z$$

3. Substitute it back

$$(\lambda x.x + (\lambda y.\$XYZ \; x \; y)) \; 6$$

# $\lambda$-lifting Example 3

$$(\lambda x.x + (\lambda y.\$XYZ\ x\ y))\ 6$$

4. Do the same for the next (now innermost) abstraction with free variables

$$\$XY\ x\ y = \$XYZ\ x\ y$$

5. Substitute

$$(\lambda x.x + \$XY\ x)\ 6$$

6. Continuing, we get

$$\begin{aligned} \$XYZ\ x\ y\ z &= x + y + z \\ \$XY\ x\ y &= \$XYZ\ x\ y \\ \$X\ x &= x + \$XY\ x \\ \$Prog &= \$X\ 6 \end{aligned}$$

# $\lambda$-lifting Example 3

- Note that, in this case we can optimize the program

$$\begin{array}{rcl} \$XYZ\ x\ y\ z & = & x + y + z \\ \$XY\ x\ y & = & \$XYZ\ x\ y \\ \$X\ x & = & x + \$XY\ x \\ \$Prog & = & \$X\ 6 \end{array}$$

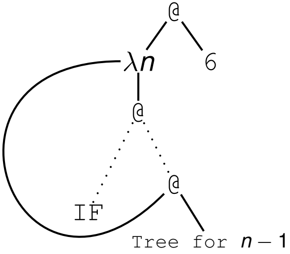
to get

$$\begin{array}{rcl} \$XYZ\ x\ y\ z & = & x + y + z \\ \$X\ x & = & x + \$XYZ\ x \\ \$Prog & = & \$X\ 6 \end{array}$$

# Recursive Supercombinators

- If we permit cycles in the graph, we can retain let(rec)s.
- There is a direct correspondence between letrecs and cyclic graphs.
- A letrec corresponds to a possibly cyclic graph

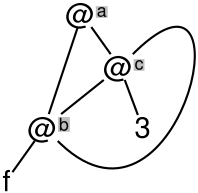
```
letrec
    fact = λn.if n==0 then 1 else n⋆fact(n−1)
in fact 6
```




- This is more efficient than using Y explicitly. If we use Y, then after two reductions we get this graph.

# Letrecs and Cyclic Graphs

- A graph can always be converted to a letrec
- For example




can be written as

```
letrec
     a = b c
     b = f c
     c = b 3
in a
```

# Supercombinators with letrecs

- A letrec at the outermost level will be converted into recursive supercombinators.
- Inner letrecs will be left as such.
- After lambda lifting, the general form of the program is

```
$X1 a b c = e1
$X2 x y z = e2
 $Prog    = e
```

- Here `e1, e2, e` etc may contain letrecs

# Example

```
letrec
   sumInts m = sum (count 1)
               where count n | n > m = []
                             | otherwise = n:count(n+1)
   sum [] = 0
   sum (x:xs) = n+sum ns
in sumInts 100
```

After conversion to the enriched form:

```
letrec
   sumInts = λm.letrec
                   count = λn.if (n>m) then [] else n:count(n+1)
                in sum (count 1)
   sum = λl.if l==[] then 0 else (head l)+sum(tail l)
in sumInts 100
```

# Example

$\lambda$-lift the RHS of `count`

```
$N count m n = if (n>m) then [] else n:count(n+1)
```

## rewrite

```
letrec
   sumInts = λm.letrec
                   count = $N count m
                in sum (count 1)
   sum = λl.if l==[] then 0 else (head l)+sum(tail l)
in sumInts 100
```

# Example

Now there are no more inner lambdas. Therefore we can write sumInts and sum as recursive supercombinators.

```
$N count m n = if (n>m) then [] else n:count(n+1)
$sumInts m =letrec
               count = $N count m
            in $sum (count 1)
$sum l = if l==[] then 0 else (head l)+$sum(tail l)
$Prog = $sumInts 100
```

# Fully-lazy $\lambda$-lifting

- The process described earlier can still result in multiple evaluations of same expression
- It can create *multiple instances of the same expression*, rather than sharing a single copy of them.

# Losing Lazyness: Example

```
f = g 4
g x y = y + (sqrt x)
(f 1) + (f 2)
```

When evaluating the expression, we get

```
  + (f 1) (f 2)
→ + ( • 1) ( • 2)
```

$((\lambda x \lambda y. + y\ \ (sqrt\ \ x))\ \ 4)$

```
→ + ( • 1) ( • 2)
```

$((\lambda y. + y\ \ (sqrt\ \ 4))$

```
→ + ( • 1) (+ 2 (sqrt 4))
```

$((\lambda y. + y\ \ (sqrt\ \ 4))$

```
→ + ( • 1) 4
```

$((\lambda y. + y\ \ (sqrt\ \ 4))$

```
→ + (+ 1 (sqrt 4)) 4
→ + 3 4
→ 7
```

(sqrt 4) is getting evaluated twice!

# Losing Lazyness: Supercombinators Don't Help

```
$g x y = y + (sqrt x)
$f = g 4
$Prog = ($f 1) + ($f 2)
```

When evaluating the expression, we get

```
   $Prog
→  + ( • 1) ( • 2)
                              ($g 4)
→  + ( • 1) (+ 2 (sqrt 4))
                              ($g 4)
→  + ( • 1) 4
                              ($g 4)
→  + (+ 1 (sqrt 4)) 4
→  + 3 4
→  7
```

# Maximal Free Expressions

- **Free Subexpression:** A subexpression E of a $\lambda$ abstraction L is free in L if all variables in E are free in L.
- **Maximal Free Subexpression (MFE):** A MFE of L is a free expression which is not a *proper* subexpression of another free subexpression of L.

# Examples

MFEs of $\lambda x$ abstractions are shaded.

- $\lambda x.$ *sqrt* $x$
- $\lambda x.$ *sqrt* $4$
- $\lambda y \lambda x. + \; x \; (* \; y \; y)$
- $\lambda y \lambda x. + (* \; y \; y) \; x$
- $\lambda x. (\lambda x.x) \, x$

# Full Laziness and Maximal Free Expressions

- When copying $\lambda$ abstractions during instantiation, do not copy MFEs.
- Instead, substitute a pointer to the single shared instance in the body of the $\lambda$ abstraction.

# Lazyness and MFEs: Example

```
f = g 4
g x y = y + (sqrt x)
```

```
  + (f 1) (f 2)
→ + ( 1) ( 2)
```

$((\lambda x \lambda y. + y \ (sqrt \ x)) \ 4)$

```
→ + ( 1) ( 2)
```

$((\lambda y. + y \ (sqrt \ 4))$

```
→ + ( 1) (+ 2 )
```

$((\lambda y. + y \ (sqrt \ 4))$

```
→ + ( 1) (+ 2 )
```

$((\lambda y. + y \ 2)$

```
→ + ( 1) 4
```

$((\lambda y. + y \ 2)$

```
→ + (+ 1 ) 4
```

2

```
→ + 3 4
→ 7
```