

# CS738: Advanced Compiler Optimizations

## The Untyped Lambda Calculus

Amey Karkare

karkare@cse.iitk.ac.in

<http://www.cse.iitk.ac.in/~karkare/cs738>

Department of CSE, IIT Kanpur



# Reference Book

Types and Programming Languages by Benjamin C. Pierce

# The Abstract Syntax

$t ::= x$                       – Variable

# The Abstract Syntax

$t ::= x$                       – Variable  
       $| \lambda x.t$                 – Abstraction

# The Abstract Syntax

$t$	$:=$	$x$	– Variable
		$ \ \lambda x.t$	– Abstraction
		$ \ t\ t$	– Application

# The Abstract Syntax

$t$	$:=$	$x$	– Variable
		$ \ \lambda x.t$	– Abstraction
		$ \ t\ t$	– Application

Parenthesis,  $(\dots)$ , can be used for grouping and scoping.

# Conventions

- ▶  $\lambda x.t_1 t_2 t_3$  is an abbreviation for  $\lambda x.(t_1 t_2 t_3)$ , i.e., the scope of  $x$  is as far to the right as possible until it is

# Conventions

- ▶  $\lambda x.t_1 t_2 t_3$  is an abbreviation for  $\lambda x.(t_1 t_2 t_3)$ , i.e., the scope of  $x$  is as far to the right as possible until it is
  - ▶ terminated by a  $)$  whose matching  $($  occurs to the left of  $\lambda$ ,  
OR



# Conventions

- ▶  $\lambda x.t_1 t_2 t_3$  is an abbreviation for  $\lambda x.(t_1 t_2 t_3)$ , i.e., the scope of  $x$  is as far to the right as possible until it is
  - ▶ terminated by a  $)$  whose matching  $($  occurs to the left of  $\lambda$ ,  
OR
  - ▶ terminated by the end of the term.

# Conventions

- ▶  $\lambda x.t_1 t_2 t_3$  is an abbreviation for  $\lambda x.(t_1 t_2 t_3)$ , i.e., the scope of  $x$  is as far to the right as possible until it is
  - ▶ terminated by a  $)$  whose matching  $($  occurs to the left of  $\lambda$ ,  
OR
  - ▶ terminated by the end of the term.
- ▶ Applications associate to the left:  $t_1 t_2 t_3$  to be read as  $(t_1 t_2) t_3$  and not as  $t_1 (t_2 t_3)$

# Conventions

- ▶  $\lambda x.t_1 t_2 t_3$  is an abbreviation for  $\lambda x.(t_1 t_2 t_3)$ , i.e., the scope of  $x$  is as far to the right as possible until it is
  - ▶ terminated by a  $)$  whose matching  $($  occurs to the left of  $\lambda$ ,  
OR
  - ▶ terminated by the end of the term.
- ▶ Applications associate to the left:  $t_1 t_2 t_3$  to be read as  $(t_1 t_2) t_3$  and not as  $t_1 (t_2 t_3)$
- ▶  $\lambda x y z.t$  is an abbreviation for  $\lambda x \lambda y \lambda z.t$  which in turn is abbreviation for  $\lambda x.(\lambda y.(\lambda z.t))$ .

## $\alpha$ -renaming

- ▶ The name of a bound variable has no meaning except for its use to identify the bounding  $\lambda$ .

# $\alpha$ -renaming

- ▶ The name of a bound variable has no meaning except for its use to identify the bounding  $\lambda$ .
- ▶ Renaming a  $\lambda$  variable, including all its bound occurrences, does not change the meaning of an expression. For example,  $\lambda x.x \ x \ y$  is equivalent to  $\lambda u.u \ u \ y$

# $\alpha$ -renaming

- ▶ The name of a bound variable has no meaning except for its use to identify the bounding  $\lambda$ .
- ▶ Renaming a  $\lambda$  variable, including all its bound occurrences, does not change the meaning of an expression. For example,  $\lambda x.x \ x \ y$  is equivalent to  $\lambda u.u \ u \ y$ 
  - ▶ But it is not same as  $\lambda x.x \ x \ w$

# $\alpha$ -renaming

- ▶ The name of a bound variable has no meaning except for its use to identify the bounding  $\lambda$ .
- ▶ Renaming a  $\lambda$  variable, including all its bound occurrences, does not change the meaning of an expression. For example,  $\lambda x.x\ x\ y$  is equivalent to  $\lambda u.u\ u\ y$ 
  - ▶ But it is not same as  $\lambda x.x\ x\ w$
  - ▶ Can not change free variables!

## $\beta$ -reduction (Execution Semantics)

- ▶ if an abstraction  $\lambda x.t_1$  is applied to a term  $t_2$  then the result of the application is



# $\beta$ -reduction (Execution Semantics)

- ▶ if an abstraction  $\lambda x.t_1$  is applied to a term  $t_2$  then the result of the application is
  - ▶ the body of the abstraction  $t_1$  with all free occurrences of the formal parameter  $x$  replaced with  $t_2$ .

# $\beta$ -reduction (Execution Semantics)

- ▶ if an abstraction  $\lambda x.t_1$  is applied to a term  $t_2$  then the result of the application is
  - ▶ the body of the abstraction  $t_1$  with all free occurrences of the formal parameter  $x$  replaced with  $t_2$ .
- ▶ For example,

$$(\lambda f \lambda x. f (f x)) g \xrightarrow{\beta} \lambda x. g (g x)$$

# Caution

- ▶ During  $\beta$ -reduction, make sure a free variable is not captured inadvertently.

# Caution

- ▶ During  $\beta$ -reduction, make sure a free variable is not captured inadvertently.
- ▶ The following reduction is **WRONG**

$$(\lambda x \lambda y. x)(\lambda x. y) \xrightarrow{\beta} \lambda y. \lambda x. y$$

# Caution

- ▶ During  $\beta$ -reduction, make sure a free variable is not captured inadvertently.
- ▶ The following reduction is **WRONG**

$$(\lambda x \lambda y. x)(\lambda x. y) \xrightarrow{\beta} \lambda y. \lambda x. y$$

- ▶ Use  $\alpha$ -renaming to avoid variable capture

$$(\lambda x \lambda y. x)(\lambda x. y) \xrightarrow{\alpha} (\lambda u \lambda v. u)(\lambda x. y) \xrightarrow{\beta} \lambda v. \lambda x. y$$

# Exercise

► Apply  $\beta$ -reduction as far as possible

1.  $(\lambda x y z. x z (y z)) (\lambda x y. x) (\lambda y. y)$

2.  $(\lambda x. x x)(\lambda x. x x)$

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

# Church-Rosser Theorem

- ▶ Multiple ways to apply  $\beta$ -reduction

# Church-Rosser Theorem

- ▶ Multiple ways to apply  $\beta$ -reduction
- ▶ Some may not terminate



# Church-Rosser Theorem

- ▶ Multiple ways to apply  $\beta$ -reduction
- ▶ Some may not terminate
- ▶ However, if two different reduction sequences terminate then they always terminate in the same term

# Church-Rosser Theorem

- ▶ Multiple ways to apply  $\beta$ -reduction
- ▶ Some may not terminate
- ▶ However, if two different reduction sequences terminate then they always terminate in the same term
  - ▶ Also called the *Diamond Property*

# Church-Rosser Theorem

- ▶ Multiple ways to apply  $\beta$ -reduction
- ▶ Some may not terminate
- ▶ However, if two different reduction sequences terminate then they always terminate in the same term
  - ▶ Also called the *Diamond Property*
- ▶ Leftmost, outermost reduction will find the normal form if it exists

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans
- ▶ Complex Types?



# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans
- ▶ Complex Types?
  - ▶ Lists

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans
- ▶ Complex Types?
  - ▶ Lists
  - ▶ Arrays

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans
- ▶ Complex Types?
  - ▶ Lists
  - ▶ Arrays
- ▶ Don't we need data?

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans
- ▶ Complex Types?
  - ▶ Lists
  - ▶ Arrays
- ▶ Don't we need data?

# Programming in $\lambda$ Calculus

- ▶ Where is the other stuff?
- ▶ Constants?
  - ▶ Numbers
  - ▶ Booleans
- ▶ Complex Types?
  - ▶ Lists
  - ▶ Arrays
- ▶ Don't we need data?

Abstractions act as functions as well as data!

# Numbers: Church Numerals

- ▶ We need a “Zero”

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count



# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count
  - ▶ “Presence of item”

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count
  - ▶ “Presence of item”
- ▶ Intuition: Whiteboard and Marker

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count
  - ▶ “Presence of item”
- ▶ Intuition: Whiteboard and Marker
  - ▶ Blank board represents Zero

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count
  - ▶ “Presence of item”
- ▶ Intuition: Whiteboard and Marker
  - ▶ Blank board represents Zero
  - ▶ Each mark by marker represents a count.

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count
  - ▶ “Presence of item”
- ▶ Intuition: Whiteboard and Marker
  - ▶ Blank board represents Zero
  - ▶ Each mark by marker represents a count.
  - ▶ However, other pairs of objects will work as well

# Numbers: Church Numerals

- ▶ We need a “Zero”
  - ▶ “Absence of item”
- ▶ And something to count
  - ▶ “Presence of item”
- ▶ Intuition: Whiteboard and Marker
  - ▶ Blank board represents Zero
  - ▶ Each mark by marker represents a count.
  - ▶ However, other pairs of objects will work as well
- ▶ Lets translate this intuition into  $\lambda$ -expressions

# Numbers

► Zero =  $\lambda m w. w$

# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard



# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard
- ▶ One =  $\lambda m w. m w$

# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard
- ▶ One =  $\lambda m w. m w$ 
  - ▶ One mark on the whiteboard

# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard
- ▶ One =  $\lambda m w. m w$ 
  - ▶ One mark on the whiteboard
- ▶ Two =  $\lambda m w. m (m w)$

# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard
- ▶ One =  $\lambda m w. m w$ 
  - ▶ One mark on the whiteboard
- ▶ Two =  $\lambda m w. m (m w)$
- ▶ ...

# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard
- ▶ One =  $\lambda m w. m w$ 
  - ▶ One mark on the whiteboard
- ▶ Two =  $\lambda m w. m (m w)$
- ▶ ...
- ▶ What about operations?

# Numbers

- ▶ Zero =  $\lambda m w. w$ 
  - ▶ No mark on the whiteboard
- ▶ One =  $\lambda m w. m w$ 
  - ▶ One mark on the whiteboard
- ▶ Two =  $\lambda m w. m (m w)$
- ▶ ...
- ▶ What about operations?
  - ▶ add, multiply, subtract, divide, ... ?

# Operations on Numbers

►  $\text{succ} = \lambda x\ m\ w. m\ (x\ m\ w)$

# Operations on Numbers

- ▶  $\text{succ} = \lambda x\ m\ w. m\ (x\ m\ w)$ 
  - ▶ Verify:  $\text{succ}\ N = N + 1$



# Operations on Numbers

- ▶  $\text{succ} = \lambda x\ m\ w. m\ (x\ m\ w)$ 
  - ▶ Verify:  $\text{succ}\ N = N + 1$
- ▶  $\text{add} = \lambda x\ y\ m\ w. x\ m\ (y\ m\ w)$

# Operations on Numbers

- ▶  $\text{succ} = \lambda x\ m\ w. m\ (x\ m\ w)$ 
  - ▶ Verify:  $\text{succ}\ N = N + 1$
- ▶  $\text{add} = \lambda x\ y\ m\ w. x\ m\ (y\ m\ w)$ 
  - ▶ Verify:  $\text{add}\ M\ N = M + N$

# Operations on Numbers

- ▶  $\text{succ} = \lambda x\ m\ w. m\ (x\ m\ w)$ 
  - ▶ Verify:  $\text{succ}\ N = N + 1$
- ▶  $\text{add} = \lambda x\ y\ m\ w. x\ m\ (y\ m\ w)$ 
  - ▶ Verify:  $\text{add}\ M\ N = M + N$
- ▶  $\text{mult} = \lambda x\ y\ m\ w. x\ (y\ m)\ w$

# Operations on Numbers

- ▶  $\text{succ} = \lambda x\ m\ w. m\ (x\ m\ w)$ 
  - ▶ Verify:  $\text{succ}\ N = N + 1$
- ▶  $\text{add} = \lambda x\ y\ m\ w. x\ m\ (y\ m\ w)$ 
  - ▶ Verify:  $\text{add}\ M\ N = M + N$
- ▶  $\text{mult} = \lambda x\ y\ m\ w. x\ (y\ m)\ w$ 
  - ▶ Verify:  $\text{mult}\ M\ N = M * N$

# More Operations

►  $\text{pred} = \lambda x\ m\ w. x\ (\lambda g\ h. h\ (g\ m))(\lambda u. w)(\lambda u. u)$

# More Operations

- ▶  $\text{pred} = \lambda x\ m\ w. x\ (\lambda g\ h. h\ (g\ m))(\lambda u. w)(\lambda u. u)$ 
  - ▶ Verify:  $\text{pred}\ N = N - 1$

# More Operations

- ▶  $\text{pred} = \lambda x\ m\ w. x\ (\lambda g\ h. h\ (g\ m))(\lambda u. w)(\lambda u. u)$ 
  - ▶ Verify:  $\text{pred}\ N = N - 1$
- ▶  $\text{nminus} = \lambda x\ y. y\ \text{pred}\ x$

# More Operations

- ▶  $\text{pred} = \lambda x\ m\ w. x\ (\lambda g\ h. h\ (g\ m))(\lambda u. w)(\lambda u. u)$ 
  - ▶ Verify:  $\text{pred}\ N = N - 1$
- ▶  $\text{nminus} = \lambda x\ y. y\ \text{pred}\ x$ 
  - ▶ Verify:  $\text{nminus}\ M\ N = \max(0, M - N)$  – natural subtraction



# Church Booleans

- ▶ True and False

# Church Booleans

- ▶ True and False
- ▶ Intuition: Selection of one out of two (complementary) choices

# Church Booleans

- ▶ True and False
- ▶ Intuition: Selection of one out of two (complementary) choices
- ▶  $\text{True} = \lambda x y. x$

# Church Booleans

- ▶ True and False
- ▶ Intuition: Selection of one out of two (complementary) choices
- ▶  $\text{True} = \lambda x y. x$
- ▶  $\text{False} = \lambda x y. y$

# Church Booleans

- ▶ True and False
- ▶ Intuition: Selection of one out of two (complementary) choices
- ▶  $\text{True} = \lambda x y. x$
- ▶  $\text{False} = \lambda x y. y$
- ▶ Predicate:

# Church Booleans

- ▶ True and False
- ▶ Intuition: Selection of one out of two (complementary) choices
- ▶  $\text{True} = \lambda x y. x$
- ▶  $\text{False} = \lambda x y. y$
- ▶ Predicate:
  - ▶  $\text{isZero} = \lambda x. x (\lambda u. \text{False}) \text{True}$

# Operations on Booleans

- ▶ Logical operations

*and* =  $\lambda p\ q. p\ q\ p$

*or* =  $\lambda p\ q. p\ p\ q$

*not* =  $\lambda p\ t\ f. p\ f\ t$

# Operations on Booleans

- ▶ Logical operations

$$\text{and} = \lambda p\ q. p\ q\ p$$

$$\text{or} = \lambda p\ q. p\ p\ q$$

$$\text{not} = \lambda p\ t\ f. p\ f\ t$$

- ▶ The conditional operator *if*

$$\text{if} = \lambda c\ e_t\ e_f. (c\ e_t\ e_f)$$



# Operations on Booleans

- ▶ Logical operations

$$\text{and} = \lambda p q. p q p$$

$$\text{or} = \lambda p q. p p q$$

$$\text{not} = \lambda p t f. p f t$$

- ▶ The conditional operator *if*

- ▶ *if*  $c e_t e_f$  reduces to  $e_t$  if  $c$  is True, and to  $e_f$  if  $c$  is False

$$\text{if} = \lambda c e_t e_f. (c e_t e_f)$$

# More...

- ▶ More such types can be found at  
[https://en.wikipedia.org/wiki/Church\\_encoding](https://en.wikipedia.org/wiki/Church_encoding)

## More...

- ▶ More such types can be found at  
`https://en.wikipedia.org/wiki/Church\_encoding`
- ▶ It is fun to come up with your own definitions for constants and operations over different types

## More...

- ▶ More such types can be found at  
`https://en.wikipedia.org/wiki/Church\_encoding`
- ▶ It is fun to come up with your own definitions for constants and operations over different types
- ▶ or to develop understanding for existing definitions.

# We are missing something!!

- ▶ The machinery described so far does not allow us to define Recursive functions
  - ▶ Factorial, Fibonacci, ...
- ▶ There is no concept of “named” functions
  - ▶ So no way to refer to a function “recursively”!
- ▶ Fix-point computation comes to rescue

# Fix-point and $Y$ -combinator

- ▶ A fix-point of a function  $f$  is a value  $p$  such that  $f\ p = p$

# Fix-point and $Y$ -combinator

- ▶ A fix-point of a function  $f$  is a value  $p$  such that  $f\ p = p$
- ▶ Assume existence of a magic expression, called  $Y$ -combinator, that when applied to a  $\lambda$ -expression, gives its fixed point

$$Y\ f = f\ (Y\ f)$$

# Fix-point and $Y$ -combinator

- ▶ A fix-point of a function  $f$  is a value  $p$  such that  $f\ p = p$
- ▶ Assume existence of a magic expression, called  $Y$ -combinator, that when applied to a  $\lambda$ -expression, gives its fixed point

$$Y\ f = f\ (Y\ f)$$

- ▶  $Y$ -combinator gives us a way to apply a function recursively



## Recursion Example: Factorial

```
fact =  $\lambda n.$  if (isZero  $n$ ) One (mult  $n$  (fact (pred  $n$ )))  
      = ( $\lambda f n.$  if (isZero  $n$ ) One (mult  $n$  ( $f$  (pred  $n$ )))) fact
```

## Recursion Example: Factorial

$$\begin{aligned}\text{fact} &= \lambda n. \text{if } (\text{isZero } n) \text{ One } (\text{mult } n (\text{fact } (\text{pred } n))) \\ &= (\lambda f n. \text{if } (\text{isZero } n) \text{ One } (\text{mult } n (f (\text{pred } n)))) \text{ fact}\end{aligned}$$
$$\text{fact} = g \text{ fact}$$

- fact is a fixed point of the function

$$g = (\lambda f n. \text{if } (\text{isZero } n) \text{ One } (\text{mult } n (f (\text{pred } n))))$$

## Recursion Example: Factorial

$$\begin{aligned}\text{fact} &= \lambda n. \text{if } (\text{isZero } n) \text{ One } (\text{mult } n (\text{fact } (\text{pred } n))) \\ &= (\lambda f n. \text{if } (\text{isZero } n) \text{ One } (\text{mult } n (f (\text{pred } n)))) \text{ fact}\end{aligned}$$
$$\text{fact} = g \text{ fact}$$

- ▶ fact is a fixed point of the function

$$g = (\lambda f n. \text{if } (\text{isZero } n) \text{ One } (\text{mult } n (f (\text{pred } n))))$$

- ▶ Using Y-combinator,

$$\text{fact} = Y g$$

# Factorial: Verify

$$\text{fact } 2 = (Y \ g) \ 2$$

## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\ &= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator}\end{aligned}$$

## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\ &= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator} \\ &= (\lambda fn. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ (f \ (\text{pred } n)))) \ (Y \ g) \ 2\end{aligned}$$

## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\&= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator} \\&= (\lambda fn. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ (f \ (\text{pred } n)))) \ (Y \ g) \ 2 \\&= (\lambda n. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ ((Y \ g) \ (\text{pred } n)))) \ 2\end{aligned}$$

## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\&= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator} \\&= (\lambda fn. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ (f \ (\text{pred } n)))) \ (Y \ g) \ 2 \\&= (\lambda n. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ ((Y \ g) \ (\text{pred } n)))) \ 2 \\&= \text{if } (\text{isZero } 2) \ 1 \ (\text{mult } 2 \ ((Y \ g) (\text{pred} 2)))\end{aligned}$$



## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\&= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator} \\&= (\lambda fn. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ (f \ (\text{pred } n)))) \ (Y \ g) \ 2 \\&= (\lambda n. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ ((Y \ g) \ (\text{pred } n)))) \ 2 \\&= \text{if } (\text{isZero } 2) \ 1 \ (\text{mult } 2 \ ((Y \ g) \ (\text{pred } 2))) \\&= (\text{mult } 2 \ ((Y \ g) \ 1))\end{aligned}$$

## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\&= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator} \\&= (\lambda fn. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ (f \ (\text{pred } n)))) \ (Y \ g) \ 2 \\&= (\lambda n. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ ((Y \ g) \ (\text{pred } n)))) \ 2 \\&= \text{if } (\text{isZero } 2) \ 1 \ (\text{mult } 2 \ ((Y \ g) (\text{pred} 2))) \\&= (\text{mult } 2 \ ((Y \ g) \ 1)) \\&\quad \dots \\&= (\text{mult } 2 \ (\text{mult } 1 \ (\text{if } (\text{isZero } 0) \ 1 \ (\dots))))\end{aligned}$$

## Factorial: Verify

$\text{fact } 2 = (Y\ g)\ 2$   
=  $g\ (Y\ g)\ 2$  – by definition of Y-combinator  
=  $(\lambda fn. \text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n))))\ (Y\ g)\ 2$   
=  $(\lambda n. \text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ ((Y\ g)\ (\text{pred } n))))\ 2$   
=  $\text{if } (\text{isZero } 2)\ 1\ (\text{mult } 2\ ((Y\ g)\ (\text{pred } 2)))$   
=  $(\text{mult } 2\ ((Y\ g)\ 1))$   
...  
=  $(\text{mult } 2\ (\text{mult } 1\ (\text{if } (\text{isZero } 0)\ 1\ (...))))$   
=  $(\text{mult } 2\ (\text{mult } 1\ 1))$

## Factorial: Verify

$$\begin{aligned}\text{fact } 2 &= (Y \ g) \ 2 \\&= g \ (Y \ g) \ 2 \quad \text{-- by definition of Y-combinator} \\&= (\lambda fn. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ (f \ (\text{pred } n)))) \ (Y \ g) \ 2 \\&= (\lambda n. \text{if } (\text{isZero } n) \ 1 \ (\text{mult } n \ ((Y \ g) \ (\text{pred } n)))) \ 2 \\&= \text{if } (\text{isZero } 2) \ 1 \ (\text{mult } 2 \ ((Y \ g) \ (\text{pred } 2))) \\&= (\text{mult } 2 \ ((Y \ g) \ 1)) \\&\quad \dots \\&= (\text{mult } 2 \ (\text{mult } 1 \ (\text{if } (\text{isZero } 0) \ 1 \ (\dots)))) \\&= (\text{mult } 2 \ (\text{mult } 1 \ 1)) \\&= 2\end{aligned}$$

# Recursion and $Y$ -combinator

- ▶  $Y$ -combinator allows to unroll the body of loop once—similar to one unfolding of recursive call

# Recursion and $Y$ -combinator

- ▶  $Y$ -combinator allows to unroll the body of loop once—similar to one unfolding of recursive call
- ▶ Sequence of  $Y$ -combinator applications allow complete unfolding of recursive calls

# Recursion and $Y$ -combinator

- ▶  $Y$ -combinator allows to unroll the body of loop once—similar to one unfolding of recursive call
- ▶ Sequence of  $Y$ -combinator applications allow complete unfolding of recursive calls

# Recursion and $Y$ -combinator

- ▶  $Y$ -combinator allows to unroll the body of loop once—similar to one unfolding of recursive call
- ▶ Sequence of  $Y$ -combinator applications allow complete unfolding of recursive calls

BUT, what about the existence of  $Y$ -combinator?



# Y-combinators

- ▶ Many candidates exist

$$Y_1 = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

# Y-combinators

- ▶ Many candidates exist

$$Y_1 = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$Y = \lambda abcdefghijklmnopqstuvwxyzr.r(\textit{thisisafixedpointcombinator})$$

# Y-combinators

- ▶ Many candidates exist

$$Y_1 = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$Y = \lambda abcdefghijklmnopqstuvwxyzr.r(\textit{thisisafixedpointcombinator})$$

$$Y_{\text{funny}} = \textit{TTTTT TTTTT TTTTT TTTTT TTTTT T}$$

# Y-combinators

- ▶ Many candidates exist

$$Y_1 = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$Y = \lambda abcdefghijklmnopqstuvwxyzr.r(\textit{thisisafixedpointcombinator})$$

$$Y_{\text{funny}} = \textit{TTTTT TTTTT TTTTT TTTTT TTTTT T}$$

- ▶ Verify that  $(Y f) = f (Y f)$  for each

# Summary

- ▶ A cursory look at  $\lambda$ -calculus

# Summary

- ▶ A cursory look at  $\lambda$ -calculus
- ▶ Functions are data, and Data are functions!

# Summary

- ▶ A cursory look at  $\lambda$ -calculus
- ▶ Functions are data, and Data are functions!
- ▶ Not covered but important to know: The power of  $\lambda$  calculus is equivalent to that of Turing Machine (“Church Turing Thesis”)