

# Functional Programming

## Lambda Calculus

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

# The Lambda Calculus

## What Wikipedia says

**Lambda calculus** (also written as  $\lambda$ -calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application [...]. It is a universal model of computation that can be used to simulate any Turing machine and was first introduced by mathematician Alonzo Church in the 1930s as part of his research [on] the foundations of mathematics.

# The Lambda Calculus

## What Wikipedia says

**Lambda calculus** (also written as  $\lambda$ -calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application [...]. It is a universal model of computation that can be used to simulate any Turing machine and was first introduced by mathematician Alonzo Church in the 1930s as part of his research [on] the foundations of mathematics.

## Further down it says

- ✓ Lambda calculus has applications in many different areas in mathematics, philosophy, linguistics, and computer science.
- ✓ Lambda calculus has played an important role in the development of the theory of programming languages.
- ✗ Functional programming languages implement the lambda calculus.

# Syntax of the $\lambda$ -calculus

## $\lambda$ terms

$M, N ::= x$	variable
$  (\lambda x. M)$	(lambda) abstraction
$  (M N)$	application

- Variables are drawn from infinite denumerable set
- $(\lambda x. M)$  **binds**  $x$  in  $M$

# Syntax of the $\lambda$ -calculus

## $\lambda$ terms

$M, N ::= x$	variable
$  (\lambda x. M)$	(lambda) abstraction
$  (M N)$	application

- Variables are drawn from infinite denumerable set
- $(\lambda x. M)$  **binds**  $x$  in  $M$

## Conventions for omitting parentheses

- abstractions extend as far to the right as possible
- application is left associative

# Working with lambda terms

## Free and bound variables

$$\text{free}(x) = \{x\}$$

$$\text{free}(M N) = \text{free}(M) \cup \text{free}(N)$$

$$\text{free}(\lambda x.M) = \text{free}(M) \setminus \{x\}$$

$$\text{bound}(x) = \emptyset$$

$$\text{bound}(M N) = \text{bound}(M) \cup \text{bound}(N)$$

$$\text{bound}(\lambda x.M) = \text{bound}(M) \cup \{x\}$$

$$\text{var}(M) = \text{free}(M) \cup \text{bound}(M)$$

A lambda term  $M$  is **closed** ( $M$  is a **combinator**) iff  $\text{free}(M) = \emptyset$ .  
Otherwise the term is **open**.

# Working with lambda terms

## Substitution $M[x \mapsto N]$

$$x[x \mapsto N] = N$$

$$y[x \mapsto N] = y \quad x \neq y$$

$$(\lambda x.M)[x \mapsto N] := \lambda x.M$$

$$(\lambda y.M)[x \mapsto N] := \lambda y.(M[x \mapsto N]) \quad x \neq y, y \notin \text{free}(N)$$

$$(\lambda y.M)[x \mapsto N] := \lambda y'.(M[y \mapsto y'])[x \mapsto N] \quad x \neq y, y \in \text{free}(N), y' \notin \text{free}(M) \cup \text{free}(N)$$

$$(M M')[x \mapsto N] := (M[x \mapsto N])(M'[x \mapsto N])$$

## Guiding principle: **capture freedom**

In every  $(\lambda x.M)$  the bound variable  $x$  is “connected” to each free occurrence of  $x$  in  $M$ . These connections must not be broken by substitution.

# Computing with lambda terms

## Reduction rules

$(\lambda x.M) \rightarrow_\alpha (\lambda y.M[x \mapsto y])$	$y \notin \text{free}(M)$	Alpha reduction (renaming bound variables)
$((\lambda x.M) N) \rightarrow_\beta M[x \mapsto N]$		Beta reduction (function application)
$(\lambda x.(M x)) \rightarrow_\eta M$	$x \notin \text{free}(M)$	Eta reduction

Left hand side of a rule: **redex**; right hand side: **contractum**



# Computing with lambda terms

## Reduction rules

$(\lambda x.M) \rightarrow_\alpha (\lambda y.M[x \mapsto y]) \quad y \notin \text{free}(M)$  Alpha reduction (renaming bound variables)

$((\lambda x.M) N) \rightarrow_\beta M[x \mapsto N]$  Beta reduction (function application)

$(\lambda x.(M x)) \rightarrow_\eta M \quad x \notin \text{free}(M)$  Eta reduction

Left hand side of a rule: **redex**; right hand side: **contractum**

## Reductions may be applied anywhere in a term

$$\frac{M \rightarrow_x M'}{(\lambda y.M) \rightarrow_x (\lambda y.M')}$$

$$\frac{M \rightarrow_x M'}{(M N) \rightarrow_x (M' N)}$$

$$\frac{N \rightarrow_x N'}{(M N) \rightarrow_x (M N')}$$

# The theory of the lambda calculus

## Computation and equivalence

For  $x \subseteq \{\alpha, \beta, \gamma\}$  and reduction relation  $\rightarrow_x$ ,

- $\rightarrow_x^*$  is the reflexive-transitive closure,
- $\leftrightarrow_x$  is its symmetric closure,
- $\leftrightarrow_x^*$  is its reflexive-transitive-symmetric closure.

# The theory of the lambda calculus

## Computation and equivalence

For  $x \subseteq \{\alpha, \beta, \gamma\}$  and reduction relation  $\rightarrow_x$ ,

- $\rightarrow_x^*$  is the reflexive-transitive closure,
- $\leftrightarrow_x$  is its symmetric closure,
- $\leftrightarrow_x^*$  is its reflexive-transitive-symmetric closure.

## Equality in lambda calculus

- Alpha equivalence:  $M =_\alpha N$  iff  $M \leftrightarrow_\alpha^* N$ .
- Standard:  $M =_\beta N$  iff  $M \leftrightarrow_{\alpha, \beta}^* N$ .
- Extensional:  $M =_{\beta\eta} N$  iff  $M \leftrightarrow_{\alpha, \beta, \eta}^* N$ .

# Computing with lambda terms

## Definition: Normal form

Let  $M$  be a lambda term.

A lambda term  $N$  is a **normal form** of  $M$  iff  $M \xrightarrow{*}_{\beta} N$  and there is no  $N'$  with  $N \rightarrow_{\beta} N'$ .

# Computing with lambda terms

## Definition: Normal form

Let  $M$  be a lambda term.

A lambda term  $N$  is a **normal form** of  $M$  iff  $M \xrightarrow{*}_{\beta} N$  and there is no  $N'$  with  $N \rightarrow_{\beta} N'$ .

Lambda terms with equivalent (equal modulo  $\alpha$  reduction) normal forms exhibit the same behavior. The reverse is not always true.

# Computing with lambda terms

## Definition: Normal form

Let  $M$  be a lambda term.

A lambda term  $N$  is a **normal form** of  $M$  iff  $M \xrightarrow{*}_{\beta} N$  and there is no  $N'$  with  $N \rightarrow_{\beta} N'$ .

Lambda terms with equivalent (equal modulo  $\alpha$  reduction) normal forms exhibit the same behavior. The reverse is not always true.

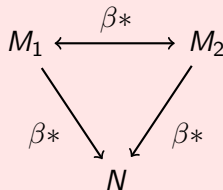
## A lambda term without normal form

$$(\lambda x. x \ x)(\lambda x. x \ x) \rightarrow_{\beta} (\lambda x. x \ x)(\lambda x. x \ x)$$

# Computing with lambda terms makes sense

## The Church-Rosser theorem

Beta reduction has the **Church-Rosser property**:

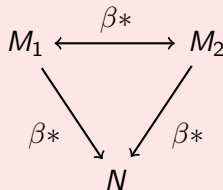


That is: For all  $M_1, M_2$  with  $M_1 \xleftrightarrow{*}_{\beta} M_2$ , there is some  $N$  with  $M_1 \xrightarrow{*}_{\beta} N$  and  $M_2 \xrightarrow{*}_{\beta} N$ .

# Computing with lambda terms makes sense

## The Church-Rosser theorem

Beta reduction has the **Church-Rosser property**:



That is: For all  $M_1, M_2$  with  $M_1 \overset{*}{\leftrightarrow}_{\beta} M_2$ , there is some  $N$  with  $M_1 \overset{*}{\rightarrow}_{\beta} N$  and  $M_2 \overset{*}{\rightarrow}_{\beta} N$ .

## Corollary

A lambda term  $M$  has at most one normal form modulo  $\alpha$  reduction.



# Programming in the pure lambda calculus

# From functions to arbitrary datatypes

Any computation may be encoded in the lambda calculus

- Booleans and conditionals
- Numbers
- Recursion
- Products (pairs)
- Variants

# Booleans and conditional

## Requirements / Specification

Wanted: Lambda terms *IF*, *TRUE*, *FALSE* such that

- $IF\ TRUE\ M\ N \xrightarrow{\beta}^* M$
- $IF\ FALSE\ M\ N \xrightarrow{\beta}^* N$

# Booleans and conditional

## Requirements / Specification

Wanted: Lambda terms *IF*, *TRUE*, *FALSE* such that

- $IF\ TRUE\ M\ N \xrightarrow{*}_{\beta} M$
- $IF\ FALSE\ M\ N \xrightarrow{*}_{\beta} N$

## Idea

*TRUE* and *FALSE* are functions that select the first or second argument, respectively

# Booleans and conditional

## Booleans

$$TRUE = \lambda x. \lambda y. x$$

$$FALSE = \lambda x. \lambda y. y$$

# Booleans and conditional

## Booleans

$$TRUE = \lambda x. \lambda y. x$$

$$FALSE = \lambda x. \lambda y. y$$

## Conditional

$$IF = \lambda b. \lambda t. \lambda f. b \ t \ f$$

# Booleans and conditional

## Booleans

$$TRUE = \lambda x. \lambda y. x$$

$$FALSE = \lambda x. \lambda y. y$$

## Conditional

$$IF = \lambda b. \lambda t. \lambda f. b \ t \ f$$

Check the spec!

...

# Natural numbers

## Requirements / Specification

Wanted: A family of lambda terms  $\lceil n \rceil$ , for each  $n \in \mathbf{N}$ , such that the arithmetic operations are *lambda definable*.

That is, there are lambda terms  $ADD$ ,  $SUB$ ,  $MULT$ ,  $DIV$  such that

- $ADD \lceil m \rceil \lceil n \rceil \xrightarrow{*}_{\beta} \lceil m + n \rceil$
- $SUB \lceil m \rceil \lceil n \rceil \xrightarrow{*}_{\beta} \lceil m - n \rceil$
- $MULT \lceil m \rceil \lceil n \rceil \xrightarrow{*}_{\beta} \lceil mn \rceil$
- $DIV \lceil m \rceil \lceil n \rceil \xrightarrow{*}_{\beta} \lceil m/n \rceil$



# Church numerals

## One approach

The **Church numeral**  $\lceil n \rceil$  of some natural number  $n$  is a function that takes two parameters, a function  $f$  and some  $x$ , and applies  $f$   $n$ -times to  $x$ .

# Church numerals

## One approach

The **Church numeral**  $[n]$  of some natural number  $n$  is a function that takes two parameters, a function  $f$  and some  $x$ , and applies  $f$   $n$ -times to  $x$ .

## Zero

$$[0] = \lambda f. \lambda x. x$$

# Church numerals

## One approach

The **Church numeral**  $[n]$  of some natural number  $n$  is a function that takes two parameters, a function  $f$  and some  $x$ , and applies  $f$   $n$ -times to  $x$ .

## Zero

$$[0] = \lambda f. \lambda x. x$$

## Successor

$$SUCC = \lambda n. \lambda f. \lambda x. f(n f x)$$

# Church numerals — addition and multiplication

## Addition

$$ADD = \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$$

# Church numerals — addition and multiplication

## Addition

$$ADD = \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$$

## Multiplication

$$MULT = \lambda m. \lambda n. \lambda f. \lambda x. m (n f) x$$

## Church numerals — conditional

### Wanted

*IF0* such that

- $IF0\ [0]\ M\ N \xrightarrow{\beta}^* M$
- $IF0\ [n]\ M\ N \xrightarrow{\beta}^* N$  if  $n \neq 0$

# Church numerals — conditional

## Wanted

*IF0* such that

- $IF0\ [0]\ M\ N \xrightarrow{*}_{\beta} M$
- $IF0\ [n]\ M\ N \xrightarrow{*}_{\beta} N$  if  $n \neq 0$

## Testing for zero

$$IF0 = \lambda n. \lambda z. \lambda s. n (\lambda x. s) z$$

# Church numerals — conditional

## Wanted

*IF0* such that

- $IF0\ [0]\ M\ N \xrightarrow{*}_{\beta} M$
- $IF0\ [n]\ M\ N \xrightarrow{*}_{\beta} N$  if  $n \neq 0$

## Testing for zero

$$IF0 = \lambda n. \lambda z. \lambda s. n (\lambda x. s) z$$

## Check the spec!

...



# Pairs

## Specification

Wanted: lambda terms *PAIR*, *FST*, *SND* such that

- $FST(PAIR\ M\ N) \xrightarrow{*}_{\beta} M$
- $SND(PAIR\ M\ N) \xrightarrow{*}_{\beta} N$

## Specification

Wanted: lambda terms *PAIR*, *FST*, *SND* such that

- $FST(PAIR\ M\ N) \xrightarrow{*}_{\beta} M$
- $SND(PAIR\ M\ N) \xrightarrow{*}_{\beta} N$

## Implementation

$$PAIR = \lambda x. \lambda y. \lambda v. v\ x\ y$$

$$FST = \lambda p. p(\lambda x. \lambda y. x)$$

$$SND = \lambda p. p(\lambda x. \lambda y. y)$$

## Variants (data Either a b = Left a | Right b)

### Specification

Wanted: lambda terms *LEFT*, *RIGHT*, *CASE* such that

- $CASE(LEFT\ M)N_l\ N_r \xrightarrow{\beta}^* N_l\ M$
- $CASE(RIGHT\ M)N_l\ N_r \xrightarrow{\beta}^* N_r\ M$

## Variants (data Either a b = Left a | Right b)

### Specification

Wanted: lambda terms *LEFT*, *RIGHT*, *CASE* such that

- $CASE(LEFT\ M)N_l\ N_r \xrightarrow{*}_\beta N_l\ M$
- $CASE(RIGHT\ M)N_l\ N_r \xrightarrow{*}_\beta N_r\ M$

### Implementation

*CASE* =  
*LEFT* =  
*RIGHT* =

# Constructor and case for Haskell data

## Scott encoding of data types

Suppose a datatype  $D$  is defined with constructors  $K_1, \dots, K_m$  where constructor  $j$  takes  $n_j$  arguments.

$$\begin{aligned}[K_j] &= \lambda x_1 \dots x_{n_j}. \lambda c_1 \dots c_m. c_j x_1 \dots x_{n_j} \\ CASE_D &= \lambda v. \lambda c_1 \dots c_m. v c_1 \dots c_m \\ &=_{\eta} \lambda v. v\end{aligned}$$

That is, the encoding of the constructor **is** the  $CASE_D$  operation.

- Pair is the special case with one constructor and two arguments
- Either is the special case with two constructors of one argument each

# Recursion

## Fixpoint theorem (see Barendregt, The Lambda Calculus)

Every lambda term has a fixpoint:

For every  $M$  there is some  $N$  such that  $M N \stackrel{*}{\leftrightarrow}_{\beta} N$ .

# Recursion

## Fixpoint theorem (see Barendregt, The Lambda Calculus)

Every lambda term has a fixpoint:

For every  $M$  there is some  $N$  such that  $M N \overset{*}{\leftrightarrow}_{\beta} N$ .

## Proof

Let  $N = Y M$  where

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)).$$

# Recursion

## Fixpoint theorem (see Barendregt, The Lambda Calculus)

Every lambda term has a fixpoint:

For every  $M$  there is some  $N$  such that  $M N \xrightarrow{*}_{\beta} N$ .

## Proof

Let  $N = Y M$  where

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)).$$

## Remark

$Y$  is Curry's **fixpoint combinator**. There are infinitely many more fixpoint combinators with various properties.



# Wrapup

- Beta reduction is the only computation rule of lambda calculus
- It applies anywhere in a lambda term
- All datatypes can be expressed in lambda calculus
- Lambda calculus is able to express the primitives of the theory of partial recursive functions
- The theory of partial recursive functions is Turing complete
- Hence is the (untyped) lambda calculus