## Lambda Calculus 1

Sven Tennie

July 15, 2018

### Lambda Calculus

- ▶ Invented by Alonzo Church (1920s)
- Equally expressive to the Turing Machine(s)
- Formal Language
- Computational Model
  - ► Lisp (1950s)
  - ML
  - Haskell
- "Lambda Expressions" in almost every modern programming language

## Why should I care?

- Simple Computational Model
  - to describe structure and behaviour (E.g. Operational Semantics)
  - to reason and proove

# Why should I care?

- Simple Computational Model
  - to describe structure and behaviour (E.g. Operational Semantics)
  - ▶ to reason and proove
- Explains why things in FP are like they are
  - pure functions
  - higher-order functions
  - currying
  - lazy evaluation

## Why should I care?

- Simple Computational Model
  - to describe structure and behaviour (E.g. Operational Semantics)
  - to reason and proove
- Explains why things in FP are like they are
  - pure functions
  - higher-order functions
  - currying
  - lazy evaluation
- Understand FP Compilers
  - Good starting-point when you want to introduce FP stuff into other languages
  - Good base when you what to write your own compiler
  - ▶ GHC uses an enriched Lambda Calculus internally

# Untyped Lambda Calculus

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

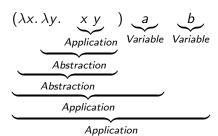
# Untyped Lambda Calculus

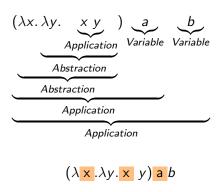
$$t := x$$
 Variable  $\lambda x.t$  Abstraction  $t \ t$  Application

### Example

Identity

$$\underbrace{\frac{\lambda x.x}{\text{Abstraction}} \underbrace{y}_{\text{Variable}} \rightarrow y}_{\text{Application}}$$





$$(\lambda x. \lambda y. \underbrace{x \ y}_{Application}) \underbrace{\begin{array}{c} a \\ b \\ Variable \end{array}}_{Variable} Variable$$

$$\underbrace{\begin{array}{c} Abstraction \\ Application \\ Application \\ \end{array}}_{Application}$$

$$\underbrace{\begin{array}{c} (\lambda x. \lambda y. x \ y) \ a \ b \\ \rightarrow (\lambda y. a \ y) \ b \\ \end{array}}_{D}$$

$$(\lambda x. \lambda y. \quad x \quad y) \quad a \quad b$$

$$Application \quad Variable \quad Variable$$

$$Abstraction$$

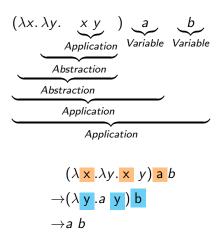
$$Application$$

$$Application$$

$$(\lambda x. \lambda y. x \quad y) \quad a \quad b$$

$$\rightarrow (\lambda y. a \quad y) \quad b$$

$$\rightarrow a \quad b$$



- Remarks
  - ▶ Lambda Expressions expand as much to the right as possible
  - ▶ We use parentheses to clearify what's meant
    - ► Even though I didn't specify them in the grammar ...



#### Full Beta-Reduction

- ► RedEx
  - Reducible Expression
  - Always an Application

$$\underbrace{(\lambda x.x) \; ((\lambda x.x) \; (\lambda z. \underbrace{(\lambda x.x) \; z}))}_{RedEx}$$

$$\underbrace{RedEx}_{RedEx}$$

- Full Beta-Reduction
  - Any RedEx, Any Time
  - ▶ Like in Arithmetics
  - ► Too fuzzy to program...
    - How to write a good test if the next step could be several expressions?

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

- Normal Order Reduction
  - Left-most, Outer-most RedEx

$$(\lambda x.x) ((\lambda x.x) (\lambda z.(\lambda x.x) z))$$
  
$$\rightarrow (\lambda x.x) (\lambda z.(\lambda x.x) z)$$

- Normal Order Reduction
  - Left-most, Outer-most RedEx

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

$$\rightarrow (\lambda x.x) (\lambda z.(\lambda x.x) z)$$

$$\rightarrow (\lambda z.(\lambda x.x) z)$$

- Normal Order Reduction
  - Left-most, Outer-most RedEx

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

$$\rightarrow (\lambda x.x) (\lambda z.(\lambda x.x) z)$$

$$\rightarrow (\lambda z.(\lambda x.x) z)$$

$$\rightarrow (\lambda z.z)$$

- Normal Order Reduction
  - Left-most, Outer-most RedEx

## Call-by-Name

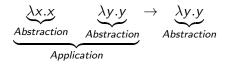
- ► Call-by-Name
  - ▶ lazy, non-strict
  - ► Save result -> Call-by-Need
  - ▶ No reduction inside Abstractions

## Call-by-Value

- ► Call-by-Value
  - ▶ eager, strict

## Higher Order Functions

- Functions that take or return functions
  - ► Are there "by definition"



# Currying

$$(\lambda x.\lambda y.xy)z \rightarrow \lambda y.zy$$

- Example
  - ▶ (+1) Section in Haskell

$$(\lambda x.\lambda y. + xy)1 \rightarrow \lambda y. + 1y$$

Partial Application is there "by definition"

#### Remarks

- Everything (Term) is an Expression
  - No statements
- ▶ No "destructive" Variable Assignments
  - ► The reason why FP Languages promote pure functions

# Some Vocabulary

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

- x is bound by the surrounding abstraction
- ▶ y is free
  - ▶ E.g. part of the environment

### Reductions and Conversions

Alpha conversion

$$\lambda x.x \rightarrow_{\alpha} \lambda y.y$$

### Reductions and Conversions

Alpha conversion

$$\lambda x.x \rightarrow_{\alpha} \lambda y.y$$

▶ Beta reduction

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

### Reductions and Conversions

► Alpha conversion

$$\lambda x.x \rightarrow_{\alpha} \lambda y.y$$

Beta reduction

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

- Eta conversion
  - iff (if and only if) x is not free in f

$$(\lambda x.f \ x) \to_{\eta} f$$
$$(\lambda x.(\lambda y.y) \ x) \to_{\eta} \lambda y.y$$

x is not free in f

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

## Church Encodings

- Encode Data into the Lambda Calculus
- ► To simplify our formulas, let's say that we have declarations

$$id \equiv \lambda x.x$$

id y 
$$\rightarrow$$
 y

### Booleans

$$true \equiv \lambda t. \lambda f. t$$
 $false \equiv \lambda t. \lambda f. f$ 

if  $\_$ then $\_$ else  $\equiv \lambda c.\lambda b_{true}.\lambda b_{false}.c$   $b_{true}$   $b_{false}$ 

### Example

$$if\_then\_else\ true\ a\ b$$
 $\equiv (\lambda c.\lambda b_{true}.\lambda b_{false}.c\ b_{true}\ b_{false})\ true\ a\ b$ 
 $\rightarrow true\ a\ b$ 
 $\equiv (\lambda t.\lambda f.t)\ a\ b$ 
 $\rightarrow (\lambda f.a)\ b$ 
 $\rightarrow a$ 

### And

$$true \equiv \lambda t. \lambda f. t$$
 $false \equiv \lambda t. \lambda f. f$ 

and 
$$\equiv \lambda p.\lambda q.p \ q \ p$$

#### Example

and true false  $\equiv (\lambda p.\lambda q.p \ q \ p) \ true \ false$   $\rightarrow (\lambda q.true \ q \ true) \ false$   $\rightarrow true false true$   $\equiv (\lambda t.\lambda f.t) \ false \ true$   $\rightarrow (\lambda f.false) true$   $\rightarrow false$ 



Or

 $\lambda p.\lambda q.ppq$ 

#### **Pairs**

$$pair \equiv \lambda x. \lambda y. \lambda z. z \times y$$
$$first \equiv (\lambda p. p)(\lambda x. \lambda y. x)$$
$$second \equiv (\lambda p. p)(\lambda x. \lambda y. y)$$

### Example

# Pair Example (continued)

$$\begin{array}{ll} \textit{pair}'_{ab} \equiv & \lambda z.z \ a \ b \\ \textit{first} \equiv & (\lambda p.p)(\lambda x.\lambda y.x) \\ \\ \textit{first pair}'_{ab} \equiv & (\lambda p.p)(\lambda x.\lambda y.x) \textit{pair}'_{ab} \\ \rightarrow & pair'_{ab}(\lambda x.\lambda y.x) \\ \equiv & (\lambda z.z \ a \ b)(\lambda x.\lambda y.x) \\ \rightarrow & (\lambda x.\lambda y.x) \ a \ b \\ \rightarrow & (\lambda y.a) \ b \\ \rightarrow & a \end{array}$$

#### Numerals

- Peano axioms
  - Every natural number can be defined with 0 and a successor function

$$0 \equiv \lambda f.\lambda x.x$$

$$1 \equiv \lambda f.\lambda x.f x$$

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

$$3 \equiv \lambda f.\lambda x.f (f (f x))$$

- Meaning
- 0 f is evaluated 0 times
- 1 f is evaluated once
- x can be every lambda term



## Numerals Example - Successor

$$0 \equiv \lambda f.\lambda x.x$$

$$1 \equiv \lambda f..f x$$

$$successor \equiv \lambda n.\lambda f.\lambda x.f (n f x)$$

$$successor 1 \equiv (\lambda n.\lambda f.\lambda x.f (n f x))1$$

$$\rightarrow \lambda f.\lambda x.f (1 f x)$$

$$\equiv \lambda f.\lambda x.f ((\lambda f.\lambda x.f x) f x)$$

$$to \lambda f.\lambda x.f ((\lambda x.f x) x)$$

$$to \lambda f.\lambda x.f (f x)$$

$$\equiv \lambda f.\lambda x.f (f x)$$

## Numerals Example - 0 + 0

$$0 \equiv \lambda f.\lambda x.x$$

$$plus \equiv \lambda m.\lambda n.\lambda f.\lambda x.m f(nfx)$$

$$plus 0 0 \equiv (\lambda m.\lambda n.\lambda f.\lambda x.m f(nfx)) 0 0$$

$$\rightarrow (\lambda n.\lambda f.\lambda x.0 f(nfx)) 0$$

$$\rightarrow (\lambda f.\lambda x.0 f(nfx))$$

$$\equiv (\lambda f.\lambda x.(\lambda f.\lambda x.x) f(0fx))$$

$$\rightarrow (\lambda f.\lambda x.(\lambda f.\lambda x.x) f(0fx))$$

$$\rightarrow (\lambda f.\lambda x.((\lambda x.x) f(0fx))$$

$$\equiv (\lambda f.\lambda x.((\lambda f.\lambda x.x) f(x)))$$

$$\rightarrow (\lambda f.\lambda x.((\lambda f.\lambda x.x) f(x)))$$

$$\rightarrow (\lambda f.\lambda x.((\lambda f.\lambda x.x) f(x)))$$

$$\rightarrow (\lambda f.\lambda x.x) f(x)$$

### Books

The implementation of programming languages Type Systems

#### **Thanks**

- ▶ Hope you enjoyed this talk and learned something new.
- ► Hope it wasn't too much math and dusty formulas . . . :)