

Untyped Lambda Calculus

Sven Tennie

July 27, 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 prove

Why should I care?

- Simple Computational Model
 - to describe structure and behaviour (E.g. Operational Semantics)
 - to reason and prove
- 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 prove
- Explains why things in FP are like they are
 - pure functions
 - higher-order functions
 - currying
 - lazy evaluation
- Understand FP Compilers
 - Introduce FP stuff into other languages
 - 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

Lambda Calculus

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

Untyped Lambda Calculus

$t ::= x$

Variable

$\lambda x. t$

Abstraction

$t \ t$

Application

Example

- Identity

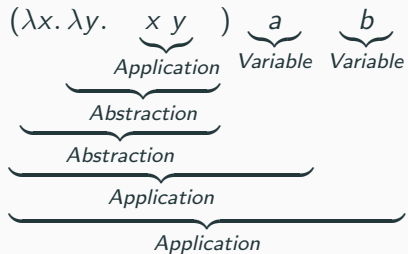
Lambda Calculus

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

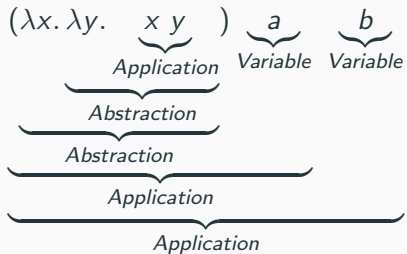
Javascript

$\underbrace{\underbrace{(\text{function } (x)\{\text{return } x; \})}_{\text{Abstraction}} \underbrace{(y)}_{\text{Variable}}}_{\text{Application}}$

Evaluation / Reduction

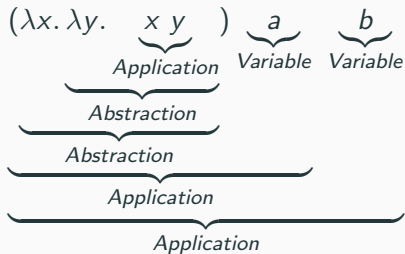


Evaluation / Reduction



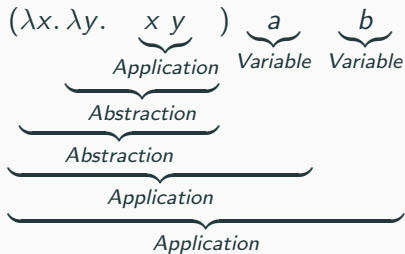
$(\lambda x. \lambda y. x y) a b$

Evaluation / Reduction

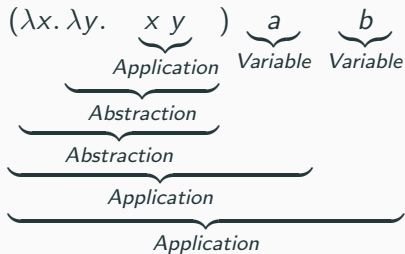


$$\begin{aligned} & (\lambda \textcolor{brown}{x}. \lambda y. \textcolor{brown}{x} \ y) \textcolor{brown}{a} \textcolor{brown}{b} \\ \rightarrow & (\lambda \textcolor{teal}{y}. \textcolor{teal}{a} \ \textcolor{teal}{y}) \textcolor{teal}{b} \end{aligned}$$

Evaluation / Reduction


$$\begin{aligned} & (\lambda \textcolor{brown}{x}. \lambda y. \textcolor{brown}{x} \ y) \textcolor{brown}{a} \textcolor{brown}{b} \\ \rightarrow & (\lambda \textcolor{blue}{y}. \textcolor{blue}{a} \ \textcolor{blue}{y}) \textcolor{blue}{b} \\ \rightarrow & \textcolor{brown}{a} \ \textcolor{brown}{b} \end{aligned}$$

Evaluation / Reduction


$$\begin{aligned} & (\lambda \textcolor{brown}{x}. \lambda y. \textcolor{brown}{x} \ y) \textcolor{brown}{a} \textcolor{brown}{b} \\ \rightarrow & (\lambda \textcolor{blue}{y}. \textcolor{blue}{a} \ \textcolor{blue}{y}) \textcolor{blue}{b} \\ \rightarrow & \textcolor{brown}{a} \ \textcolor{brown}{b} \end{aligned}$$

{Parentheses are not part of the grammar? See next slide :) }

Notational Conventions

- We use parentheses to clarify what's meant
- Applications associate to the left

$$s\ t\ u \equiv (s\ t)\ u$$

- Lambda Expressions expand as much to the right as possible

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

$\lambda x. \lambda y. x \ y \ z$

λy y is *bound*, x and z are *free*

λx x and y are *bound*, z is *free*

$\lambda x, \lambda y$ *binder*

A term with no free variables is "*closed*"

- A "*combinator*"
- $id \equiv \lambda x. x$

- We learned how to write down and talk about Lambda Calculus Terms
- How to evaluate them?
- Different Strategies
 - Interesting outcomes

Full Beta-Reduction

- RedEx
 - **R**educible **E**xpression
 - Always an Application

$$\begin{array}{c} (\lambda x.x) \ ((\lambda x.x) \ (\lambda z. \underbrace{(\lambda x.x) \ z}_{\text{RedEx}})) \\ \underbrace{\hspace{10em}}_{\text{RedEx}} \end{array}$$

Full Beta-Reduction

- RedEx
 - **R**educible **E**xpression
 - Always an Application

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

- 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

$$\underline{(\lambda x.x) ((\lambda x.x) (\lambda z.(\lambda x.x) z))}$$

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

$$\frac{(\lambda x.x) ((\lambda x.x) (\lambda z.(\lambda x.x) z))}{\rightarrow (\lambda x.x) (\lambda z.(\lambda x.x) z)}$$

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

$$\begin{array}{c} \underline{(\lambda x.x) ((\lambda x.x) (\lambda z.(\lambda x.x) z))} \\ \rightarrow \underline{(\lambda x.x) (\lambda z.(\lambda x.x) z)} \end{array}$$

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

Normal Order Reduction

$$\begin{aligned} & \underline{(\lambda x.x) ((\lambda x.x) (\lambda z.(\lambda x.x) z))} \\ \rightarrow & \underline{(\lambda x.x) (\lambda z.(\lambda x.x) z)} \\ \rightarrow & (\lambda z.(\lambda x.x) z) \\ \rightarrow & (\lambda z.z) \end{aligned}$$

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

- Call-by-Name
 - lazy, non-strict
 - Parameters are NOT evaluated before they are passed to Lambdas
 - Lambdas are values
 - Save result -> Call-by-Need
 - No reduction inside Abstractions

- Call-by-Value
 - eager, strict

Higher Order Functions

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

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

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

- Example
 - (+1) Section in Haskell

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

- Partial Application is there "by definition"

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

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

$$(\lambda x.f\ x) \rightarrow_{\eta} f$$

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

- x is not free in f

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

Translate Lambda Calculus to Javascript

Variable \rightarrow Variable Abstraction \rightarrow Function Declaration
Application \rightarrow Function Call

Church Encodings

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

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

$$id\ y \rightarrow y$$

$$true \equiv \lambda t.\lambda f.t$$
$$false \equiv \lambda t.\lambda f.f$$
$$if_then_else \equiv \lambda c.\lambda b_{true}.\lambda b_{false}.c\ b_{true}\ b_{false}$$

Example

$$\begin{aligned} & if_then_else\ true\ a\ b \\ & \equiv (\lambda c.\lambda b_{true}.\lambda b_{false}.c\ b_{true}\ b_{false})\ true\ a\ b \\ & \rightarrow true\ a\ b \\ & \equiv (\lambda t.\lambda f.t)\ a\ b \\ & \rightarrow (\lambda f.a)\ b \\ & \rightarrow a \end{aligned}$$

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

- Example

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

Or

$\lambda p. \lambda q. p p q$

$$\text{pair} \equiv \lambda x. \lambda y. \lambda z. z \ x \ y$$

$$\text{first} \equiv (\lambda p. p)(\lambda x. \lambda y. x)$$

$$\text{second} \equiv (\lambda p. p)(\lambda x. \lambda y. y)$$

Example

$$\begin{aligned} \text{pair}_{AB} &\equiv \text{pair} && a \ b \\ &\equiv && (\lambda x. \lambda y. \lambda z. z \ x \ y) \ a \ b \\ &\rightarrow && (\lambda y. \lambda z. z \ a \ y) b \\ &\rightarrow && \lambda z. z \ a \ b \\ &\equiv && \text{pair}'_{ab} \end{aligned}$$

Pair Example (continued)

$$pair'_{ab} \equiv \lambda z. z \ a \ b$$

$$first \equiv (\lambda p. p)(\lambda x. \lambda y. x)$$

$$first \ pair'_{ab} \equiv (\lambda p. p)(\lambda x. \lambda y. x) pair'_{ab}$$

$$\rightarrow pair'_{ab}(\lambda x. \lambda y. x)$$

$$\equiv (\lambda z. z \ a \ b)(\lambda x. \lambda y. x)$$

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

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

$$\rightarrow a$$

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

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

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

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

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

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

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

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

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

$\equiv 2$

Numerals Example - $0 + 0$

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

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

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

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

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

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

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

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

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

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

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

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 ... :)