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

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

Evaluation / Reduction

$$\underbrace{\underbrace{\underbrace{(\lambda x.\underbrace{\lambda y.\ \underbrace{x\ y}_{\textit{Application}}}_{\textit{Abstraction}})}_{\textit{Abstraction}}\ \underbrace{a}_{\textit{Variable}}}_{\textit{Application}}\ \underbrace{b}_{\textit{Variable}}}_{\textit{Application}}$$

Evaluation / Reduction

$$\underbrace{\underbrace{(\underbrace{\lambda x.\underbrace{\lambda y.\underbrace{x\ y}_{\textit{Application}}}_{\textit{Abstraction}}}_{\textit{Abstraction}})\ \underbrace{a}_{\textit{Variable}}}_{\textit{Application}}\ \underbrace{b}_{\textit{Variable}}}_{\textit{Application}}$$

$$(\lambda \mathsf{x}.\lambda y.\mathsf{x}\ y)\ \mathsf{a}\ b$$

Evaluation / Reduction

$$\underbrace{\underbrace{(\underbrace{\lambda x.\underbrace{\lambda y.\underbrace{x\ y}_{\textit{Application}}}_{\textit{Abstraction}}}_{\textit{Abstraction}})\ \underbrace{a}_{\textit{Variable}}}_{\textit{Application}}\ \underbrace{b}_{\textit{Variable}}}_{\textit{Application}}$$

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

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

Evaluation / Reduction

$$\underbrace{\underbrace{(\underbrace{\lambda x.\,\underbrace{\lambda y.\,\underbrace{x\ y}_{\textit{Application}}}_{\textit{Abstraction}}}_{\textit{Abstraction}})\ \underbrace{a}_{\textit{Variable}}}_{\textit{Application}}\ \underbrace{b}_{\textit{Variable}}}_{\textit{Application}}$$

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

Evaluation / Reduction

$$\underbrace{\underbrace{(\underbrace{\lambda x.\underbrace{\lambda y.\underbrace{x\ y}_{\textit{Application}}}_{\textit{Abstraction}}}_{\textit{Abstraction}})\ \underbrace{a}_{\textit{Variable}}}_{\textit{Application}}\ \underbrace{b}_{\textit{Variable}}}_{\textit{Application}}$$

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

- 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
  - **Red**ucible **Ex**pression
  - Always an Application

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

- 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?

Normal Order Reduction

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

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

Normal Order Reduction

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

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

Normal Order Reduction

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

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

Normal Order Reduction

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

- 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"

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

Currying

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

- Example
  - (+1) Section in Haskell

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

- 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

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

- x is not free in f

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

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

$$
\begin{aligned}
true &\equiv \lambda t.\lambda f.t \\
false &\equiv \lambda t.\lambda f.f
\end{aligned}
$$

$$
if\_then\_else \equiv \lambda c.\lambda b_{true}.\lambda b_{false}.c\ \ b_{true}\ \ b_{false}
$$

Example

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

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

$$\begin{aligned}
& and\ \ true\ \ false \\
\equiv & (\lambda p.\lambda q.p\ \ q\ \ p)\ \ true\ \ false \\
\rightarrow & (\lambda q.true\ \ q\ \ true)\ \ false \\
\rightarrow & truefalsetrue \\
\equiv & (\lambda t.\lambda f.t)\ \ false\ \ true \\
\rightarrow & (\lambda f.false) true \\
\rightarrow & false
\end{aligned}$$

Or

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

Pairs

$$
\begin{aligned}
pair &\equiv \lambda x.\lambda y.\lambda z.z\ x\ y \\
first &\equiv (\lambda p.p)(\lambda x.\lambda y.x) \\
second &\equiv (\lambda p.p)(\lambda x.\lambda y.y)
\end{aligned}
$$

Example

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

Pair Example (continued)

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

Numerals

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

$$
\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
1 &\equiv \lambda f.\lambda x.f\ x \\
2 &\equiv \lambda f.\lambda x.f\ (f\ x) \\
3 &\equiv \lambda f.\lambda x.f\ (f\ (f\ x))
\end{aligned}
$$

- Meaning

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

Numerals Example - Successor

$$
\begin{aligned}
0 &\equiv && \lambda f.\lambda x.x \\
1 &\equiv && \lambda f..f\ x \\
\\
successor &\equiv && \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \\
\\
successor 1 &\equiv && (\lambda n.\lambda f.\lambda x.f\ (n\ f\ x))1 \\
&\rightarrow && \lambda f.\lambda x.f\ (1\ f\ x) \\
&\equiv \lambda f.\lambda x.f\ ((\lambda f.\lambda x.f\ x)\ f\ x) \\
&to && \lambda f.\lambda x.f\ ((\lambda x.f\ x)\ x) \\
&to && \lambda f.\lambda x.f\ (f\ x) \\
&\equiv && 2
\end{aligned}
$$

Numerals Example - 0 + 0

$$\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
plus &\equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)
\end{aligned}$$

$$\begin{aligned}
plus\ 0\ 0 &\equiv (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))\ 0\ 0 \\
&\rightarrow (\lambda n.\lambda f.\lambda x.0f(nfx))\ 0 \\
&\rightarrow (\lambda f.\lambda x.0f(0fx)) \\
&\equiv (\lambda f.\lambda x.(\lambda f.\lambda x.x)f(0fx)) \\
&\rightarrow (\lambda f.\lambda x.(\lambda x.x)(0fx)) \\
&\rightarrow (\lambda f.\lambda x.(0fx)) \\
&\equiv (\lambda f.\lambda x.((\lambda f.\lambda x.x)fx)) \\
&\rightarrow (\lambda f.\lambda x.((\lambda x.x)x)) \\
&\rightarrow (\lambda f.\lambda x.x \\
&\equiv 0
\end{aligned}$$

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 . . . :)