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 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
  - Introduce FP stuff into other languages
  - Write your own compiler
  - GHC uses an enriched Lambda Calculus internally

Untyped Lambda Calculus

$$
\begin{aligned}
t ::= {} & x && \text{Variable} \\
& \lambda x.t && \text{Abstraction} \\
& t\ t && \text{Application}
\end{aligned}
$$

Untyped Lambda Calculus

$$
\begin{aligned}
t ::= {} & x && \text{Variable} \\
& \lambda x.t && \text{Abstraction} \\
& t\ t && \text{Application}
\end{aligned}
$$

Example

- Identity

Lambda Calculus

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

Untyped Lambda Calculus

$$\begin{aligned} t ::= \; & x && \text{Variable} \\ & \lambda x.t && \text{Abstraction} \\ & t \; t && \text{Application} \end{aligned}$$

Example

- Identity

Lambda Calculus

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

Javascript

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

Evaluation / Reduction

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

Evaluation / Reduction

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

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

Evaluation / Reduction

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

$$\textit{Application}$$

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

Evaluation / Reduction

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

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

$$\rightarrow(\lambda \mathrm{y}.a\ \mathrm{y})\ \mathrm{b}$$

$$\rightarrow a\ b$$

Evaluation / Reduction

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

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

$$\rightarrow(\lambda \mathrm{y}.a\ \mathrm{y})\ \mathrm{b}$$

$$\rightarrow a\ \ b$$

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

Notational Conventions

- We use parentheses to clearify 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

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

Scope

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

λy y is *bound*, x and z are *free*

λx x and y are *bound*, z is free

λx, λy *binder*

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

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

Operational Semantics

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

Full Beta-Reduction

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

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

Full Beta-Reduction

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

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

- 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

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

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

Normal Order Reduction

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

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

Normal Order Reduction

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

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

- 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

- 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

- Call-by-Value
    - eager, strict

Higher Order Functions

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

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

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

Reductions and Conversions

- Alpha conversion

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

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 -> Variable Abstraction -> Function Declaration
Application -> 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 $\rightarrow y$

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

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

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

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

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

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

Pair Example (continued)

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

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

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

$$
\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
1 &\equiv \lambda f..f\ x
\end{aligned}
$$

$$
\mathit{successor} \equiv \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)
$$

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

Numerals Example - 0 + 0

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

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 . . . :)