From λx.x to Facebook - practical Lambda Calculus and its origins

Functional Miners Meetup

May 21, 2019

Contents

Contents

Introduction

Let us talk some history

Introduction
history()

Alonzo Church (1903-1995)

American mathematician and logician who made a major contributions to mathematical logic and theoretical computer science, creator of Lambda Calculus. Professor at Princeton and California(UCLA). Teacher of Alan Turing, Stephen Cole Kleene and Rosser, J. Barkley.

Biggest accomplishments:

- Church-Rosser Theorem

Introduction

history()

Alonzo Church (1903-1995)

American mathematician and logician who made a major contributions to mathematical logic and theoretical computer science, creator of Lambda Calculus. Professor at Princeton and California(UCLA). Teacher of Alan Turing, Stephen Cole Kleene and Rosser, J. Barkley.

Biggest accomplishments:

- Church-Rosser Theorem
- Church-Turing Theorem

Introduction

history()

Alonzo Church (1903-1995)

American mathematician and logician who made a major contributions to mathematical logic and theoretical computer science, creator of Lambda Calculus. Professor at Princeton and California(UCLA). Teacher of Alan Turing, Stephen Cole Kleene and Rosser, J. Barkley.

Biggest accomplishments:

- Church-Rosser Theorem
- Church-Turing Theorem
- Church Thesis

Introduction
history()

Alonzo Church (1903-1995)

American mathematician and logician who made a major contributions to mathematical logic and theoretical computer science, creator of Lambda Calculus. Professor at Princeton and California(UCLA). Teacher of Alan Turing, Stephen Cole Kleene and Rosser, J. Barkley.

Biggest accomplishments:

- Church-Rosser Theorem
- Church-Turing Theorem
- Church Thesis
- Formal system of computation - Lambda Calculus

Introduction

history()

- Work on Foundations Of Mathematics

Introduction
history()

- Work on Foundations Of Mathematics
- (1936) formal system: computability based on notion of function and logic formalization

Introduction

history()

- Work on Foundations Of Mathematics
- (1936) formal system: computability based on notion of function and logic formalization
- Kleene/Rosser proved system inconsistent

Introduction

history()

- Work on Foundations Of Mathematics
- (1936) formal system: computability based on notion of function and logic formalization
- Kleene/Rosser proved system inconsistent
- Church's formal system stripped of logic formalization - Lambda Calculus

Introduction

history()

- Work on Foundations Of Mathematics
- (1936) formal system: computability based on notion of function and logic formalization
- Kleene/Rosser proved system inconsistent
- Church's formal system stripped of logic formalization - Lambda Calculus
- Church and Rosser prove LC to be confluent under β-reduction

Introduction

history()

- Work on Foundations Of Mathematics
- (1936) formal system: computability based on notion of function and logic formalization
- Kleene/Rosser proved system inconsistent
- Church's formal system stripped of logic formalization - Lambda Calculus
- Church and Rosser prove LC to be confluent under β-reduction

Introduction

history()

- Scarce capabilities of the system

Introduction
history()

- Scarce capabilities of the system
- Kleene defines predecessor function (during a dentist visit he says)

Introduction
history()

- Scarce capabilities of the system
- Kleene defines predecessor function (during a dentist visit he says)
- Church Thesis about universal description of computation again on the spotlight

Introduction

history()

- Scarce capabilities of the system
- Kleene defines predecessor function (during a dentist visit he says)
- Church Thesis about universal description of computation again on the spotlight
- Turing's model of computation

Introduction
history()

Types of Lambda Calculus:

- 1934 - Simply (implicitly) Typed Lambda Calculus (Haskell Curry)
- 1940 - Simply (explicitly) Typed Lambda Calculus (Church)
- 1972 - System F, Systemω (Girard)

Contents

What is Lambda Calculus

What it really is?

What is Lambda Calculus
definition()

Definition

A formal system in mathematical logic for expressing computation based on function abstraction

What is Lambda Calculus

definition()

Definition

A formal system in mathematical logic for expressing computation based on function abstraction

Let X be the infinite, countable set of variables then lambda expression is defined as:

- if $a \in X$ then a is a lambda expression

What is Lambda Calculus

definition()

Definition

A formal system in mathematical logic for expressing computation based on function abstraction

Let X be the infinite, countable set of variables then lambda expression is defined as:

- if $a \in X$ then a is a lambda expression
- if M is a lambda expression and $x \in X$, then $\lambda x.M$ is a lambda expression

What is Lambda Calculus

definition()

Definition

A formal system in mathematical logic for expressing computation based on function abstraction

Let X be the infinite, countable set of variables then lambda expression is defined as:

- if $a \in X$ then a is a lambda expression
- if M is a lambda expression and $x \in X$, then $\lambda x.M$ is a lambda expression
- if N and M are lambda expressions then $(N\ M)$ is a lambda expression

What is Lambda Calculus

grammar()

BNF:

-
  ```
  <exp> ::= <var>
          | \<var> . <exp>
          | ( <exp> <exp> )
  ```

What is Lambda Calculus

grammar()

BNF:

-
  ```
  <exp> ::= <var>
          | \<var> . <exp>
          | ( <exp> <exp> )
  ```

Haskell:

-
  ```haskell
  data Expr = Name String
            | Lam String Expr
            | App Expr Expr
  ```

What is Lambda Calculus

grammar()

```
<exp> ::= <var>
        | \<var> . <exp>
        | ( <exp> <exp> )
```

What is Lambda Calculus

grammar()

```
<exp> ::= <var>
        | \<var> . <exp>
        | ( <exp> <exp> )
```

- a variable, mainly letters e.g. a, b, x, y

What is Lambda Calculus

grammar()

```
<exp> ::= <var>
        | \<var> . <exp>
        | ( <exp> <exp> )
```

- a variable, mainly letters e.g. a, b, x, y
- a function abstraction called lambda abstraction which corresponds directly to function definition, e.g. λ x.y

What is Lambda Calculus

grammar()

```
<exp> ::= <var>
        | \<var> . <exp>
        | ( <exp> <exp> )
```

- a variable, mainly letters e.g. a, b, x, y
- a function abstraction called lambda abstraction which corresponds directly to function definition, e.g. λ x.y
- An application, for us programmers a function invocation e.g. (λx. y x)

What is Lambda Calculus

$$\underbrace{\lambda x}_{\text{head}} . \underbrace{xyz}_{\text{body}}$$

What is Lambda Calculus

reductionorders()

There are two ways of evaluating function applications. All occurrences of bound variable are then replaced by either:

What is Lambda Calculus

reductionorders()

There are two ways of evaluating function applications. All occurrences of bound variable are then replaced by either:

- the value of the argument expression - **Applicative Order**

What is Lambda Calculus

reductionorders()

There are two ways of evaluating function applications. All occurrences of bound variable are then replaced by either:

- the value of the argument expression - **Applicative Order**
- the unevaluated argument expression - **Normal Order**

What is Lambda Calculus

reductionorders()

There are two ways of evaluating function applications. All occurrences of bound variable are then replaced by either:

- the value of the argument expression - **Applicative Order**
- the unevaluated argument expression - **Normal Order**

$$double\ x = plus\ x\ x$$

$$average\ x\ y = divide\ (plus\ x\ y)\ 2$$

Lets evaluate:

$$double\ (average\ 2\ 4)$$

What is Lambda Calculus
reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4)* $=>$

What is Lambda Calculus
reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4)* $=>$
- *plus (average 2 4) (average 2 4)* $=>$

What is Lambda Calculus

reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4)* $=>$
- *plus (average 2 4) (average 2 4)* $=>$
- *plus (divide (plus 2 4) 2) (average 2 4)* $=>$

What is Lambda Calculus

reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4)* $=>$
- *plus (average 2 4) (average 2 4)* $=>$
- *plus (divide (plus 2 4) 2) (average 2 4)* $=>$
- *plus (divide 6 2) (average 2 4)* $=>$

What is Lambda Calculus

reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4)* $=>$
- *plus (average 2 4) (average 2 4)* $=>$
- *plus (divide (plus 2 4) 2) (average 2 4)* $=>$
- *plus (divide 6 2) (average 2 4)* $=>$
- *plus 3 (average 2 4)* $=>$

What is Lambda Calculus
reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4)* $=>$
- *plus (average 2 4) (average 2 4)* $=>$
- *plus (divide (plus 2 4) 2) (average 2 4)* $=>$
- *plus (divide 6 2) (average 2 4)* $=>$
- *plus 3 (average 2 4)* $=>$
- *plus 3 (divide (plus 2 4) 2)* $=>$

What is Lambda Calculus
reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4) =>*
- *plus (average 2 4) (average 2 4) =>*
- *plus (divide (plus 2 4) 2) (average 2 4) =>*
- *plus (divide 6 2) (average 2 4) =>*
- *plus 3 (average 2 4) =>*
- *plus 3 (divide (plus 2 4) 2) =>*
- *plus 3 (divide 6 2) =>*

What is Lambda Calculus

reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4) =>*
- *plus (average 2 4) (average 2 4) =>*
- *plus (divide (plus 2 4) 2) (average 2 4) =>*
- *plus (divide 6 2) (average 2 4) =>*
- *plus 3 (average 2 4) =>*
- *plus 3 (divide (plus 2 4) 2) =>*
- *plus 3 (divide 6 2) =>*
- *plus 3 3 =>*

What is Lambda Calculus
reductionorders()

Normal order of evaluation - rewrite the leftmost outermost occurrence of a function application

- *double (average 2 4) =>*
- *plus (average 2 4) (average 2 4) =>*
- *plus (divide (plus 2 4) 2) (average 2 4) =>*
- *plus (divide 6 2) (average 2 4) =>*
- *plus 3 (average 2 4) =>*
- *plus 3 (divide (plus 2 4) 2) =>*
- *plus 3 (divide 6 2) =>*
- *plus 3 3 =>*
- *6*

What is Lambda Calculus

reductionorders()

Applicative Order of reduction - rewrite the leftmost innermost occurrence of a function application first

- *double (average 2 4)* $=>$

What is Lambda Calculus

reductionorders()

Applicative Order of reduction - rewrite the leftmost innermost occurrence of a function application first

- *double (average 2 4) =>*
- *double (divide (plus 2 4) 2) =>*

What is Lambda Calculus
reductionorders()

Applicative Order of reduction - rewrite the leftmost innermost occurrence of a function application first

- *double (average 2 4) =>*
- *double (divide (plus 2 4) 2) =>*
- *double (divide 6 2) =>*

What is Lambda Calculus

reductionorders()

Applicative Order of reduction - rewrite the leftmost innermost occurrence of a function application first

- *double (average 2 4)* $=>$
- *double (divide (plus 2 4) 2)* $=>$
- *double (divide 6 2)* $=>$
- *double 3* $=>$

What is Lambda Calculus

reductionorders()

Applicative Order of reduction - rewrite the leftmost innermost occurrence of a function application first

- *double (average 2 4)* $=>$
- *double (divide (plus 2 4) 2)* $=>$
- *double (divide 6 2)* $=>$
- *double 3* $=>$
- *plus 3 3* $=>$

What is Lambda Calculus

reductionorders()

Applicative Order of reduction - rewrite the leftmost innermost occurrence of a function application first

- *double (average 2 4)* $=>$
- *double (divide (plus 2 4) 2)* $=>$
- *double (divide 6 2)* $=>$
- *double 3* $=>$
- *plus 3 3* $=>$
- *6*

What is Lambda Calculus

evaluationstrategies()

- Strict
    - Call by value (Swift, C)
    - Call by address/reference (C++)
    - Call by sharing (Java)
- Non-strict
    - Call by name (Haskell)
    - Call by need (memoization)
    - Lazy Evaluation (Miranda)

What is Lambda Calculus

βreduction()

Definition

Beta reduction is a reduction in form of susbtitution of lambda expressions among terms called beta redexes that may lead to beta normal form of the expression

- Beta redex - is a term of form (λx.A)M
- Beta Normal Form - term is in normal form if no beta reduction is possible

What is Lambda Calculus

practicalbetareduction()

Let M $= \underline{\text{x}}\text{y}$

What is Lambda Calculus
practicalbetareduction()

Let $M = \underline{x}y$

$(\lambda x.M\ E)$

What is Lambda Calculus

practicalbetareduction()

$$\text{Let M} = \underline{\text{x}}\text{y}$$

$$(\lambda \text{x.M E}) \xrightarrow[M[x := E]]{\cancel{(\lambda \text{x.}}\text{M } \cancel{\text{E})}}$$

What is Lambda Calculus

practicalbetareduction()

$$\text{Let M} = \underline{\text{x}}\text{y}$$

$$(\lambda \text{x.M E}) \xrightarrow[M[x := E]]{\cancel{(\lambda \text{x.}}\text{M }\cancel{\text{E})}} \underline{\text{E}}\text{y}$$

What is Lambda Calculus

normalform()

λy.y is a normal form of:

What is Lambda Calculus
normalform()

λy.y is a normal form of:

- (λx.λy.y (λz.z z λz.zz))

What is Lambda Calculus

normalform()

λy.y is a normal form of:

- (λx.λy.y (λz.z z λz.zz))
- λx.x
- ...

What is Lambda Calculus

normalform()

λy.y is a normal form of:

- (λx.λy.y (λz.z z λz.zz))
- λx.x
- ...

Is there only one unique normal form?

What is Lambda Calculus

normalform()

λy.y is a normal form of:

- (λx.λy.y (λz.z z λz.zz))
- λx.x
- ...

Is there only one unique normal form?

Yes!

What is Lambda Calculus

normalform()

λy.y is a normal form of:

- (λx.λy.y (λz.z z λz.zz))
- λx.x
- ...

Is there only one unique normal form?
Yes!

Can we always obtain a normal form of an expression?

What is Lambda Calculus
normalform()

λy.y is a normal form of:

- (λx.λy.y (λz.z z λz.zz))
- λx.x
- ...

Is there only one unique normal form?
Yes!

Can we always obtain a normal form of an expression?
No! Consider:

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

What is Lambda Calculus

normalform()

- (λz. (z z) λx. (x x)) $=>$

What is Lambda Calculus

normalform()

- $(\lambda$z. (z z) λx. (x x)) $=>$
- $(\lambda$x. (x x) λx. (x x)) $=>$

What is Lambda Calculus

normalform()

- (λz. (z z) λx. (x x)) $=>$
- (λx. (x x) λx. (x x)) $=>$
- (λx. (x x) λx. (x x)) $=>$

What is Lambda Calculus

normalform()

- (λz. (z z) λx. (x x)) $=>$
- (λx. (x x) λx. (x x)) $=>$
- (λx. (x x) λx. (x x)) $=>$
- ...

What is Lambda Calculus

churchrossertheorem()

Church Rosser Theorem I Corollary

Value of normal form does not depend on order, if reduction terminates it provides a unique normal form.

Church Rosser Theorem II Corollary

If an expression has a normal form, it can be reached by normal order evaluation.

What is Lambda Calculus

alphaconversion()

$$((\lambda \text{func}.\lambda \text{arg}.(\text{func arg})\ \text{arg})\ z)$$

- $=>$ $(\lambda \text{arg}.(\text{arg arg})\ z)$
- $=>$ (z z)(!)

using α conversion we rename arg to arg1:

- $==$ $((\lambda \text{func}.\lambda \text{arg1}.(\text{func arg1})\ \text{arg})\ z)$
- $=>$ $(\lambda \text{arg1}.(\text{arg arg1})\ z)$
- $=>$ (arg z)

What is Lambda Calculus
debruijn()

Nicolaas Govert de Bruijn

$$\lambda\text{x}.\lambda\text{y}.\lambda\text{z}.\ \text{x z (y z)} \xrightarrow[\text{indexing}]{\text{de bruijn}} \lambda\ \lambda\ \lambda\ 3\ 1\ (2\ 1)$$

Contents

Practical Lambda Calculus

identity()

Identity: $\lambda\mathbf{x.x}$

Practical Lambda Calculus

identity()

Identity: $\lambda\mathbf{x.x}$

$$(\lambda \text{x.x } \underline{5}) => \underline{5}$$

Practical Lambda Calculus

identity()

Identity: λ**x.x**

$$(\lambda x.x\ \underline{5}) => \underline{5}$$

$$(\lambda x.x\ \underline{\lambda f.\lambda x.(f\ x)}) => \underline{\lambda f.\lambda x.(f\ x)}$$

Practical Lambda Calculus
identity()

Identity: $\lambda\mathbf{x.x}$

$$(\lambda x.x\ \underline{5}) => \underline{5}$$

$$(\lambda x.x\ \underline{\lambda f.\lambda x.(f\ x)}) => \underline{\lambda f.\lambda x.(f\ x)}$$

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

What can we encode?

but can we represent numbers, boolean expressions, types?

Practical Lambda Calculus

pairs()

Lets start with a pair:

Practical Lambda Calculus

pairs()

Lets start with a pair:

Definition

pair $= \lambda$x.

Practical Lambda Calculus
pairs()

Lets start with a pair:

Definition

pair $= \lambda x.\lambda y.$

Practical Lambda Calculus

pairs()

Lets start with a pair:

Definition

pair = λx.λy.λf. ((f x) y)

Practical Lambda Calculus

pairs()

Lets start with a pair:

Definition

pair = λx.λy.λf. ((f x) y)

- pair 1 2

Practical Lambda Calculus

pairs()

Lets start with a pair:

Definition

pair $= \lambda$x.λy.λf. ((f x) y)

- pair 1 2
- $==$ ((λx.λy.λf. ((f x) y) 1) 2)

Practical Lambda Calculus

pairs()

Lets start with a pair:

Definition

pair $= \lambda$x.λy.λf. ((f x) y)

- pair 1 2
- $==$ ((λx.λy.λf. ((f x) y) 1) 2)
- $=>$ (λy.λf. ((f 1) y) 2)

Practical Lambda Calculus

pairs()

Lets start with a pair:

Definition

pair $= \lambda$x.λy.λf. ((f x) y)

- pair 1 2
- $==$ ((λx.λy.λf. ((f x) y) 1) 2)
- $=>$ (λy.λf. ((f 1) y) 2)
- $=>$ λf. ((f 1) 2)

Practical Lambda Calculus
pairs()

What if we want to get the first value?

Practical Lambda Calculus

pairs()

What if we want to get the first value?

Definition

first $= \lambda$x.

Practical Lambda Calculus

pairs()

What if we want to get the first value?

Definition

first $= \lambda$x.λy.

Practical Lambda Calculus
pairs()

What if we want to get the first value?

Definition

first $= \lambda$x.λy.x

Practical Lambda Calculus

pairs()

What if we want to get the first value?

Definition

first = λx.λy.x

pair $\underline{1}$ 2 first

- => ... => (λf. ((f 1) 2) first)

Practical Lambda Calculus

pairs()

What if we want to get the first value?

Definition

first = λx.λy.x

pair $\underline{1}$ 2 first

- => ... => (λf. ((f 1) 2) first)
- => ((first 1) 2)

Practical Lambda Calculus

pairs()

What if we want to get the first value?

Definition

first $= \lambda x.\lambda y.x$

pair $\underline{1}$ 2 first

- => ... => (λf. ((f 1) 2) first)
- => ((first 1) 2)
- == (($\lambda x.\lambda y.x$ 1) 2)

Practical Lambda Calculus

pairs()

What if we want to get the first value?

Definition

first $= \lambda$x.λy.x

pair $\underline{1}$ 2 first

- => ... => (λf. ((f 1) 2) first)
- => ((first 1) 2)
- == ((λx.λy.x 1) 2)
- => (λy.1) 2)

Practical Lambda Calculus
pairs()

What if we want to get the first value?

Definition

first $= \lambda$x.λy.x

pair $\underline{1}$ 2 first

- => ... => (λf. ((f 1) 2) first)
- => ((first 1) 2)
- == ((λx.λy.x 1) 2)
- => (λy.1) 2)
- => $\underline{1}$

Practical Lambda Calculus

pairs()

You have just encoded pairs!

Pair

λx.λy.λf. ((f x) y)

Select First

λx.λy.x

Select Second

λx.λy.y

Practical Lambda Calculus

pairs()

Haskell implementation

```
pair :: a -> b -> (forall c. (a -> b -> c) -> c)
pair x y = \f -> f x y

first :: a -> b -> a
first x y = x

second :: a -> b -> b
second x y = y
```

Practical Lambda Calculus

conditionals()

$$< condition >? < expression >:< expression >$$

Practical Lambda Calculus

conditionals()

$$< condition >? < expression >:< expression >$$

e.g. $max = x > y?x : y$

Practical Lambda Calculus

conditionals()

$$< condition >? < expression >:< expression >$$

$$\text{e.g. } max = x > y?x : y$$

Lets model a condition abstraction using our pair definition:

Definition

if $= \lambda$e1.λe2.λc.((c e1) e2)

Practical Lambda Calculus

conditionals()

if expression1 expression2

- $==$ ((λe1.λe2.λc.((c e1) e2) expression1) expression2)

Practical Lambda Calculus

conditionals()

if expression1 expression2

- == ((λe1.λe2.λc.((c e1) e2) expression1) expression2)
- => (λe2.λc.((c expression1) e2) expression2)

Practical Lambda Calculus
conditionals()

if expression1 expression2

- $==$ ((λe1.λe2.λc.((c e1) e2) expression1) expression2)
- $=>$ (λe2.λc.((c expression1) e2) expression2)
- $=>$ λc.((c expression1) expression2)

Practical Lambda Calculus

conditionals()

if expression1 expression2

- $==$ ((λe1.λe2.λc.((c e1) e2) expression1) expression2)
- $=>$ (λe2.λc.((c expression1) e2) expression2)
- $=>$ λc.((c expression1) expression2)

Practical Lambda Calculus

conditionals()

if expression1 expression2

- $==$ ((λe1.λe2.λc.((c e1) e2) expression1) expression2)
- $=>$ (λe2.λc.((c expression1) e2) expression2)
- $=>$ λc.((c expression1) expression2)

(λc.((c expression1) expression2) first)

- $==$ (λc.((c expression1) expression2) λ.x..x)
- $=>$ ((λ.x.λ y.x expression1) expression2)
- $=>$ (λy.expression1 expression2)
- $=>$ $\underline{\text{expression1}}$

Practical Lambda Calculus

conditionals()

Definition

true = λ.x.λy.x

Definition

false = λ.x.λy.y

if e1 e2 first == if e1 e2 true

=> ... => e1

Practical Lambda Calculus

not()

X	NOT X
TRUE	FALSE
FALSE	TRUE

Lets look at C-ish example:

x ? false : true

and using if:

(((if false) true) x)

Practical Lambda Calculus

not()

(((if false) true) x)

- == (((if false) true) x)

Practical Lambda Calculus

not()

(((if false) true) x)

- == (((if false) true) x)
- == (((λe1.λe2.λc.((c e1) e2) false) true) x)

Practical Lambda Calculus

not()

(((if false) true) x)

- == (((if false) true) x)
- == (((λe1.λe2.λc.((c e1) e2) false) true) x)
- => ((λe2.λc.((c false) e2) true) x)

Practical Lambda Calculus

not()

(((if false) true) x)

- == (((if false) true) x)
- == (((λe1.λe2.λc.((c e1) e2) false) true) x)
- => ((λe2.λc.((c false) e2) true) x)
- => ((λc.((c false) true) x)

Practical Lambda Calculus
not()

(((if false) true) x)

- == (((if false) true) x)
- == (((λe1.λe2.λc.((c e1) e2) false) true) x)
- => ((λe2.λc.((c false) e2) true) x)
- => ((λc.((c false) true) x)
- => ((x false) true)

Practical Lambda Calculus

not()

(((if false) true) x)

- == (((if false) true) x)
- == (((λe1.λe2.λc.((c e1) e2) false) true) x)
- => ((λe2.λc.((c false) e2) true) x)
- => ((λc.((c false) true) x)
- => ((x false) true)

Definition

not = λx.((x false) true)

Practical Lambda Calculus
not()

Lets check if our negation is correctly encoded

Practical Lambda Calculus

not()

Lets check if our negation is correctly encoded

$$\mathbf{not\ true}$$

Practical Lambda Calculus
not()

Lets check if our negation is correctly encoded

not true

- $==$ (λx.((x false) true) true)

Practical Lambda Calculus

not()

Lets check if our negation is correctly encoded

not true

- $==$ (λx.((x false) true) true)
- $=>$ ((true false) true)

Practical Lambda Calculus

not()

Lets check if our negation is correctly encoded

not true

- $==$ (λx.((x false) true) true)
- $=>$ ((true false) true)
- $==$ ((λx.λy.x false) true)

Practical Lambda Calculus

not()

Lets check if our negation is correctly encoded

not true

- $==$ (λx.((x false) true) true)
- $=>$ ((true false) true)
- $==$ ((λx.λy.x false) true)
- $=>$ (λy.false true)

Practical Lambda Calculus

not()

Lets check if our negation is correctly encoded

not true

- $==$ (λx.((x false) true) true)
- $=>$ ((true false) true)
- $==$ ((λx.λy.x false) true)
- $=>$ (λy.false true)
- $=>$ <u>false</u>

Practical Lambda Calculus
not()

not false

Practical Lambda Calculus

not()

not false

- $==$ (λx.((x false) true) false)

Practical Lambda Calculus
not()

not false

- $==$ (λx.((x false) true) false)
- $=>$ ((false false) true)

Practical Lambda Calculus

not()

not false

- $==$ (λx.((x false) true) false)
- $=>$ ((false false) true)
- $==$ ((λx.λy.y false) true)

Practical Lambda Calculus

not()

not false

- $==$ (λx.((x false) true) false)
- $=>$ ((false false) true)
- $==$ ((λx.λy.y false) true)
- $=>$ (λy.y true)

Practical Lambda Calculus
not()

not false

- $==$ (λx.((x false) true) false)
- $=>$ ((false false) true)
- $==$ ((λx.λy.y false) true)
- $=>$ (λy.y true)
- $=>$ <u>true</u>

Practical Lambda Calculus

and()

X	Y	X AND Y
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE

Lets look at C-ish example:

```
x ? y : false
```

and using if:

```
(((if y) false) x)
```

Practical Lambda Calculus
and()

(((if y) false) x)

- == (((if y) false) x)

Practical Lambda Calculus

and()

(((if y) false) x)

- == (((if y) false) x)
- == (((λe1.λe2.λc.((c e1) e2) y) false) x)

Practical Lambda Calculus
and()

(((if y) false) x)

- == (((if y) false) x)
- == (((λe1.λe2.λc.((c e1) e2) y) false) x)
- => ((λe2.λc.((c y) e2) false) x)

Practical Lambda Calculus

and()

(((if y) false) x)

- == (((if y) false) x)
- == (((λe1.λe2.λc.((c e1) e2) y) false) x)
- => ((λe2.λc.((c y) e2) false) x)
- => (λc.((c y) false) x)

Practical Lambda Calculus

and()

(((if y) false) x)

- == (((if y) false) x)
- == (((λe1.λe2.λc.((c e1) e2) y) false) x)
- => ((λe2.λc.((c y) e2) false) x)
- => (λc.((c y) false) x)
- => ((x y) false)

Practical Lambda Calculus
and()

(((if y) false) x)

- == (((if y) false) x)
- == (((λe1.λe2.λc.((c e1) e2) y) false) x)
- => ((λe2.λc.((c y) e2) false) x)
- => (λc.((c y) false) x)
- => ((x y) false)

Definition

and = λx.λy.((x y) false)

Practical Lambda Calculus
and()

Lets check if our and is correctly encoded

Practical Lambda Calculus
and()

Lets check if our and is correctly encoded

and <u>true true</u>

Practical Lambda Calculus

and()

Lets check if our and is correctly encoded

and <u>true true</u>

- $==$ ((λx.λy.((x y) false) true) true)

Practical Lambda Calculus
and()

Lets check if our and is correctly encoded

and <u>true true</u>

- $==$ ((λx.λy.((x y) false) true) true)
- $=>$ (λy.((true y) false) true)

Practical Lambda Calculus
and()

Lets check if our and is correctly encoded

and <u>true true</u>

- $==$ ((λx.λy.((x y) false) true) true)
- $=>$ (λy.((true y) false) true)
- $=>$ ((true true) false)

Practical Lambda Calculus

and()

Lets check if our and is correctly encoded

and <u>true true</u>

- $==$ ((λx.λy.((x y) false) true) true)
- $=>$ (λy.((true y) false) true)
- $=>$ ((true true) false)
- $==$ ((λx.λy.x true) false)

Practical Lambda Calculus
and()

Lets check if our and is correctly encoded

and <u>true true</u>

- $==$ ((λx.λy.((x y) false) true) true)
- $=>$ (λy.((true y) false) true)
- $=>$ ((true true) false)
- $==$ ((λx.λy.x true) false)
- $=>$ (λy.true false)

Practical Lambda Calculus

and()

Lets check if our and is correctly encoded

and <u>true true</u>

- $==$ ((λx.λy.((x y) false) true) true)
- $=>$ (λy.((true y) false) true)
- $=>$ ((true true) false)
- $==$ ((λx.λy.x true) false)
- $=>$ (λy.true false)
- $=>$ <u>true</u>

Practical Lambda Calculus

and()

and <u>true false</u>

Practical Lambda Calculus

and()

and <u>true false</u>

- == ((λx.λy.((x y) false) true) false)

Practical Lambda Calculus

and()

and <u>true false</u>

- $==$ ((λx.λy.((x y) false) true) false)
- $=>$ (λy.((true y) false) true)

Practical Lambda Calculus

and()

and <u>true false</u>

- $==$ ((λx.λy.((x y) false) true) false)
- $=>$ (λy.((true y) false) true)
- $=>$ ((true false) false)

Practical Lambda Calculus

and()

and <u>true false</u>

- == ((λx.λy.((x y) false) true) false)
- => (λy.((true y) false) true)
- => ((true false) false)
- == ((λx.λy.x false) false)

Practical Lambda Calculus

and()

and <u>true false</u>

- == ((λx.λy.((x y) false) true) false)
- => (λy.((true y) false) true)
- => ((true false) false)
- == ((λx.λy.x false) false)
- => (λy.false) false)

Practical Lambda Calculus

and()

and true false

- == ((λx.λy.((x y) false) true) false)
- => (λy.((true y) false) true)
- => ((true false) false)
- == ((λx.λy.x false) false)
- => (λy.false) false)
- => false

Practical Lambda Calculus

or()

X	Y	X OR Y
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE

Lets look at C-ish example:

x ? true : y

and using if:

(((if true) y) x)

Practical Lambda Calculus

or()

(((if true) y) x)

- == (((if true) y) x)

Practical Lambda Calculus

or()

(((if true) y) x)

- == (((if true) y) x)
- => ... => ((x true) y)

Definition

or = λx.λy.((x true) y)

What can we encode?

numbers()

Definition

zero = identity

Definition

succ = λn.λs.((s false) n)

What can we encode?

numbers()

$$\underline{\text{one}} = \text{succ zero}$$

- $== (\lambda \text{n}.\lambda \text{s}.((\text{s false})\ \text{n})\ \text{zero})$
- $=> \underline{\lambda \text{s}.((\text{s false})\ \text{zero})}$

What can we encode?
numbers()

$$\underline{\text{one}} = \text{succ zero}$$

- $== (\lambda n.\lambda s.((s\ \text{false})\ n)\ \text{zero})$
- $=> \underline{\lambda s.((s\ \text{false})\ \text{zero})}$

$$\underline{\text{two}} = \text{succ one}$$

- $== (\lambda n.\lambda s.((s\ \text{false})\ n)\ \lambda s.((s\ \text{false})\ \text{zero}))$
- $=> \underline{\lambda s.((s\ \text{false})\ \lambda s.((s\ \text{false})\ \text{zero}))}$

What can we encode?

numbers()

$$\underline{\text{one}} = \text{succ zero}$$

- $== (\lambda\text{n}.\lambda\text{s}.((\text{s false})\ \text{n})\ \text{zero})$
- $=> \underline{\lambda\text{s}.((\text{s false})\ \text{zero})}$

$$\underline{\text{two}} = \text{succ one}$$

- $== (\lambda\text{n}.\lambda\text{s}.((\text{s false})\ \text{n})\ \lambda\text{s}.((\text{s false})\ \text{zero}))$
- $=> \underline{\lambda\text{s}.((\text{s false})\ \lambda\text{s}.((\text{s false})\ \text{zero}))}$

$$\underline{\text{three}} = \text{succ two}$$

- $== (\lambda\text{n}.\lambda\text{s}.((\text{s false})\ \text{n})\ \lambda\text{s}.((\text{s false})\ \lambda\text{s}.((\text{s false})\ \text{zero})))$
- $=> \underline{\lambda\text{s}.((\text{s false})\ \lambda\text{s}.((\text{s false})\ \lambda\text{s}.((\text{s false})\ \text{zero})))}$

What can we encode?
numbers()

By having a number encoded as a function with an argument that can be used as a selector we can try to extract values from them by using our pair first and second functions

What can we encode?

numbers()

By having a number encoded as a function with an argument that can be used as a selector we can try to extract values from them by using our pair first and second functions

$$(\lambda \text{s.(s false) } \textit{number}) \text{ first)}$$

What can we encode?

numbers()

By having a number encoded as a function with an argument that can be used as a selector we can try to extract values from them by using our pair first and second functions

$$(\lambda \text{s.(s false) } \textit{number}) \text{ first})$$

- $== (\lambda$s.(s false) *number*) λx.λy.x)
- $=>$ (λx.λy.x false) *number*)
- $=>$ (λy.false *number*)
- $=>$ false

What can we encode?

numbers()

We used *first* as a selector before, lets now use *second* and see what will we get from the number:

$$(\lambda \text{s.(s false) } \textit{number}) \text{ second})$$

What can we encode?
numbers()

We used *first* as a selector before, lets now use *second* and see what will we get from the number:

$$(\lambda \text{s.(s false) } \textit{number}\text{) second)}$$

- $== (\lambda$s.(s false) *number*) λx.λy.y)

What can we encode?
numbers()

We used *first* as a selector before, lets now use *second* and see what will we get from the number:

$$(\lambda \text{s.(s false) } \textit{number}\text{) second)}$$

- $== (\lambda \text{s.(s false) } \textit{number}) \ \lambda \text{x.} \lambda \text{y.y)}$
- $=> (\lambda \text{x.} \lambda \text{y.y false) } \textit{number})$

What can we encode?
numbers()

We used *first* as a selector before, lets now use *second* and see what will we get from the number:

$$(\lambda\text{s.(s false) } \textit{number}\text{) second)}$$

- $==$ (λs.(s false) *number*) λx.λy.y)
- $=>$ (λx.λy.y false) *number*)
- $=>$ *number*

What can we encode?

numbers()

Definition

pred $= \lambda$n.(n second)

Now we cought up to where **Kleene** was in 1936.

Sources I

Greg Michaelson.
An Introduction to Functional Programming Through Lambda Calculus.
Dover Publications, 1989.

Pat Helland.
Immutability changes everything.

Felice Cardone Roger Hindley.
History of lambda-calculus and combinatory logic.

Henk Barendregt.
The impact of the lambda calculus in logic and computer science.

Stephanie Weirich Justin Hsu Richard A. Eisenberg.
System fc with explicit kind equality.

Sources II

Russel Church.
Some properties of conversion.

Haskell.org.
Representation of the ghc core language.

From λx.x to Facebook - practical Lambda Calculus and its origins

Functional Miners Meetup

May 21, 2019