

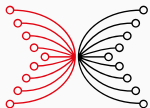
Lambda Calculus

Haskell and Cryptocurrencies

Dr. Andres Löb, Well-Typed LLP

Dr. Lars Brünjes, IOHK

2018-01-26



INPUT | OUTPUT

Goals

- Untyped Lambda Calculus
- α -Equivalence
- β -Equivalence
- Reduction Strategies
- Church Encodings
- General Recursion & Y-Combinator

Untyped Lambda Calculus

High-level properties

- Provides simple semantics and a formal model for *computation*.
- Turing complete.
- Makes two simplifications:
 - only *anonymous* functions
 - only functions of one argument (*curried* functions)
- *Haskell* is based upon and compiles to a (typed!) version of the Lambda Calculus (as first intermediate compiler target, *Core*).

History

- Based upon work by *Frege* from 1893 and *Schönfinkel* from the 1920s.
- Introduced by *Alonzo Church* in the 1930s.
- Shown to be logically inconsistent in 1935 by *Stephen Kleene* and *J. B. Rosser*.
- Fixed by Church in 1936 – *Untyped Lambda Calculus*.
- Relation to programming languages clarified in the 1960s.

Lambda expressions

Lambda expressions (or lambda terms) are composed of

- variables $v_1, v_2, \dots, v_n, \dots$,
- the abstraction symbols λ and $.$,
- parentheses $()$.

The set of lambda expressions Λ is inductively defined as:

- If x is a variable, then $x \in \Lambda$. (variable)
- If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$. (lambda abstraction)
- If $M, N \in \Lambda$, then $(MN) \in \Lambda$. (application)

Notation

To keep the notation of lambda expressions uncluttered, the following conventions are normally applied:

- Outermost parentheses are dropped: MN instead of (MN) .
- Applications are assumed to be left associative: MNP instead of $(MN)P$.
- The body of an abstraction extends as far right as possible: $\lambda x.MN$ means $\lambda x.(MN)$, *not* $(\lambda x.M)N$.
- A sequence of abstractions is contracted: $\lambda x.\lambda y.\lambda z.M$ is abbreviated as $\lambda xyz.M$.

Notation (contd.)

This is just as in Haskell:

- `f x = (f x)`
- `f x y = (f x) y`
- `\ x -> \ y -> \ z -> f = \ x y z -> f`

- As “syntactic sugar”, we can also introduce **let bindings**:
- For a variable x and lambda expressions M and N , we can define **let** $x = N$ **in** M as $(\lambda x.M) N$.
- Later, once we learn about β -reduction, we will see that this “means” substituting N for x in M , which coincides with the intuitive idea we have of how **let** “should” behave.
- This technique is actually used frequently in JavaScript ...

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of **free variables** $FV(M) \subset V$ as follows:
 - For a variable $x \in V$, $FV(x) = \{x\}$.
 - For an abstraction, $FV(\lambda x.M) = FV(M) \setminus \{x\}$.
 - For an application, $FV(MN) = FV(M) \cup FV(N)$.
- Given a lambda expression $M \in \Lambda$, we call a variable $x \in V$ **free** (in M) if $x \in FV(M)$.
- In an abstraction $\lambda x.M$, we call the variable x **bound**.

Subexpressions

We define the notion of **subexpression** of a lambda expression inductively as follows:

- Each term is a subexpression of itself. Sub-expressions other than the term itself are called **proper** subexpressions.
- A variable has no proper subexpressions.
- The proper subexpressions of an application MN are the subexpressions of M and the subexpressions of N .
- The proper subexpression of an abstraction $\lambda x.M$ are the subexpressions of the body M .

Substitution

Let $x \in V$ be a variable, and let $M, N \in \Lambda$ be lambda expressions. We define $M[x := N]$, the substitution of x by N in M recursively as follows:

Substitution

Let $x \in V$ be a variable, and let $M, N \in \Lambda$ be lambda expressions. We define $M[x := N]$, the substitution of x by N in M recursively as follows:

- $x[x := N] = N$.
- For a variable $y \neq x$, $y[x := N] = y$.

Substitution

Let $x \in V$ be a variable, and let $M, N \in \Lambda$ be lambda expressions. We define $M[x := N]$, the substitution of x by N in M recursively as follows:

- $x[x := N] = N$.
- For a variable $y \neq x$, $y[x := N] = y$.
- $(P Q)[x := N] = P[x := N] Q[x := N]$

Substitution

Let $x \in V$ be a variable, and let $M, N \in \Lambda$ be lambda expressions. We define $M[x := N]$, the substitution of x by N in M recursively as follows:

- $x[x := N] = N$.
- For a variable $y \neq x$, $y[x := N] = y$.
- $(P Q)[x := N] = P[x := N] Q[x := N]$
- $(\lambda x.P)[x := N] := \lambda x.P$.
- For a variable $y \neq x$ with $y \notin FV(N)$,
 $(\lambda y.P)[x := N] := \lambda y.(P[x := N])$.
- For an abstraction $\lambda y.P$ with $y \neq x$ and $y \in FV(N)$, pick a variable $z \notin FV(N) \cup FV(P)$, then
 $(\lambda y.P)[x := N] := \lambda z.(P[y := z][x := N])$.

Substitution (contd.)

- It looks as if substitution was not well defined, because the result depends on a *choice* of variable in the last case.
- In the next section, we will define an equivalence relation on lambda expressions, *α -equivalence*, which will make the choice of variable name irrelevant.

Example substitutions

$$\bullet (\lambda x.x)[x := z (\lambda u.u)] = \lambda x.x$$

Example substitutions

- $(\lambda x.x)[x := z (\lambda u.u)] = \lambda x.x$
- $(\lambda y.x)[x := z (\lambda u.u)] = \lambda y.z (\lambda u.u)$

Example substitutions

- $(\lambda x.x)[x := z (\lambda u.u)] = \lambda x.x$
- $(\lambda y.x)[x := z (\lambda u.u)] = \lambda y.z (\lambda u.u)$
-

$$\begin{aligned} & (\lambda z.x)[x := z (\lambda u.u)] \\ &= \lambda z'.(x[z := z'] [x := z (\lambda u.u)]) \\ &= \lambda z'.(x[x := z (\lambda u.u)]) \\ &= \lambda z'.z (\lambda u.u) \end{aligned}$$

α -Equivalence

α -Equivalence

We define the notion $M \sim_{\alpha} N$ of α -equivalence between two lambda expressions M and N inductively as follows:

- For two variables x and y , $x \sim_{\alpha} y$ iff $x = y$.
- $(MN) \sim_{\alpha} (PQ)$ iff $M \sim_{\alpha} P$ and $N \sim_{\alpha} Q$.
- For abstractions $\lambda x.M$ and $\lambda y.N$, choose a variable z with $z \notin FV(M) \cup FV(N)$. Then $\lambda x.M \sim_{\alpha} \lambda y.N$ iff $M[x := z] \sim_{\alpha} N[y := z]$.
- All other expressions are *not* α -equivalent.

Intuitively, α -equivalence means that two expressions are equal except for the naming of bound variables.

α -Equivalence (contd.)

Equality

From now on, we will always consider Λ / \sim_α instead of Λ , i.e. we will consider two α -equivalent lambda expressions as *equal*.

Substitution

As soon as α -equivalence becomes equality, the problem in our definition of substitution goes away, because different “picks” of variable name in the last case clearly lead to α -equivalent results.

α -Equivalence (contd.)

- The necessity to deal with α -equivalence makes implementation of substitution and equality checking quite tricky.
- There are ways (for example *De Bruijn indices*) of handling variable names differently in the definition of lambda expressions that lead to *syntactically identical* terms for α -equivalent expressions.
- However, the presentation we chose seems to be the most “human readable”.

β -Equivalence

β -Equivalence captures “computation”

- As explained above, α -equivalence is more of a technical nuisance, capturing the intuitive idea that the names of bound variables should not matter.
- β -equivalence, on the other hand, lies at the very heart of Lambda Calculus and captures the notion of computation.
- Intuitively, the *act of computation* preserves β -equivalence.
- So what does *computation* mean in the context of Lambda Calculus?

β -Reduction

- Consider a lambda expression of the form $(\lambda x.M) N$, i.e. an application where the first argument is an abstraction.
- By definition, this term β -reduces to $M[x := N]$.
- This act of “plugging in” an expression for the bound variable in an abstraction is what constitutes the idea of computation in Lambda Calculus.

Redex, β -equivalence & normal form

- A **redex** of a lambda expression P is a subexpression of P of the form $(\lambda x.M) N$.
- Let P' denote the lambda expression obtained by replacing a redex $(\lambda x.M) N$ with $M[x := N]$. We say that P **β -reduces** to P' (in one step).
- We say that P **β -reduces** to P' (or that P' is a **β -reduct** of P) if reducing zero or more redexes transforms P into P' .
- Two lambda expressions P and P' are **β -equivalent** ($P \sim_{\beta} P'$) if one can be transformed into the other by a chain of β -reductions (or their inverses).
- A lambda expression without redex is said to be in **normal form**.
- We say a lambda expression P **has normal form** P' if P β -reduces to P' and P' is in normal form.

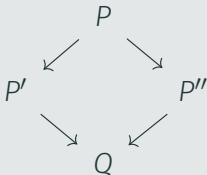
Questions

- Can lambda expression have more than one normal form?
- Does any lambda expression have a normal form?
- If a lambda expression *does* have a normal form, how can I find it/one?

The Church-Rosser theorem

Theorem

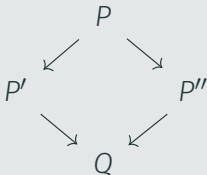
Let P be a lambda expression, and let P' and P'' be two β -reducts of P . Then there is a lambda expression Q such that both P' and P'' β -reduce to Q .



The Church-Rosser theorem

Theorem

Let P be a lambda expression, and let P' and P'' be two β -reducts of P . Then there is a lambda expression Q such that both P' and P'' β -reduce to Q .



At most one

Church-Rosser immediately(!) implies that a lambda expression has *at most one* normal form.

No normal form ...

- Consider the lambda expression $\omega := (\lambda x.xx)(\lambda x.xx)$.

No normal form ...

- Consider the lambda expression $\omega := (\lambda x.xx)(\lambda x.xx)$.
- There is only one redex, ω itself.
- Let's reduce it!

No normal form ...

- Consider the lambda expression $\omega := (\lambda x.xx)(\lambda x.xx)$.
- There is only one redex, ω itself.
- Let's reduce it!
- $(xx)[x := \lambda x.xx] = (\lambda x.xx)(\lambda x.xx) = \omega$.

No normal form ...

- Consider the lambda expression $\omega := (\lambda x.xx)(\lambda x.xx)$.
- There is only one redex, ω itself.
- Let's reduce it!
- $(xx)[x := \lambda x.xx] = (\lambda x.xx)(\lambda x.xx) = \omega$.
- ω only has one redex, and β -reducing that one redex gives ω as a reduct!
- By definition, ω is not(!) in normal form. So ω **has no normal form!**

Open question

- So we know from Church-Rosser that a lambda expression's normal form is unique if it exists.
- We have seen an example of a lambda expression that does *not* have a normal form.
- Open question: How to find the normal form if it exists?
- To be more precise: If there is more than one redex, which one do we reduce first?

Reduction Strategies

Reduction strategies

Reduction Strategy

A **reduction strategy** is an algorithm that, given a lambda expression, decides which redex to reduce (if at least one exists).

Call by value

- Consider the strategy of always reducing the *leftmost innermost* redex first.
- This strategy is called **call by value** or **eager evaluation**.
- Intuitively, it means evaluating the arguments to a function first, then applying the function.
- Example:

$$\begin{aligned} & ((\lambda xy.x) (\lambda x.x)) ((\lambda x.x) y) \\ \sim_{\beta} & (\lambda yx.x) ((\lambda x.x) y) \\ \sim_{\beta} & (\lambda yx.x) y \\ \sim_{\beta} & \lambda x.x \end{aligned}$$

Call by name

- Now consider the strategy of always reducing the *leftmost outermost* redex first.
- This strategy is called **call by name**.
- Intuitively, it means applying a function before evaluating its arguments.
- Example:

$$\begin{aligned} & ((\lambda x y. x) (\lambda x. x)) ((\lambda x. x) y) \\ \sim_{\beta} & (\lambda y x. x) ((\lambda x. x) y) \\ \sim_{\beta} & \lambda x. x \end{aligned}$$

Call by name

- Now consider the strategy of always reducing the *leftmost outermost* redex first.
- This strategy is called **call by name**.
- Intuitively, it means applying a function before evaluating its arguments.
- Example:

$$\begin{aligned} & ((\lambda x y. x) (\lambda x. x)) ((\lambda x. x) y) \\ \sim_{\beta} & (\lambda y x. x) ((\lambda x. x) y) \\ \sim_{\beta} & \lambda x. x \end{aligned}$$

Comparison

With both strategies, we arrive at the same normal form (Church-Rosser!). But call by name needs one step less.

Call by need

- There is an optimization of *call by name*, called **call by need** (or **lazy evaluation**).
- The problem with call by name is that when substituting, we may duplicate function arguments and then maybe will have to evaluate them several times.
- In call by need, instead of substituting arguments by copying them, a pointer to the argument is substituted,
- If an argument needs to be evaluated once, all copies of it profit from the evaluation.

Theorem on call by name

Theorem

Let P be a lambda expression which has a normal form. Then the *call by name/need* strategy will reduce to this normal form.

Counterexample

- Consider the lambda expression $(\lambda xy.x) (\lambda x.x) \omega$.
- It has a normal form, which call by name/need reduces to in two steps:

$$\begin{aligned} & (\lambda xy.x) (\lambda x.x) \omega \\ \sim_{\beta} & (\lambda yx.x) \omega \\ \sim_{\beta} & (\lambda x.x) \end{aligned}$$

- Call by value, on the other hand, enters an infinite loop, because it tries to reduce ω in the second step.
- So we see that call by name finds strictly more normal forms than call by value.
- This example corresponds to the Haskell expression `const id undefined`.

The reduction strategy of Haskell

- The Haskell standard does **not** dictate the use of one particular reduction strategy.
- Instead, it prescribes that Haskell has **non-strict semantics**, which essentially means that reduction must lead to the same results as via call by name.
- In practice, lazy evaluation (call by need) is mostly used by the Haskell compiler, with semantics-preserving optimisations in many places.

Church Encodings

- It is possible to encode an amazing range of datatypes in the Untyped Lambda Calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.

Church encoding of booleans

- Booleans are encoded as functions that take two arguments.
- If the boolean is **True**, the first argument is returned, if it is **False**, the second is returned.
- `true := $\lambda xy.x$`
- `false := $\lambda xy.y$`
- With this, we can define `ifThenElse`:
`ifThenElse := $\lambda bxy.bxy$`
- And logical functions like `not`:
`not := $\lambda b.\text{ifThenElse } b \text{ false true}$` .

Church encoding of natural numbers

- Natural numbers are also encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n -times to x :
- **zero** $:= \lambda fx.x$.
- **succ** $:= \lambda nfx.f(nfx)$.
- We can define addition: **add** $:= \lambda mnfx.mf(nfx)$
- and multiplication: **mul** $:= \lambda mnfx.m(nf)x$
- and test for zero: **isZero** $:= \lambda n.n(\lambda xzy.y)(\lambda xy.x)$
- and predecessor (more complicated!):
pred $:= \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$.
- and many more...

Church encoding of pairs

- Pairs are encoded as functions that take a function of two arguments. The idea is to supply the argument function with the two components of the pair as arguments:
- `pair` := $\lambda xyf.fxy$.
- `fst` := $\lambda p.p(\lambda xy.x)$.
- `snd` := $\lambda p.p(\lambda xy.y)$.

General Recursion & Y-Combinator

Fixpoint operators

- A **fixpoint operator** F is a lambda expression F with the property that for all lambda expressions g , we have $g(Fg) \sim_{\beta} Fg$.
- In Haskell, we have the function `fix` in `Control.Monad.Fix`:

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

- `fix` can be used to implement recursive functions:

```
factorial :: Int -> Int
factorial = fix $ \ f n ->
  if n == 0 then 1 else n * f (n - 1)
```

The Y-combinator

- Consider the lambda expression
 $Y := \lambda f. (\lambda x. f(xx)) (\lambda x. f(xx)).$
- We claim that Y is a fixpoint operator:

$$\begin{aligned} & Yg \\ &= (\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))) g \\ &\sim_{\beta} (\lambda x. g(xx)) (\lambda x. g(xx)) \\ &\sim_{\beta} g((\lambda x. g(xx)) (\lambda x. g(xx))) \\ &\sim_{\beta} g(Yg) \end{aligned}$$

- Using call by name, we can use Y to define recursive functions. For call by value, this won't work, but there are (more complicated) fixpoint operators that work for call by value as well.

```
factorial :=  $Y \lambda f n.$ ifThenElse(isZero  $n$ )  
                                (succ zero)  
                                (mul  $n$  ( $f$  (pred  $n$ )))
```