Calchylus Documentation

Release 1a

Marko Manninen

Mar 05, 2019

Contents

`calchylus` is a computer installable Hy module that is used to evaluate Lambda expressions, and furthermore through this documentation, shine light to the basics of Lambda calculus (also written as λ-calculus).

$$\Large = x.(y.x \space (y \space y)) \space (y.x \space (y \space y))$$

CHAPTER 1

Contents

1.1 Introduction

`calchylus` is a computer installable Hy module that is used to evaluate Lambda expressions, and furthermore through this documentation, shine light to the basics of Lambda calculus (also written as λ-calculus).

Note: Lambda calculus is a formal system in mathematical logic for expressing computation that is based on function abstraction and application using variable binding and substitution.

The target audience is those who:

a) are interested in the theory and the history of the programming languages

b) may have or are interested to gain some experience in Python and/or Lisp

c) who wants to narrow the gap between mathematical notation and programming languages, especially by means of logic

Andrew Bayer writes in his blog post about formal proofs and deduction:

> *Traditional logic, and to some extent also type theory, hides computation behind equality.*

Lambda calculus, on the other hand, reveals how the computation in logic is done by manipulation of the Lambda terms. Manipulation rules are simple and were originally made with a paper and a pen, but now we rather use computers for the task. Lambda calculus also addresses the problem, what can be proved and solved and what cannot be computed in a finite time. Formally these are called the decidability and the halting problem.

Beside evaluating Lambda expressions, `calchylus` module can serve as a starting point for a mini programming language. Via custom macros representing well known Lambda forms, `calchylus` provides all necessary elements for boolean, positive integer, and list data types as well as conditionals, loops, variable setters, imperative do structure, logical connectives, and arithmetic operators. You can build upon that, for example real numbers, even negative complex numbers if that makes any sense. Your imagination is really the only limit.

Finally, when investigating the open source `calchylus` implementation that is hosted on GitHub , one can expect to get a good understanding of the higher order functions and the combinatory logic, not the least of the fixed point

combinator or shortly, combinator:

$$\Large = x.(y.x \space (y \space y)) \space (y.x \space (y \space y))$$

1.2 Quick start

For people willing to get hands quickly on coding:

Install

```
$ pip install calchylus
```

Open Hy

```
$ hy
```

Import

```
(require [calchylus.lambdas [*]])
```

Initialize

```
(with-alpha-conversion-and-macros L ,)
```

Lambda dance

```
(L x y , (x (x (x (x (x y))))) a b) ; output: (a (a (a (a (a b)))))
```

```
(FIBONACCI SEVEN x y) ; output: (x (x (x (x (x (x (x (x (x (x (x (x (x y))))))))))))))
```

1.2.1 Explanation

`calchylus` module works in Windows, Linux, and MacOS operating systems. 3.7 or greater is required. The whole great Python ecosystem can be installed from Anaconda.

Install Hy language interpreter and `calchylus` module by using `pip` Python package management tool:

```
$ pip install calchylus
```

Open Hy, since `calchylus` is mostly written as Hy macros:

```
$ hy
```

Import Lambda calculus macros:

```
(require [calchylus.lambdas [*]])
```

Define Lambda function indicator letter `L` and Lambda argument-body separator character `,` with one of the initializer macros:

```
(with-alpha-conversion-and-macros L ,)
```

By `with-alpha-conversion-and-macros` we want to say that arguments should be internally renamed to prevent argument name collision and that we want to load custom macros representing Lambda forms.

Now, we are ready to evaluate Lambda expressions. Here we apply Church numeral five to the two values, `a` and `b`:

```
(L x y , (x (x (x (x (x y))))) a b)
```

[output]

```
(a (a (a (a (a b)))))
```

Without going deeper into this yet, we can see that all `x` got replaced by `a` and all `y` got replaced by `b`.

Predefined macros are available as shorthands for the most common Lambda forms. For example, calculating the seventh Fibonacci number can be done by using the Church numeral `SEVEN` and the `FIBONACCI` shorthands:

```
(FIBONACCI SEVEN x y)
```

[output]

```
(x (x (x (x (x (x (x (x (x (x (x (x y))))))))))))
```

That is the Church numeral 13, the seventh Fibonacci number.

In `calcylus` these custom macro shorthands representing Lambda forms serves as a mathematical and logical foundation for a prototype programming language that is based on purely untyped Lambda calculus.

1.3 Theory

1.3.1 Historical perspectives

Notation and ideas of differentiable functions comes from the 17th century mathematician Gottfried Leibniz. Early functions dealt with the continuous change of the values. By the 20th century, functions were generalized to map between any two sets, inputs and outputs.

Lambda calculus was invented by Alonzo Church in the 1930s. That happened actually a decade before modern electrically powered computers were created. Lambda calculus can be described as the simplest and the smallest universal Turing complete programming language.

Note: In mathematics, functions are often seen as graphs. In the Lambda calculus, functions are seen as formulas and rules instead. It is a system for manipulating functions as expressions.

LISP (LISt Processor) 1.5 was specified in 1958 by John McCarthy. It is the second oldest language after FORTRAN (FORmula TRANslator), both currently in active use.

In February 1991, Guido van Rossum published the codebase for Python version 0.9.0.

Hy, that is meant to operate as a transparent LISP front end to Python's AST, was introduced at PyCon 2013 by Paul Tagliamonte.

`calchylus` module was programmed with Hy by Marko Manninen in 09/2017.

1.4 Concepts of Lambda calculus

Lambda calculus takes everything to the very few basic computational ideas. First of all, there are only three concepts necessary to express Lambda calculus:

1. variables, that are any single or multiple letter identifiers designating parameters or mathematical values

2. abstractions, that are function definitions which binds arguments to the function body
3. application, that applies the function abstraction to the variables

In the original Lambda calculus you could define one and one only argument per function, but even before Lambda calculus in 1920's Schönfinkel showed that nested unary functions can be used to imitate multiary functions.

Later this mechanism settled down to be called as "currying" and is fully implemented in the `calchylus` module.

Two other syntactic rules must be introduced to be able to write and evaluate Lambda applications:

1. Lambda function indicator, or binding operator that is usually a Greek lambda letter: λ
2. Lambda function argument and body separator, that is usually a dot: `.`

Optionally two more syntax rules can be implemented:

3. Parentheses to group and indicate Lambda applications, abstractions, function bodies and variables. The most convenient way is to use left `(` and right `)` parentheses. Other purpose of using parentheses is to visually make Lambda expressions easier to read and to avoid arbitrarities in Lambda expressions.
4. Space character to distinct function indicator, separator, variables, body, and arguments. This is optional, because in the simplest Lambda calculus implementation single character letters are used to denote variables. But it is easy to see that this would be quite limiting for the practical purposes.

Knowing these we should be fine to write Lambda expressions.

1.5 Lambda expressions in `calchylus` module

All three concepts and four rules are implemented in the `calchylus` module so that for example the very basic Lambda calculus *identity* application λ`x.x y` becomes `(L x , x y)` in `calchylus` notation. Infact, the function indicator and the separator character can be freely defined in `calchylus` by with- initialization macros.

In the most of the examples we will use `L` and `,` because it will be easier to type `L` from the keyboard. Using the comma rather than the dot comes from the Hy programming language environment restrictions, where the dot is a reserved letter for cons in list processing.

Let us strip down the former expression and show how all rules are taking place in it.

In `(L x , x y)`, `L` is the Lambda function indicator and parentheses `()` indicate the whole application that should be evaluated. `x` before the separator `,` is the function argument. `x` after the separator, but **:underline:'before the next space'** is the function body. Finally `y` is the value for the function, thus we have a full application here, rather than just an abstraction. Abstraction would, on the other hand be: `(L x , x)`.

Note:

In mathematics, identity function can be denoted either by $$f(x) = x$$ or by

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

Because these rules are notable in any functional and Lisp like language, there is a great temptation to implement Lambda calculus evaluator as a native anonymous function calls. The problem with this approach is very subtle and will bring practicer to the deep foundations of the programming languages. That is, to decide in which order to evaluate arguments and functions and how to deal with argument name collisions.

Let us first see the easy native implementation of the Lambda calculus to learn what all this means.

1.6 Easy native implementation

This is the simple implementation of the Lambda calculus evaluator. It will utilize the native anonymous function declaration `fn` in Hy.

Most of the main programming languages supports anonymous functions, with notable exceptions of Ada, C, COBOL, Fortran, and Pascal.

```
(eval-and-compile
  ; specify separator char
  (setv separator '·)
  ; find the index of the element from the list
  (defn index [elm lst]
    ; if the element is not found, return -1
    (try (.index lst elm) (except [ValueError] -1))))
; main lambda expression macro
(defmacro  [&rest expr]
  ; get the index of the argument-body separator
  (setv idx (index separator expr))
  ;  cut the arguments before the separator and append to the function
  `(fn ~(cut expr 0 (if (pos? idx) idx 0))
    ; cut the body of the expression and append to the function
    ~@(cut expr (inc idx))))
```

```
#! /usr/bin/env hy
; native function utilizer by lambda calculus syntax in Hy
(eval-and-compile
    ; specify separator char
    (setv separator '·)
    ; find the numerical index of the element from the list
    (defn index [elm lst]
      ; if the element is not found, return -1
      (try (.index lst elm) (except [ValueError] -1))))
  ; main lambda expression macro
  (defmacro λ [&rest expr]
    ; get the index of the argument-body separator
    (setv idx (index separator expr))
    ;  cut the arguments before the separator and append to the function
    `(fn ~(cut expr 0 (if (pos? idx) idx 0))
      ; cut the body (And the rest of the arguments) of the expression and append to␣
↪the function
      ~@(cut expr (inc idx))))
```

In Hy anonymous function is created with `(fn [args] body)`. Because Hy is Lisp at frontend, evaluation order of the elements in the program expression is very similar to Lambda calculus syntax. The first element will be the function and the rest of the elements are arguments to the function, where arguments can of cource be functions themselves.

So the usage of the anonymous function in Hy is:

((fn [args] (print args)) ‘args)

From that point of view, it really is just a matter of implementing Lambda calculus syntax to functionality, that already exists in Hy.

http://docs.hylang.org/en/stable/language/api.html#fn

1.7 Evaluation stages

Next we need some evaluation rules to call the function with given input and give the result. These rules or procedures are called:

- alpha conversion
- beta reduction

Optional:

- eta conversion

The most of the modern computer languages utilizes some notation of functions. More precicely, anonymous functions that are not supposed to be referenced by a name in a computer program, at first seems to be equivalent to Lambda calculus. But there are some catches one needs to be aware of.

1.8 Initialization macros

After importing `calchylus` module with `(require [calchylus.lambdas [*]])`, the system itself is initialized by one of the four start up macros:

- `with-macros` that loads macro shorthands
- `with-alpha-conversion` that makes argument renaming behind the curtain
- `with-alpha-conversion-and-macros` that loads both features
- `with-alpha-conversion-nor-macros` that loads neither one

`with-alpha-conversion-and-macros` is the recommended way of using the `calchylus` module, because it will provide automatic alpha conversion for variable names and import useful shorthands for dozens of well known custom lambda forms.

There are two parameters you must use on initialization macros to specify the syntax of the Lambda expressions:

1. lambda function identifier (more formally called the binding operator)
2. argument and body separator (delimitter)

Identifier is usually the Greek lambda letter λ, but it can be any character or string you desire. One can find letters like `\`, `^`, `|lambda|`, and `L` used on different Lambda calculus implementations.

In Hy, even unicode letters are supported, which is sometimes typographically satisfying for printing purposes. Writing unicode letters from the keyboard however, can be tricky. Most probably, one needs to rely on extensive copy and paste, if an unicode identifier and separator is used.

For a demonstration, let us load the full recommended set from the `calchylus` module:

```
(with-alpha-conversion-and-macros λ ·)
```

There will be an output after the successful initialization, which indicates the last created lambda function, something like:

```
<function <lambda> at 0x000001790B7208C8>
```

Now we can start evaluating Lambda expressions with the given identifiers:

```
(λ x · (λ y · (y x)) 'first 'second)
```

[output]

```
(second first)
```

Althought using `with-alpha-conversion-and-macros` is the recommended, for efficiency, testing, and benchmarking purposes one would sometimes want to load only macros to the global Hy environment by `with-macros` initializer.

Similarly, by initializing `with-alpha-conversion`, macros are discarded but alpha conversion is activated. In that case you cannot use macro shorthands on Lambda expressions, which may or may not be a good idea, depending on your purpose.

Lastly, there is an option to do neither one by `with-alpha-conversion-nor-macros`. You will like hit some problems like recursion error if using this with complex Lambda expressions. With a very careful argument name selection you could pass these problems, but one should note that there aren't any internal warnings triggered if argument name collision happens. Similarly, if there are any arguments that are not bound, not used, or not replaced, system is mute with it.

Example from notebook...

1.9 Macro shorthands

Whilst in Lambda calculus there is no limit on how many or what kind of forms one can create, there is a set of common forms useful for constructing Lambda expressions. Named forms are provided as macros in Hy based `calchylus` module and they serve for shorthands when coding in Lambda calculus. Named forms are useful in explaining Lambda calculus and they make expressions more compact, readable, and understandable.

This is the list of the all available macros for Lambda forms in `calchylus` module:

Basic constructors

- `APP` - an application
- `CONST` - a constant
- `IDENT` - an identity

Boolean constructors

- `TRUE` - an arbitrary tautology
- `FALSE` - an arbitrary contradiction

Logical connective constructors

- Unary
  - `NOT` - a negation
- Binary
  - `AND` - a conjuction
  - `OR` - a disjunction
  - `XOR` - an exclusive disjunction
  - `IMP` - a material implication
  - `EQV` - an equivalence / a material biconditional

Structural constructors

- `COND` - a condition block for flow control

- LET - introduce a variable/variables with a value / values, last term is the function body!
- LET* - same as LET, but consequencing variables can use former variables in the body
- DO - do things in sequence, similar to LET*, but except setters are disclosed

2-tuple constructor

- PAIR - a nested ordered pair
- HEAD - a head of the pair
- TAIL - a tail of the pair

List constructors

- LIST - create list with sequential items, generates nested pairs from given terms
- PREPEND - prepend to the beginning of the list
- APPEND - append to the end of the list
- FIRST - the first item of the list
- SECOND - the second item of the list
- LAST - the last item of the list
- LEN - length of the list
- EMPTY? - is given term an empty list?
- NIL? - is given term nil?
- Internal
  - NIL - a nil for the empty set
  - - an empty set

Church numerals

- NUM - a Church numeral generator
- ZERO - the number zero, same as FALSE
- numerals from one to ten
  - ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE TEN
- NUM? - is given term a number, unary predicate checker for Church numbers

Number equivalence

- ZERO? - is number zero?
- EQ? - are two numbers equal?
- LEQ? - is number a equal or smaller than b?
- GEQ? - is number a equal or greater than b?
- LE? - is number a less than b?
- GE? - is number a greater than b?

Arithmetic constructors

- SUCC - a successor of a number
- PRED - a predecessor of a number

- `SUM` - a summa of two numbers
- `SUB` - a substraction of two numbers
- `PROD` - a product of two numbers
- `EXP` - the nth power of number x

Recursive constructors

- `SELF` - a self application
- `YCOMB` - an Y combinator

Sample mathematical functions

- `SUMMATION` - the nth triangular number
- `FACTORIAL` - a product of numbers up to n
- `FIBONACCI` - the nth Fibonacci number

1.10 Tests

Church numerals are one of the most common number representations in Lambda calculus.

just the Lambda term, as it is more conventionally called

Undecidability proof of the halting problem using lambda calculus

https://yinwang0.wordpress.com/2012/10/25/halting/

Computability and Complexity - From a Programming Perspective Neil D. Jones

http://www.diku.dk/~neil/comp2book2007/book-whole.pdf

Lambda Calculus, Prof. Tobias Nipkow (2012)

https://www21.in.tum.de/teaching/logik/SS13/lambda-en.pdf

Decidability for Non-Standard Conversions in Typed Lambda-Calculi, Freiric Barral (2008)

http://www2.tcs.ifi.lmu.de/~barral/doc/BarralThesis.pdf

Short Introduction to the Lambda-calculus, Franco Barbanera ()

http://www.dmi.unict.it/~barba/LinguaggiII.html/READING_MATERIAL/LAMBDACALCULUS/LAMBDACALCULUS.1.HTM

Collected Lambda Calculus Functions

http://jwodder.freeshell.org/lambda.html

Deriving Recursive Programs

http://faculty.ycp.edu/~dhovemey/fall2011/cs340/lecture/lecture14.html

©

1.11 References

No work could be done without the work of previous ones. Here are some references that has been used with the implementation and the documentation of the `calchylus` module:

Lambda Calculus with Types by Henk Barendregt, Wil Dekkers, Richard Statman

Concepts in Programming Languages by John C. Mitchell

Lecture Notes on the Lambda Calculus by Peter Selinger

A Brief History of Computing by Gerard O'Regan

The Lambda Calculus in the Stanford Encyclopedia of Philosophy.