Introduction to Lambda Calculus

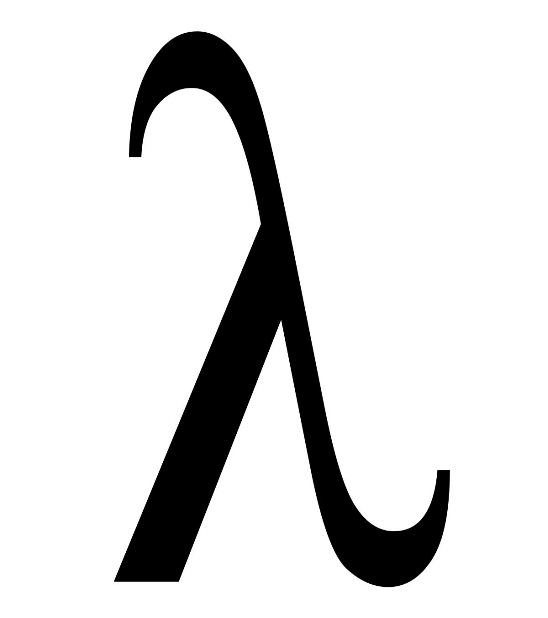
CSCI 3434 Final Presentation

Gustav Cedergrund, Luke Sellmayer, Zane Perry

Outline

Lambda Calculus - “The Smallest Universal Programming Language in the World”

- ➢ Developmental History
- ➢ Description
- ➢ Modern Connection



Origins + Development

Lambda Calculus was developed by Alonzo Church in the 1930s.

Created to investigate nature of computation & the Entscheidungsproblem (undecidability problem)

Also a significant development in Mathematics as it gave a formal way of defining functions.

What is Lambda Calculus?

Lambda Calculus is *a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application.*

Universal model of computation - basis for the theoretical limit of Computability.

Turing Machines and Lambda Calculus are equal in computational power

Church-Turing Thesis -

“Every effectively *calculable* function is a *computable* function.”

Components

$$E \rightarrow x \mid \lambda x.E \mid EE \mid (E)$$

Variables

- Represent all basic primitive types
- Independent of name
- Free or bound

Abstractions

- Function Definitions
- “Binding” of variables
- Define the scope of a variable

Applications

- Calling a function
- Expression is applied to another

Notational Clarity

- Parentheses keep expressions separate
- Curried Functions

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

Reductions

$$\lambda x.x \longleftrightarrow \lambda y.y$$

$$(\lambda x.e_1)e_2 \longleftrightarrow [e_2/x]e_1$$

$$\lambda x.(ex) \longleftrightarrow e \quad \text{if } x \notin fv(e)$$

Goal: Reach Normal Form

α - reductions (renaming)

- Mostly used to maintain function scope

β - reductions (substitutions)

- Used for applications
- Replaces all instances of a variable
- Normal Order vs Applicative Order (ambiguity)

η - reductions (simplifications)

- Not as common
- Not used for actual computation

Boolean Logic

- Many different possible definitions for primitives
- Not defined intuitively, but functionally

$$\mathbf{T} \equiv \lambda xy.x$$

- Returns the first argument of an application
- Returns the constant function with only one application

$$\mathbf{F} \equiv \lambda xy.y$$

- Returns the second argument of an application
- Returns the identity function with only one application

$$\wedge \equiv \lambda xy.xy\mathbf{F}$$

$$\vee \equiv \lambda xy.x\mathbf{T}y$$

$$\neg \equiv \lambda x.x\mathbf{F}\mathbf{T}$$

- Analyze the behavior of the *x* variable

- Allows for the definition of if-then-else blocks

Church Numerals and Arithmetic

$$\begin{aligned} \mathbf{0} &\equiv \lambda fx.x \\ \mathbf{1} &\equiv \lambda fx.f(x) \\ \mathbf{2} &\equiv \lambda fx.f(f(x)) \\ &\vdots \\ \mathbf{n} &\equiv \lambda fx.f^{(n)}(x) \end{aligned}$$

- Curried function that results in n applications of a function
- Can be applied to a function and argument for repeated calls
- 0 only returns the second argument

$$\mathbf{n}fa \to (\lambda gx.g^{(n)}(x))fa \to f^{(n)}(a)$$

$$\mathrm{S} \equiv \lambda xyz.x(xyz)$$

$$\mathbf{S0} \to \lambda yz.\mathbf{0}(\mathbf{0}yz) \to \lambda yz.z \to \mathbf{1}$$

- Based on the behavior of the expression *x*

$$\mathbf{n} + \mathbf{m} = \mathbf{n}S\mathbf{m}$$

- Repeated application of the successor function

$$\mathbf{n} \times \mathbf{m} = \mathbf{n}(\mathbf{m}S)\mathbf{0}$$

- Repeated application of the addition function

Recursion

Base Cases:

$$\mathbf{Z} \equiv \lambda x.x\mathbf{F}\neg\mathbf{F}$$

- Tests if the expression *x* is equal to 0
- FALSE applied 0 times results in NOT
- Otherwise results in the identity

Further Extensions:

- Pairs and Tuples
- Predecessor Function (and Division)
- Equalities/Inequalities
- Integers/Real Numbers

Y Combinator:

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$\mathbf{YR} \to (\lambda x.\mathbf{R}(xx))(\lambda x.\mathbf{R}(xx)) \to \mathbf{R}((\lambda x.\mathbf{R}(xx))(\lambda x.\mathbf{R}(xx))) \to \mathbf{R}(\mathbf{YR})$$

- Strategy to write recursive functions in a non-recursive way
- Function body has no normal form
- Leads to Curry's paradox
- Extends to programming but disqualifies from mathematical consistency

Self Encoding

- Lambda expressions cannot inspect their own arguments
    a. Ex: the function *isVariable*, called with argument a, that returns *True* if a is a variable and *False* otherwise, does not exist
    b. Problematic: lambda calculus seems to have no input validation
- In order to have a self-referential lambda expression a, we need to call it with an argument that is an encoding of itself, $<a>$

One possible encoding scheme to create any $<a>$ uses the following four lambda functions for encoding

$$var\ \lambda f.\lambda x.f^{(i)}(x) \to \langle x_i \rangle$$

$$app\ \langle a \rangle\ \langle b \rangle \to \langle a\ b \rangle$$

$$lam\ \langle a \rangle\ \ \lambda f.\lambda x.f^{(i)}(x) \to \langle \lambda x_i.\, a \rangle$$

$$wrap\ \langle a \rangle \to \langle \langle a \rangle \rangle$$

<u>All four of these lambda functions are one-to-one</u>

Self Encoding

We also need the *self* function in order to encode *<a <a>>*

$$self\ x \equiv app\ x\,(wrap\ x)$$

$$self\ \langle a \rangle \rightarrow \langle a\ \langle a \rangle \rangle$$

Now we can encode our lambda expressions to pass them as arguments to themselves

More Undecidability

Definition: a lambda expression *p* is a decider for a set of lambda expressions *A* if for any lambda expression *a*, applying *p* to *<a>* returns *True* if *a* is in *A* and *False* otherwise

The ability to encode lambda expressions allows us to prove the following theorems:

1. *Any set of lambda expressions which is non-trivial and closed does not have a decider.*
2. *It is undecidable if two arbitrary lambda expressions a and b are beta-equivalent*
3. *It is undecidable if an arbitrary lambda expression a reduces to a normal form (equivalent to the Halting problem)*

Church’s Thesis

Church-Rosser Theorems

- If two expressions are intra-convertible, there exists a third expression that they can both be reduced to (can be unchanged, but unique)
- If a normal form of an expression exists, it can always be found (use normal-order)

Equivalency of Models

- Lambda Definable Functions (Church)
- Turing Computable Functions (Turing)
- General Recursive Functions (Gödel)

- All used to show the unsolvability of the Entscheidungsproblem

General Outline of Proof:

- Lambda abstractions to represent the transition function (applied to initial state)
- States to represent variables

“Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus”

Example Lambda Encoding

Objective - Sort a list

"Imperative" Code

Algorithm 1 - Sort(list A)

pivot = first element of A
smaller, equal, greater = [], [], []
for each $k \in A$ **do**
  if $k <$ pivot **then** append k to smaller
  else if $k >$ pivot **then** append k to greater
  else if $k =$ pivot **then** append k to equal
  end if
end for
sortedSmaller = Sort(smaller)
sortedGreater = Sort(greater)
return concatenate(sortedSmaller, equal, sortedGreater)

Lambda Calculus Expression

$$\textbf{Sort} = \lambda bcd.(\lambda fx.f(xx))(\lambda fx.f(xx))(\lambda ef.f(\lambda ghi.g(\lambda j.h(\lambda kl.kj(ikl)))(hi))e \ldots (\lambda gh.h))(\lambda e.d)(\lambda e.b(\lambda f.e(f(\lambda ghi.hg)(\lambda gh.cfh))))$$

Operates on Church Encoded list of numbers
Example of [1,7,2,3] is seen below

$$\begin{aligned}
&(\lambda \text{cons nil}.\\
&\quad (\text{cons}(\lambda fx.(fx)) - 1\\
&\quad (\text{cons}(\lambda fx.(f(f(f(f(f(f(fx)))))))) - 7\\
&\quad (\text{cons}(\lambda fx.(f(fx))) - 2\\
&\quad (\text{cons}(\lambda fx.(f(f(fx)))) - 3\\
&\ \text{nil})))))
\end{aligned}$$

Reference: https://codegolf.stackexchange.com/a/55604

Modern Connection - Functional Programming

Lambda Calculus is a basis for more modern “functional programming” languages.

Created by Mccarthy in late 1950’s with LISP. Modern examples include Haskell, ML, OCaml, Scala.

Properties:

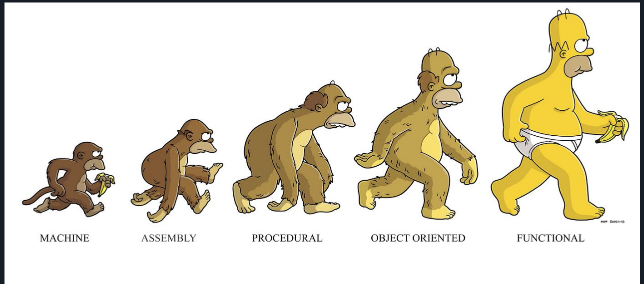
- Declarative style
- Built on immutability (value binding with beta reductions)
- Higher-order functions (currying with functions as input).

Anonymous (Lambda) functions more recently implemented in many imperative programming languages. Examples in Python, C++, MATLAB, etc.

Functional Programming in Industry

Extensions of Lambda Calculus in use today:

- **Google -** Haskell used for internal IT infrastructure support
- **Microsoft** - uses functional programming for production serialization system, Bond
- **Discord** - built almost entirely on Elixir
- **AT&T** - Haskell used in the Network Security division to automate processing of internet abuse complaints
- **Ericsson** - developed Erlang function programming language in the 1980s. Used today by companies like Ericsson, WhatsApp, and Goldman Sachs
- **Intel** - Haskell compiler as part of research on scaled multicore parallelism



Conclusions

Turing machines can be seen as representing computer hardware

- Makes them ideal for measuring resources used during an algorithm
- Ideas/fields developed from Turing machines:
  - Complexity theory
  - P v. NP
  - RAM models
  - Hardness of approximation

Lambda calculus can be seen as representing computer software

- Makes them ideal for analyzing the structure of an algorithm
- Ideas/fields developed from lambda calculus:
  - Type theory
  - Implicit-memory management
  - Polymorphism
  - Proof-checkers

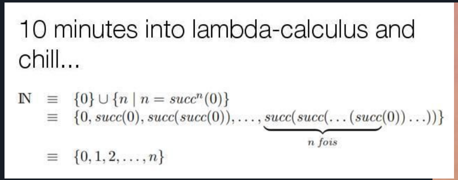
Despite being used for different applications, they both represent the limits of computability

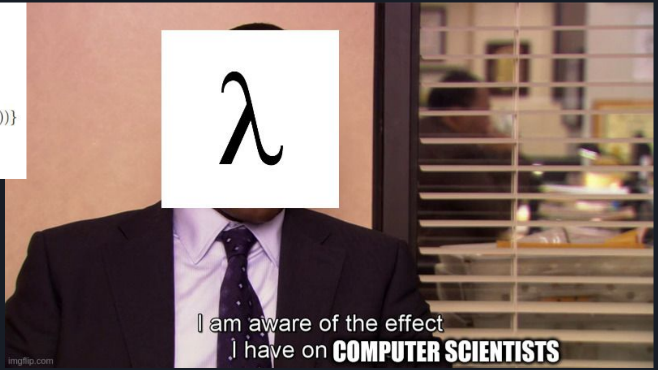
Q&A

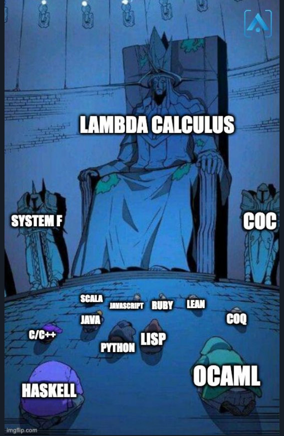
Any Questions?

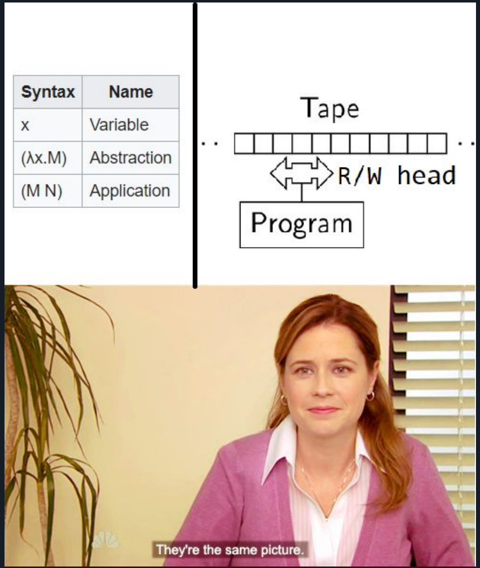
References

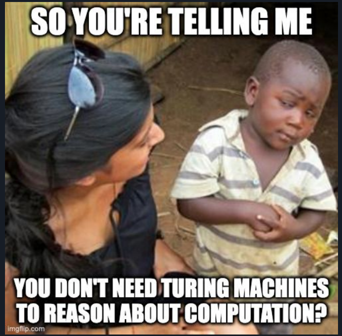
- Jung, A. (2004). A short introduction to the Lambda Calculus. University of Birmingham. https://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf
- Rojas, R. (2014). A Tutorial Introduction to the Lambda Calculus. Freie Universität Berlin, Version 2.0. https://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/lambda.pdf
- Scott, D. (2013). Lambda Calculus: Then and Now. Carnegie Mellon University. https://turing100.acm.org/lambda_calculus_timeline.pdf
- Brandl, H. (n.d.). Limits of computability in lambda calculus. https://hbr.github.io/Lambda-Calculus/computability/text.html
- Hudak, P. (1989). Conception, evolution, and application of functional programming languages. ACM Computing Surveys, 21(3), 359–411.
- Bibliotheca Mathematica. Science 68, 474-475(1928). DOI:10.1126/science.68.1768.474.bhttps://doi.org/10.1145/72551.72554
- Turing, A.M. (1937), On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, s2-42: 230-265. https://doi.org/10.1112/plms/s2-42.1.230
- Haskell in industry - HaskellWiki. (n.d.). https://wiki.haskell.org/Haskell_in_industry
- Hughes, J. (1990). Why Functional Programming Matters. “Research Topics in Functional Programming,” 17–42. https://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf











Lambda Calculus Memes