Introduction to Barendregt's Lambda Cube

Silvia Ghilezan

June 2023

1 Background & Introduction

In 1928, David Hilbert and Wilhelm Ackerman proposed challenge called Entscheidungsproblem, or Decision Problem, which asks for the following:

> "Given all the axioms of math, is there an algorithm that can tell if a proposition is universally valid i.e. deducible from the axioms?"

The answers were published just a few years later with both Alonzo Church and Alan Turing. No! There is no such algorithm. During the 1930s, λ-calculus , Turing Machines and combinatory logic were introduced, as complete models of computation. They were later proven to be equivalent (Turing complete).

2 Untyped λ-Calculus

λ-calculus is a successful model for computable functions. It is known for its simplicity, yet expressiveness, being able to represent any computation problem. You can represent untyped λ-calculus using.

$$M ::= \quad x \mid c \mid (M\, M) \mid (\lambda x.M)$$

where:

1. $V = x, y, z, x_1, \ldots$ a countable set of variables
2. $C = a, b, c, a, \ldots$ a countable set of constants.

NOTE: pure λ-calculus has no constants.

Functions bind variables and represent a computation. A term x is bound in M if it appears in a subterm of the form λ.N. Any unbounded variable is considered a free variable.

2.1 Reductions

2.1.1 α-reductions

$$\lambda x.M \rightarrow_\alpha \lambda y.M[x := y],\ y \notin FV(M)$$

α-reductions are used for renaming variables, since names of bounded variables are irrelevant. λx.$(x^2 + 1)$ and $\lambda.(y^2 + 1)$ must be considered as equal. It also characterizes an equivalence relation.

2.1.2 β-reductions

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

β-reductions are used for function applications. Applying N to $\lambda x.M$ using the past example: $(\lambda x.x^2{+}1)5 \rightarrow_\beta 5^2{+}1 = 26$. It characterizes to an equivalence relation and a symmetric closure.

2.2 Properties

2.2.1 Confluence

Theorem 1. *If $M \rightarrow N$ and $M \rightarrow P$, then there exists some S such that both $N \rightarrow S$ and $P \rightarrow S$.*

While the proof is beyond the current scope, there is a very important corollary associated with the confluence theorem.

Corollary 1. *If $M \rightarrow N$ and $M \rightarrow P$, then $P = Q$.*

The Corollary implies that order of the applied reductions is arbitrary and always leads to the same result. Furthermore, it allows concurrency.

2.2.2 Normal Forms

Definition 1. *$N \in \Lambda$ is a normal form (NF) if there is no S such that $N \rightarrow S$.*

Definition 2. *$P \in \Lambda$ is strongly normalizing (SN) if all reductions of P are finite.*

The concept of SN was one of the motivators for Church to add types to λ-calculus, as we will discuss on the following chapter.

2.2.3 β-normal forms

Expression	Description
$xyzx$	Normal form (NF),
$I \equiv \lambda x.x$	Normal form (NF)
$K \equiv \lambda xy.x$	Normal form (NF)
$S \equiv \lambda xyz.xz(yz)$	Normal form (NF)
$KI(KII)$	Normal form (NF)
$\Omega \equiv \Delta\Delta \equiv (\lambda x.xx)(\lambda x.xx)$	strongly normalizing (SN), unsolvable
$KI\Omega$	Normalizing (N)
$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$	Head normalizing (HN), solvable

The one worth highlighting the most is the self application (Ω). It is unsolvable because it loops forever:

$$\Omega \to \Omega \to \Omega \to \Omega \to \dots$$

2.3 Logic and Arithmetic

As we previously mentioned, λ-calculus is able to represent all computable functions. Therefore there has to be a way to represent propositional logic and numerals/arithmetic.

2.3.1 Propositional logic in λ-calculus

$\top := \lambda xy.x$
$\bot := \lambda xy.y$
$\neg := \lambda x.x\bot\top$
$\wedge := \lambda xy.xy\bot$
$\vee := \lambda xy.x\top y$

2.3.2 Arithmetic in λ-calculus

add $:= \lambda xypg.xp(ypq)$
mult $:= \lambda xyz.x(yz)$
succ $:= \lambda xyz.y(xyz)$
exp $:= \lambda xy.yx$
iszero $:= \lambda n.n(\top\bot)\top$

The best reference to get to know more about untyped lambda calculus is H.P. Barendregt's "Lambda Calculus: Its syntax and semantics." book.

3 Simply Typed Lambda Calculus

The Simply Typed Lambda Calculus (STLC) is motivated by some disadvantages of the untyped lambda calculus:

- Terms like Ω do not have a normal form
- You can end up with meaningless terms like **sin log** because there's no way to restrict the arguments you provide to functions

We want types so that we can limit the computations we carry out to the ones that are "safe" and make sense.

There are generally two approaches to equipping lambda calculus with type systems: Curry's approach, which implicitly assigns types to terms, and Church's approach, which does so explicitly instead.

3.1 Syntax of Types

The type system can be defined over the signature $(T, \rightarrow)$, where T is some countable set of variables - the *base types* or *atomic types*:

$$\sigma ::= \alpha \in T \mid (\sigma \rightarrow \sigma)$$

where $\rightarrow$ is the function type and is **right-associative**. $\sigma \rightarrow \tau$ refers to the type of functions that take an input of type σ and produce an output of type τ.

3.2 The Language

3.2.1 Type Assignment

A type assignment is an expression of the form

$$M : \sigma$$

where M is a λ-term and σ is a type. The type σ is assigned to the term M.

3.2.2 Declaration

A declaration is a type assignment where M (the term) is a variable:

$$x : \sigma$$

3.2.3 Basis (AKA context AKA environment)

A basis $\Gamma = \{x_1 : \sigma_1 \dots x_n : \sigma_n\}$ is a set of declarations in which all the variables are different.

3.2.4 Statement AKA Sequent

A statement of the form

$$\Gamma \vdash M : \sigma$$

is saying that the expression M has type σ under the context Γ. M is said to be well-typed (with type σ).

3.3 Typing Rules

$$(Ax) \qquad \frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

The axiom states that if x has type σ in the context, then x has type σ.

$$(\rightarrow_{elim}) \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

The elimination rule pertains to applications, and states that if, in a certain context, M has a function type $\sigma \rightarrow \tau$ and N has type σ, then the application of M to N (i.e. MN) has type τ. It gets its name because it eliminates the $\rightarrow$ in the types of terms.

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \rightarrow \tau} (\rightarrow_{intr}) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x . M : \sigma \rightarrow \tau}$$

à la Church à la Curry

The introduction rule (so called because it introduces a $\rightarrow$), says that if, in a certain context where x has type σ, M has type τ, then in the same context **without x**, $\lambda x : \sigma . M$ has type $\sigma \rightarrow \tau$.

3.4 Fundamental properties of STLC

3.4.1 Uniqueness of types

Not every term is typeable, but if a term has a type, then the type is unique. More formally -

$$\Gamma \vdash M : \sigma \text{ and } \Gamma \vdash M : \tau \implies \sigma = \tau$$

3.4.2 Church-Rosser Property

When reducing terms, the order in which reduction rules are applied does not affect the eventual result. More formally, for beta reduction:

$\forall M, N_1, N_2 \in \Lambda :$ if $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then $\exists X \in \Lambda : N_1 \twoheadrightarrow_\beta X$ and $N_2 \twoheadrightarrow_\beta X$

3.4.3 Type preservation under reduction

Reduction or evaluation of terms does not change their type.

$$\text{if } M \rightarrow P \text{ and } M : \sigma \text{ then } P : \sigma$$

3.5 Strong Normalization

Strong normalization says that every well-typed term in STLC is strongly normalizing, i.e. it cannot reduce indefinitely. Every reduction of the term must end in a value.

$$\text{If } M : \sigma \text{ then } M \text{ is strongly normalizing.}$$

3.6 Expressiveness

Not every term is typeable - one example is self-application.

$$\not\vdash \lambda x.xx : \sigma$$

Intuitively, self application cannot be typed in the STLC because we have to assign the same type to both bounded occurrences of x. If we assign some type σ to the second occurrence of x, then in order to end up with a well-typed term, you have to assign the type $\sigma \to \sigma$ to the first occurrence. However, σ and $\sigma \to \sigma$ cannot be the same type, leading to a contradiction.

An interesting result is that natural numbers can be encoded, by terms of the type $(\sigma \to \sigma) \to \sigma \to \sigma$. The natural number n is encoded by a lambda abstraction that, given a function and an argument, applies the function to the argument n times.

$$n \equiv \lambda f \lambda x.f^n x : (\sigma \to \sigma) \to \sigma \to \sigma$$

Another interesting result is that when equality is taken as beta-conversion and one interprets integers over the type $(\sigma \to \sigma) \to \sigma \to \sigma$ where σ is a base type, the integer functions definable in the STLC are exactly the extended polynomials.

3.6.1 What are the extended polynomials?

The class of extended polynomials is the smallest class of functions over $\mathbb{N}$ containing

- the constant functions 0 and 1
- projections
- addition
- multiplication
- the function ifzero(n, m, p): if $n = 0$ then m else p

3.7 STLC, Logic, and the Curry-Howard Correspondence

Under the Curry-Howard Correspondence, the simply-typed lambda calculus corresponds to minimal logic. A formula is provable in minimal logic iff it is inhabited in STLC.

3.7.1 What is the Curry-Howard correspondence?

At a high level, the Curry-Howard correspondence is the observation that there is an isomorphism between proof systems and models of computation. A common generalization is that *a proof is a program, and the formula it proves is the type of the program.*

3.7.2 What is minimal logic?

Minimal logic is intuitionistic logic without the *principle of explosion* ($\bot \vdash P$). The principle of explosion is the law that from a contradiction, any statement can be proven.

3.7.3 What is intuitionistic logic?

Intuitionistic logic is classical logic without the principle of the excluded middle ($A \vee \neg A$) or the double-negation rule ($\neg\neg A = A$).

3.8 Decidability properties

For STLC, type checking, type inference, and type inhabitation are all **decidable**.

3.8.1 Type checking

Given a term M and type σ, does M have type σ?

$$((M : \sigma)?)$$

3.8.2 Type inference

Given a term M, what is its principal type?

$$(M :?)$$

This is a functional problem, since it asks to find the principal type of M. As a decision problem, this would be framed as "Given term M and type σ, is σ the principal type of M?"

3.8.3 Type inhabitation

Given a type σ, does there exist some term M that has type σ?

$$(? : \sigma)$$

3.9 Summary

Advantages of STLC

- All terms are strongly normalizing
- Type inference, type checking, and type inhabitation are all decidable
- Types exactly all extended polynomials

Drawbacks of STLC

- No self-application
- No recursion
- No total function
- Not Turing-complete

References

[Bar93] H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993.

[BDS13] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[NG14] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[nLa23a] nLab authors. intuitionistic logic. `https://ncatlab.org/nlab/show/intuitionistic+logic`, June 2023. https://ncatlab.org/nlab/revision/intuitionistic+logic/25Revision 25.

[nLa23b] nLab authors. minimal logic. `https://ncatlab.org/nlab/show/minimal+logic`, June 2023. https://ncatlab.org/nlab/revision/minimal+logic/22Revision 22.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, February 2002.

[Wik23] Wikipedia contributors. Simply typed lambda calculus — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Simply_typed_lambda_calculus&oldid=1149705670`, 2023. [Online; accessed 29-June-2023].

[Zak07] Mateusz Zakrzewski. Definable functions in the simply typed lambda-calculus, 2007.