An Introduction to the Invariance Thesis in Lambda Calculus

Haileselassie Gaspar

Supervisors: Femke V. Raamsdonk, Jörg Endrullis

June 18, 2025

1 Abstract

– –

2 Introduction

During the start of the 20th century, the idea of computability started to be a main mathematical problem. David Hilbert posed the question in his set of problems to solve during the 20th century: What does it mean for a function to be *computable*? The first step in answering this question would be to define the concept of *computability*. Both Alonzo Church and Alan Turing proved almost simultaneously that there existed a model defining the group of computable functions, the former through general recursive functions and the latter with the conceptual "Turing" machines. The analysis of computability and complexity of functions and algorithms is done through a computational model, or *machine model*.

2.1 Machine models

While there are many models of computation, for the context of this paper it is only necesary to introduce two.

Turing Machines In 1936, Alan Turing proposed a model now referred as a *Turing Machine* in order to analyze the halting problem [8]. This machine has an unlimited memory in the form of a tape, and a set of symbols already present on the tape. It uses a head to read and write symbols from the tape, and a transition function determines where the head moves after a read, either left or right. This is the main computational model most used when talking about complexity of algorithms [7].

Lambda calculus The lambda calculus was conceived as a foundational system for mathematics and logic in the 1930s by Alonzo Church. Although this initial idea was proved inconsistent by Kleene and Rosser in [5]. This led to Church publishing in 1936 a simplified version of this system with a focus on computability, now called the *untyped lambda calculus*. A more formal introduction to this system will be provided in section3.1. *lambda*-calculus can also be seen as a Term Rewriting System, another concept that will be introduced in detail in section**??**.

2.2 Equivalence and Invariance

The importance of Turing machines and their relation to modern computing has made them a central model in the study of computational complexity, in particular when discussing time and space complexity. This has made them a good model by which to measure these qualities, specially since most computational classes are defined in their context (P, NP). Therefore, the equivalence and simulation of other models in TMs is one of the main qualities studied in complexity theory. First, it is important to note the difference between equivalence and completeness, two terms very often used in the literature of computation. Completness refers to the ability of any model to simulate a Turing machine. For those acquainted with languages, one could define completeness as:

$$\text{for a model } \mathcal{M}, \text{ and a deterministic turing machine } TM,$$
$$L(\mathcal{M}) \supseteq L(TM)$$

Equivalence however, is a much more restrictive set. As its name implies, it is the class of every model that is able to compute exactly what a turing machine can. In the previous notation:

$$L(\mathcal{M}) = L(TM)$$

When we talk about invariance we refer to it in the sense of Van Embde Boas [4]:

> *"Reasonable" machines can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space.*

This is an extension to the idea of equivalence between computational models. It proposes that not only is there a method in which machine models can simulate each other, but that there exists such a method so that the overhead is polynomial in time and constant in space. In other words, it would only take a polynomial amount of extra time and a constant amount of extra space to evaluate a term in *lambda*-calculus than in a Turing machine and viceversa. This paper will use the *weak* invariance thesis, meaning the space requirements, a long-standing problem with lambda calculus, will be dropped. In [2], it is proven that there is an implementation of Turing machines in the deterministic λ-calculus, with only a linear overhead in time when only head reduction is considered. This paper will introduce the reader to many terms and concepts necessary to understand complexity theory, but it should be

viewed as nothing more than a small introduction to the subject with an interesting example, the invariance of the complete untyped λ-calculus with regards to Turing machines.
In order to acquire a more in depth understanding of the topic, refer to the references, that although incomplete, may provide a more formal introduction into the subject.

2.3 Introduction to Term Rewriting

Since this paper is based in great part in the λ-calculus, it is useful to introduce the idea of it not only as a specific set of terms and equations in an unfamiliar form to anyone without a background in theoretical computer science, but as a member of a greater group of systems that rely on the idea of reduction as a relation on terms. This group is referred as Term Rewriting Systems, and it originated on predicate logic, and was first formally defined by Axel Thue in a paper in 1910.
It builds on the notion of "directed" computation, in the sense that the relations defined in such systems ar unidirectional unless otherwise specified. These systems can therefore represent programs by using the notion of reductions as a correspondance to the idea of program evaluation. An interesting result, is the fact that there exists a class of Term Rewriting Systems (TRS for short) that are Turing Complete. It was later proven by Rosser and Kleene that λ is also Turing Equivalent, and therefore, the idea of invariance is not out of the realm of the impossible. Since both models define the same group of functions, would it not follow that there is an algorithm that can "reasonably" simulate each of them?
The problem with this, is the fact that the main computational step in λ, the β-reduction step, is not a unitary measure of complexity costs. There exists a family of terms and subterms that in time linear to the size of themselves, produce an exponential output. This is solved in [2] and [1] by the use of sharing. This paper aims to provide a somewhat simpler version of the results of both papers, and demonstrate it through an example, a member of this size exploding family.

3 Theoretical Background

3.1 Introduction to lambda calculus

While an introduction to TRS is a proper base to start understanding the concepts of λ, the idea of it predates that of TRS.
The idea of λ-calculus, or its current use, is based on the idea of computability of functions. When talking about functions there are two main ways to view them. The *extensional* view, which observes only the mapping from input to output, and the *intensional* view, which treats functions not as just a mapping, but a rule. This means that if two functions are given by the same formula, they are *intensionally equal*. This allows mathematicians and computer scientists to talk about the behaviour of a function outside of just what it produces [6].

The λ-calculus uses the intensional view of functions to treat them as expressions, and analyze their behaviour and, more importantly, their computability. It is important to notice that since Turing machines and the λ-calculus treat functions *intensionally*, most of the proof is centered around the process of computation. In order to talk about the invariance of λ-calculus it is first necesary to define some notation that will be used in this paper.

Terms Let $M, N, P....$ denote arbitrary λ-terms, and $x, y, z...$ denote variables. The set of λ-terms Λ is inductively defined as:

$$\begin{aligned} &\text{Variables: } x \in \Lambda \\ &\text{Abstraction: } M \in \Lambda \implies (\lambda x.M) \in \Lambda \\ &\text{Application: } M, N \in \Lambda \implies (MN) \in \Lambda \end{aligned} \tag{1}$$

Free Variables $FV(M)$ is the set of free variables in M and it includes every variable in M not bound by an abstraction.
A context is a lambda term with a hole in it that can be replaced by another term or a context. They are defined as:

$$C ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \tag{2}$$

Substitution This topic will be expanded on section 3.1.2, but a basic understanding is provided here. The result of substituting N for the free ocurrences of x in M (notation $M\{x := N\}$) is inductively defined as:

$$\begin{aligned} &x\{x := N\} \equiv N \\ &y\{x := N\} \equiv y \text{ iff } x \neq y \\ &(\lambda y.M_1)\{x := N\} \equiv \lambda y.(M_1\{x := N\}) \\ &(M_1M_2)\{x := N\} \equiv (M_1\{x := N\})(M_2\{x := N\}) \end{aligned} \tag{3}$$

Note that we use braces to denote *implicit* substitutions due to the fact that when introducing *explicit* substitutions we will use the classical brackets, but when reading literature of the classical lambda-calculus, square brackets will mean implicit substitution. For further reading on the syntax and axioms of the lambda calculus, refer to [3].

3.1.1 Conversion

This section will be introduced non-mathematically, since an intuitive understanding of α conversion suffices to understand the main points of the proof. However, to understand the more finer points, the reader should again refer to [3]. When referring to conversion in the λ-calculus, it is usually in the context of renaming bound variables. As these variables are already "locked" in regards to an abstraction, any renaming of these with the abstraction to another variable name is an idempotent operation, and

can be done at any time. The need for this comes from the unintentional possible binding of a variable during a substitution step. A short example:

$$\begin{aligned} &\text{Let } F = \lambda xy.yx, \text{ then } \forall M, N : \\ &FMN \equiv NM \end{aligned} \tag{4}$$

This would follow from the inductive definition of substitution, however, when taking $M = y$ and $N = x$, this leads to the expression $xy \equiv xx$. This comes due to the fact that the substitution of N in M should not capture any free variables in N.

3.1.2 Reduction

A reduction in λ-calculus can be defined as a conversion between terms that contracts the term tree. In this sense, it can be seen as a simplification of the multiple abstractions and applications in a term to a more simple, albeit long, form. Once a term is at a point were it has no possible reduction possible, it is in what is called "normal form". It is simple to see the relation with the concept of reduction in TRS, and most of the terminology introduced in that section can be used in regards to λ.

Reduction Let **R** be a notion of reduction on Λ. Then **R** induces the binary relations:

$$\begin{aligned} &\to_R \textit{ one step R-reduction} \\ &\to_R^* \textit{ R-reduction} \\ &=_R \textit{ R-equality or R-convertibility} \end{aligned} \tag{5}$$

On this simple idea of reduction we can define the classical notion of reduction in the lambda calculus, β-reduction. When talking about measuring time complexity in the *lambda*-calculus, this is a good place to start, as it is the main computational device used. It is based on the substitution rule introduced earlier:

$$\beta : (\lambda x.M)N \to M[x := N] \tag{6}$$

An introduction to some terms associated with reduction; An *R-redex* is a term or subterm that is not in R-normal form. Now when talking about introducing a "computational cost",β-reduction is the reasonable choice, as it seems to provide a relation to transitions in a Turing machine. However, the problem in this case is with the arbitrary duplication of terms that can occur during a reduction. But before diving into that, a brief explanation of reduction strategies.

Reduction Strategies When talking about reduction strategies, we refer to a map $F : \Lambda \to \Lambda$ such that for all terms $M \in \Lambda : M \xrightarrow{*} F(M)$. That is, there exists one and only one term to which M reduces in this map. Strategies are defined as terminating if for every M with a normal form, for some $n \in \mathbb{N}$,$F^n(M)$ is in normal form, where F^n represents the n-th step in the reduction path of M.

The reduction strategy that [1] uses is named Leftmost Outermost. This is a normalizing strategy where if M is not in normal form, the leftmost-outermost redex is reduced until it is. Formally, we can define this in the syntax of contexts inductively, but first we introduce the notion of ordering on terms. As ppreviously stated, TRS have a notion of ordering that is based on strings, but for a better understanding of it, it could also be seen as a tree. The ordering introduced by Accatoli and Dal Lago in their paper is based on the idea of contexts, generalizing the notion and allowing for relative positioning of subterms without needing to understand or define the whole term. It is defined as follows [1]:

1. Outside-in order:
   (a) Root: $\langle\cdot\rangle \prec_O C$ for every context $C \neq \langle\cdot\rangle$.
   (b) Contextual closure: If $C \prec_O D$ then $E\langle C\rangle \prec_O E\langle D\rangle$ for any context E.
2. Left-to-right order:
   (a) Application: If $C \prec_p t$ and $D \prec_p u$ then $Cu \prec_L tD$.
   (b) Contextual Closure: If $C \prec_L D$ then $E\langle C\rangle \prec_L E\langle D\rangle$ for any context E.

Both of these orderings are partial orders in relation to terms. A total order is achieved by joining the two:

$$\text{If } C \prec_O D \text{ and } C \prec_L D \text{ then } C \prec_{LO} D.$$

As mentioned in the introduction, there is a family of terms in λ that in a linear number of steps, reduces to a term of exponential size. Consider the example provided by Accatoli and Dal Lago in [1]. It is introduced here to make the following sections easier to follow. We will also define the ordering of the subterms, and show that LO is a total order on it.

$$\begin{gathered}\text{Let } u = yxx, \text{ and consider the sequence of terms } t_n \\ \text{for } n \in \mathbb{N} \text{ be defined inductively as} \\ t_0 = u \ , \ t_{n+1} = (\lambda x.t_n)u \text{ for every } n \in \mathbb{N}.\end{gathered}$$

This term has size linear in n, but its normal form has 2 copies of the previous term, t_{n-1}. That is, the size of the normal form is exponential in the size of the initial term. As we continue, we will use a member of this family as an example, namely $t_2 \equiv (\lambda x.(yxx)(yxx))(yxx)$.

In order to define the ordering of this term, it must be first converted to a context based syntax:

$$C_0\langle C_1\langle \lambda x.C_3\langle C_4\langle yxx\rangle C_5\langle yxx\rangle\rangle C_2\langle yxx\rangle\rangle$$

An ordering on this term in LO would then be:

$$C_0 \prec_{LO} C_1 \prec_{LO} C_2 \prec_{LO} C_3 \prec_{LO} C_4 \prec_{LO} C_5 \tag{7}$$

by means of the contextual closure of LO.

3.1.3 Residuals

An important notion for understanding the LSC strategy is that of residuals. It can intuitively be understood as the remains of previously reduced redexes in a term after an arbitrary amount of reduction steps. In order to properly understand this concept, we will introduce the notion of "marking" redexes. In order to track the remains of a redex after a reduction, each redex will have an index associated with it:

$$v_0, v_1 \tag{8}$$

As an example, the previous term and a one step reduction from it:

$$(\lambda x.(yxx)(yxx))(yxx) \rightarrow \tag{9}$$

For further reading on the syntax and axioms of the lambda calculus, refer to [3].

3.2 LSC

In order to properly illustrate the properties of the LSC, we will use as an example the **size exploding** family that was introduced in 3.1. Any member of this family could be a good example, but for brevity, this paper will use the previously introduced $t_3 \equiv \lambda x.((\lambda x.(yxx))(yxx)(yxx))(yxx)$. As explained before, this term, under regular β-reduction, will evaluate to an exponential length in a number of steps linear to its size, the main obstacle in the invariance of λ. The linear substitution calculus is based in **explicit substitutions**, as oposed to the regular implicit substitutions that *beta*-reduction produces. The grammar is defined as follows:

$$t, u, r, p = x \mid \lambda x.t \mid tu \mid t[x \leftarrow u] \tag{10}$$

where $t[x \leftarrow u]$ is an explicit substitution. We need to introduce the idea of a new kind of contexts in order to properly define the reduction strategies for the proof of invariance. Besides the addition of explicit substitution to regular contexts, we introduce **shallow** and **substitution** contexts (S and L respectively):

$$\begin{aligned} S &= \langle\cdot\rangle \mid \lambda x.S \mid St \mid tS \mid S[x \leftarrow t] \\ L & \end{aligned} \tag{11}$$

4 Proof Overview

As stated before the measure employed to analyze the time invariance of lambda calculus is the number of transitions in a turing machine. By means of the Linear Substitution Calculus, it is possible to represent even size-exploding terms in Turing machines in polynomial time.

4.1 High level implementation systems

The purpose of the high level implementation system definition is to provide a rewriting system invariant to lambda calculus. This step is a bridge of sorts in between lambda calculus and turing machines. For this, we need to define this class of rewriting systems, and which properties should they satisfy in order to be invariant to lambda calculus. We want spcifically termintation and polynomial overhead.

$$\rightsquigarrow \text{ terminates iff} \rightsquigarrow_X \text{terminates}$$

Furthermore,

$$t \rightsquigarrow_X^k u \text{ iff } t \rightsquigarrow^h u \downarrow \text{ with } O(h) \in O(k^n) \text{ for some } n \in \mathbb{R}$$

4.1.1 Properties of high level systems

This section will provide a basis for the properties of high level systems, and define the properties that they contain.

4.2 Useful derivations

The use of

5 Comparing LSC terms

- Look into the algorithm to compare them and talk about it - If we can define an equality relation in the LSC, then we can prove basically the same as we can in Lambda calculus.

6 Conclusion

References

[1] Beniamino Accattoli and Ugo Dal Lago. "Beta Reduction is Invariant, Indeed (Long Version)". In: *CoRR* abs/1405.3311 (2014). arXiv: 1405.3311. URL: http://arxiv.org/abs/1405.3311.

[2] Beniamino Accattoli and Ugo Dal Lago. "On the Invariance of the Unitary Cost Model for Head Reduction (Long Version)". In: *CoRR* abs/1202.1641 (2012). arXiv: 1202.1641. URL: http://arxiv.org/abs/1202.1641.

[3] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984. ISBN: 9780444875082. URL: https://books.google.es/books?id=KbZFAAAAYAAJ.

[4] Peter van Emde Boas. "Machine models and simulations". In: *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. Cambridge, MA, USA: MIT Press, 1991, pp. 1–66. ISBN: 0444880712.

[5] S. C. Kleene and J. B. Rosser. “The Inconsistency of Certain Formal Logics”. In: *Annals of Mathematics* 36 (1935). URL: http://www.jstor.org/stable/1968646.

[6] Peter Selinger. *Lecture Notes on the Lambda Calculus*. 2001. DOI: https://doi.org/10.48550/arXiv.0804.3434.

[7] Michael Sipser. *Introduction to the theory of Computation*. 2013.

[8] Alan Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: (1936). DOI: https://doi.org/10.1112/plms/s2-42.1.230.