An Introduction to the Invariance Thesis in Lambda Calculus

Haileselassie Gaspar

Supervisors: Femke V. Raamsdonk, Jörg Endrullis

June 12, 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 section3.2. *lambda*-calculus can also be seen as a Term Rewriting System, another concept that will be introduced in detail in section3.1.

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 one looks for. 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 – encoding of turing machines in the lambda calculus –, it is proven that there is an implementation of Turing machines in the deterministic λ-calculus, with only a linear overhead in time. 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 a TRS 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 necesary 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?

3 Theoretical Background

3.1 Term Rewriting Systems

While term rewriting knowledge is not necessary to understand and use the λ-calculus, in order to talk more in depth about its qualities and properties, it will be helpful to have a small introduction into the topic. It is also necessary to understand the reason why some of the properties of λ extend to other TRS as the one used by Accatoli and Dal Lago in [1], the one used in this paper.
It is important to note the fact that λ and other TRS based on it belong to a more specific class of TRS, called Higher-Order Reduction Systems. This distiction is necessary from the fact that λ requires a binding in order to properly represent the idea of abstraction. However, most of the properties of first-order are also properties of higher-order, since higher-order term rewriting is a generalization of the former. Therefore, a brief introduction to the idea of term-rewriting systems as a whole can be helpful before diving into the more complex topic of higher-order rewriting. To talk about the idea of Term Rewriting Systems, we first need to introduce the idea of an Abstract Reduction System, a pair $(A, \rightarrow)$ of a set A and a *reduction*, that is, a directed relation $\rightarrow \subseteq A \times A$. Some notions about this idea of reduction are

introduced in [2]. For this paper, these are the essential ones:

$$\begin{aligned}
&\xrightarrow{0} := \ (x,x)|x \in A \textbf{ identity} \\
&\xrightarrow{i+1} := \xrightarrow{i} \circ \rightarrow \ (i+1)\textbf{-fold composition, } i \leq 0 \\
&\xrightarrow{+} := \bigcup_{i \leq 0} \xrightarrow{i} \textbf{ transitive closure} \\
&\xrightarrow{*} := \xrightarrow{+} \cup \xrightarrow{0} \textbf{ reflexive transitive closure} \\
&\xrightarrow{=} := \rightarrow \cup \xrightarrow{0} \textbf{ reflexive closure}
\end{aligned} \tag{1}$$

A path from an element $x \in A$ to $y \in A$ can be defined as $\sigma : x \xrightarrow{n} y$, for a path of lenght n from x to y. The length of said path can be represented as $|\sigma|$. For some finite path from x to y, $\sigma : x \xrightarrow{*} y$, and for a non-empty, finite path from x to y, $\sigma : x \xrightarrow{+} y$. Some terminology that will be essential when talking about not only the computational aspects of λ but the idea of reduction as a whole is the idea of *normal forms*.

$$x \text{ is in normal form iff } \forall y \in A, \nexists \sigma : x \rightarrow y \tag{2}$$

3.1.1 Universal Algebra

Before defining the idea of term rewriting, we need to introduce the idea of terms. This concept is very complex and wide, and again, this paper will only introduce the reader to some notions of the topic in the context of this paper, and the references should provide a more complete introduction to the topic [2]. Building on the idea of Abstract Rewriting Systems, what if not every member of the set A in the ARS $(A, \rightarrow)$ is a variable? Here is where the idea of terms comes in. Terms are built from both variables and a set of function symbols in order to represent more complex structures. To define the set of function symbols, one introduces *signatures*.

Signature A **signature** Σ is a set of **function symbols**, where each $f \in \Sigma$ is associated with a non-negative integer n defining the number of elements that f needs, called the **arity** of f. Every function symbol with arity 0 is called a **constant symbol**. Once introduced signatures, the definition of terms is inductively defined as:

- $X \subset T(\Sigma, X)$ for a set of variables X.
- for all $n \leq 0$, all $f \in \Sigma^{(n)}$, meaning the set of functions of arity n, and all $t_1, ..., t_n \in T(\Sigma, X)$, we have $f(t_1, ..., t_n) \in T(\Sigma, X)$.

Once defined the idea of terms, it is necessary to introduce the idea of changes to that term in order to have an abstract reduction system. A step that ensures that terms are not static. This is called a **substitution**. First, an introduction to the idea of positions in ARS will give a basis for the idea, and provide a suitable base to build upon when talking about positions in λ. The set of **positions** of a term s is defined

as a set $Pos(s)$ of strings over the alphabet of positive integers, defined inductively as:

$$\begin{aligned} s = x \in X &\rightarrow Pos(s) = \{\epsilon\} \\ s = f(s_1, ..., s_n) &\rightarrow Pos(s) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip | p \in Pos(s_i)\} \end{aligned} \tag{3}$$

When referring to ϵ we refer to it as the **root position** of the term s. It is also possible to define a partial order on positions, the **prefix order**, defined as

$$p \leq q \text{ iff there exists } p' \text{ such that } pp' = q$$

A later definition of this partial order will be presented based on contexts. The idea of positions of terms and subterms is essential in the discussion on invariance, since the largest problems of cost models for λ come from its ability to arbitrarily move and duplicate terms, and so this concepts will be essential during the discussion of the proof. A substitution is a function on a set of terms $T(\Sigma, V)$ defined as $\sigma : V \rightarrow T(\Sigma, V)$ such that $\sigma(x) \neq x$ for only finitely many x. In other words, it is a mapping from variables to terms that includes at least one non-identity conversion on the set of variables in V. This idea can be inductively extended to $\hat{\sigma} : T(\Sigma, V) \rightarrow T(\Sigma, V)$ trivially, by propagating the operation to the subterms of a term. The main basis of TRS is the idea of simplifying terms. Now that the terminology is laid out, it is possible to introduce the idea of $\Sigma - identities$ or equations. These are binary relations on the set of terms, written as $s \approx t$, which somewhat lay out the rules of a system. Based on this, a **reduction relation** $\rightarrow_E$ can be seen as a one-direction conversion from term s to term t.

3.1.2 Term Rewriting Systems

Once all these concepts are defined, we can define TRS as a set of rewrite rules $l \approx r$ so that l is not a variable and the set of variables of r is a subset of the set of variables of l. If the reader is familiar with the λ calculus, this may seem as a sufficient definition of the concept of TRS to catalogue λ as one, but the main flaw with this idea is in the concept of abstraction. Classical or First-Order TRS do not allow the idea of nesting or ordering function symbols arguments, or the idea of binding, a crucial aspect of not only λ, but most mathematical systems. It is therefore necessary to go further to completely define the group of systems in which λ is. However, that is outside the scope of this paper. Most of the ideas of First-Order TRS provide an understanding of the functioning of these types of systems, and since the framework that the invariance proof is essentially selfcontained, the previous concepts should serve as a brief introduction to the subject and as an aid to understand most of the ideas that Accatoli and Dal Lago use in its paper [1].

3.2 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:

$$\begin{aligned} &\text{Variables: } x \in \Lambda \\ &\text{Abstraction: } M \in \Lambda \implies (\lambda x.M) \in \Lambda \\ &\text{Application: } M, N \in \Lambda \implies (MN) \in \Lambda \end{aligned} \tag{4}$$

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:

$$C ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \tag{5}$$

Substitution This topic will be expanded on section 3.2.2, 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:

$$\begin{aligned} &x\{x := N\} \equiv N \\ &y\{x := N\} \equiv y \text{ iff } x \neq y \\ &(\lambda y.M_1)\{x := N\} \equiv \lambda y.(M_1\{x := N\}) \\ &(M_1M_2)\{x := N\} \equiv (M_1\{x := N\})(M_2\{x := N\}) \end{aligned} \tag{6}$$

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].

3.2.1 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:

$$\begin{aligned} &\text{Let } F = \lambda xy.yx, \text{ then } \forall M, N : \\ &FMN \equiv NM \end{aligned} \tag{7}$$

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.

3.2.2 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:

$$\begin{aligned} &\to_R \textit{ one step R-reduction} \\ &\to_R^* \textit{ R-reduction} \\ &=_R \textit{ R-equality or R-convertibility} \end{aligned} \tag{8}$$

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:

$$\beta : (\lambda x.M)N \to M[x := N] \tag{9}$$

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 $n \in F^n(M)$ 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:

(10)

Previously, we introduced the notion of positions in First-Order TRS.

3.2.3 Church Rosser And Standarization

- Diamond property - Is LO CR? - What does it mean to be standard? - Is LO Standard? For further reading on the syntax and axioms of the lambda calculus, refer to [3].

3.3 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 3.2. Any member of this family could be a good example, but for brevity, this paper will use $t_2 \to (\lambda x.(yxx)(yxx))(yxx)$. 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:

$$t, u, r, p = x \mid \lambda x.t \mid tu \mid t[x \leftarrow u] \tag{11}$$

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):

$$S = \langle\cdot\rangle \mid \lambda x.S \mid \tag{12}$$

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.

4.1.2 Proof of high level properties

- Termination and polynomial overhead for a generic LSC term

4.2 Useful derivations

The use of

4.3 Standarization of Useful derivations

- Why do LSC srategies contain the subterm property? - Why does LOU have the subterm property?

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] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[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.