Cost Models in the Lambda Calculus, and its impact in the Invariance thesis

Haileselassie Gaspar

Supervisors: Femke V. Raamsdonk, Jörg Endrullis

June 7, 2025

1 Abstract

The idea of a cost model for the *lambda*-calculus has

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. In order to analyze the 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

– equivalence part ; turing equivalence, turing completeness ; 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 *lambda*-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 *lambda*-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 Term Rewriting

Systems (TRS for short) are Turing Complete, and therefore, seeing as *lambda* is a type of TRS, it follows that it is also Turing Complete. It was later proven by Rosser and Kleene that it 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 *lambda*-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 *lambda* extend to other TRS as the one used by Accatoli and Dal Lago in [1], the one used in this paper.
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, $\sigma : x \xrightarrow{+} y$. == UNFINISHED ==

3.2 Introduction to lambda calculus

The idea of *lambda*-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 *lambda*-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 *lambda*-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:

$$\begin{aligned}
&\text{Variables: } x \in \Lambda \\
&\text{Abstraction: } M \in \Lambda \implies (\lambda x.M) \in \Lambda \\
&\text{Application: } M, a
\end{aligned} \tag{2}$$

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:

$$C ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \tag{3}$$

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:

$$\begin{aligned}
&x[x := N] \equiv N \\
&y[x = N] \equiv y \text{ iff } x \neq y \\
&(\lambda y.M_1)[x := N] \equiv \lambda y.(M_1[x := N]) \\
&(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])
\end{aligned} \tag{4}$$

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 *alpha* 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 *lambda*-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:

$$\begin{aligned}
&\text{Let } F = \lambda xy.yx, \text{ then } \forall M, N : \\
&FMN \equiv NM
\end{aligned} \tag{5}$$

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 *lambda*-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".

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

$$\begin{aligned} &\rightarrow_R \textit{ one step R-reduction} \\ &\rightarrow_R^* \textit{ R-reduction} \\ &=_R \textit{ R-equality or R-convertibility} \end{aligned} \tag{6}$$

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:

$$\beta : (\lambda x.M)N \rightarrow M[x := N] \tag{7}$$

A brief 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 - LO order on reductions

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

- Pretty obvious

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. It will be shown that by converting the LSC to a NPDA, the lambda calculus is indeed quadraticly bound in time when representing Turing machines.

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,

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

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 Low level implementation

A high level implementation system is implemented on a turing machine with an overhead in time polynomial to k and the size of the initial term.

4.2.1 Properties of low level systems

- Subterm - Selection

4.2.2 Proof of low level propeties

- Polynomial bound on reductions in LSC strategy.

4.3 Useful derivations

- What does it mean for a derivation to be useful? - Why do we need useful derivations? - Leftmost Outermost Useful

4.4 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.

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`.