Vrije Universiteit Amsterdam

Bachelor Thesis

Introduction to the Weak Invariance Thesis in λ-Calculus

Author: Haileselassie Gaspar (2762335)

1st supervisor: Femke van Raamsdonk
2nd reader: Jörg Endrullis

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

October 25, 2025

Abstract

The idea of computational Invariance is one that is essential to complexity theory, ever since Van Emde Boas proposed it [4]. While this paper relates to many areas of computability theory, we will focus on the invariance of λ-calculus. Accattoli and Dal Lago proposed, using a calculus with explicit substitutions called the **Linear Substitution Calculus**, a reduction strategy that relates polynomially to Turing Machines when only time complexity is concerned. In this paper we will introduce said calculus and strategy, and provide an example of how they relate to the classical untyped λ-calculus .

1 Introduction

During the start of the 20th century, the idea of computability began to be a main mathematical problem. David Hilbert posed the following 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.* Intuitively, computability is a property of problems or functions that can be solved by some mechanical process. Formally, the set of computable functions can be defined as every function $f : \mathbb{N}^k \to \mathbb{N}$ for which there exists an effective procedure to compute $f(x)$, where $\mathbb{N}^k$ represents a tuple of natural numbers. By effective procedure we refer to a deterministic collection of operations that can be performed to a collection of symbolic inputs in order to produce a corresponding symbolic output– cite theory of recursive functions and effective computability –.
Both Alonzo Church and Alan Turing proved almost simultaneously that there exists a model defining the group of computable functions, the former through general recursive functions and the latter with the conceptual "Turing" machines. Besides the computability of said functions, both models provide a method of studying the effective procedure for any function that is computable.

Turing Machines In 1936, Alan Turing proposed a model now referred to 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 from said tape, one symbol at a time. This is the main computational model most used when talking about complexity of algorithms [7].

While the details of its logical implementation are not important in this paper, we will note that the reason they are very useful when analyzing the complexity of algorithms is due to the fact that they posess a very clear and stable cost model. For every step in an algorithm implemented in a Turing Machine, there is a constant cost in both terms of space, the use of a single cell of the tape, and time, a single movement of the head that reads and writes symbols on the tape.

Lambda calculus The λ-calculus was conceived as a foundational system for mathematics and logic in the 1930s by Alonzo Church, although this initial calculus 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 Section 2.

There is an encoding from the natural numbers to both λ-terms and Turing Machines. Therefore, as every function $f : \mathbb{N}^k \to \mathbb{N}$ can be represented in both models, they are computationally equivalent. This came to be known as the Church-Turing thesis.

1.1 Invariance

Due to the design of Turing Machines, they serve as an accurate and measurable way of understanding the space and time that computations need in order to produce an output. While they represent the same functions as λ-calculus does (computable functions), the use of a single unitary cost model for a computational step in time, with the number of head movements, and in space, with the amount of cells used, provide a clearer view of the complexity of an algorithm, which is why they are the most used method of measuring complexity classes nowadays.
Since both models represent the same functions, it follows that there must be a way to represent the procedures in one model in the other and vice-versa. Here is where the idea of invariance comes in. Invariance of models, in the words of Van Embde Boas [4], stipulates the following:

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

What this statement 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 compute a function $f : \mathbb{N}^k \to \mathbb{N}$ in λ-calculus than in a Turing Machine and vice-versa, so every algorithm that could theoretically run in a Turing Machine could be replicated in λ-calculus. This paper will use the *weak* invariance thesis, meaning the space requirements, a long-standing problem with lambda calculus, will be dropped. The statement then follows the following form:

Reasonable machines can simulate each other within a polynomial bound 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 the complete un-typed λ-calculus regarding Turing machines.
This subject has a large body of literature, and we only consider a small fragment of papers. For a more detailed introduction to the topics mentioned, refer to the references.

2 Introduction to the λ-calculus

2.1 Mathematical Preliminaries

A term is a mathematical object built recursively from constants and functions. In this paper, we will focus on λ terms and their extension into the LSC. A tuple is an unordered group of elements of an arbitrary size. It is represented as $t = (x_1, x_2, ..., x_n)$.
Given a set A, a subset $A \times A$ is called a relation. That is, a set consisting of ordered pairs (a_n, a_m) so that $a_n, a_m \in A$. Note that a_n and a_m can be equal. It is usually denoted by $a_n \to a_m \in R :\to$. A relation is a reduction if it has the reflexive and transitive properties, and in that case, $a_n \to a_m$ is called a reduction step. The reflexive and transitive closures

of reductions are denoted as $\overset{0}{\to}$ and $\overset{i}{\to}$ for some $i > 0$ respectively. It is important to note that reductions do not have symmetric properties. For every $(a, b) \in R$, given that $a \neq b$, there is no ordered pair $(b, a) \in R$.
An element $a \in A$ with a reduction relation R defined on A is said to be in normal form if $\nexists b \in A$ s.t $(a, b) \in R$. A finite or infinite sequence of reduction steps is called a reduction sequence. A term is normalizing in R if it can reach a normal form after a finite reduction sequence. If there is a unique reduction sequence from said term to its normal form, then we say the term is **strongly normalizing**(SN). If not, we say it is **weakly normalizing**(WN).
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 behavior of a function outside just what it produces [6].

The λ-calculus uses the intensional view of functions to treat them as expressions, and analyze their behavior and, more importantly, their computability.
In order to talk about the invariance of λ-calculus it is first necessary to define some notation that will be used in this paper.

2.2 λ-Terms

Terms Assuming a countably infinite set $\mathcal{VAR} = \{x, y, z, \cdots\}$ of variables, let $M, N, P....$ denote arbitrary λ-terms. The set of λ-terms Λ is inductively defined as:

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

A context is a λ-term with one (or more) holes in it. In this paper we will only utilize the single-hole contexts.

$$C ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC$$

The plugging of a term in a context is represented as $C\langle t\rangle$. We also introduce a relation $\prec_p$, called the **prefix** relation, between contexts and terms. It is defined as follows

$$u, t \in \Lambda; C\langle u\rangle = t \implies C \prec_t u$$

2.2.1 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. Revisiting the previous example:

$$\begin{aligned}&\text{Let } F = \lambda xy.yxz, \text{ then } \forall M, N :\\&FMN = NMz\end{aligned} \tag{1}$$

This would follow from the inductive definition of substitution, however, when taking $M = y$ and $N = x$, substituting these terms into the expression leads to the equality $xyz = xxz$, an obvious contradiction. This comes due to the fact that the substitution of N in M should not capture any free variables in N.

Substitution This topic will be expanded on Section 2.2.2, but a basic understanding is provided here. The result of substituting N for the free occurrences of x in M (notation $M\{x := N\}$) is inductively defined as:

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

A free variable is any variable not bound by an abstraction. For example:

$$(\lambda xy.yxz)$$

In this term, y and z are not bound, therefore free, while x is bound by the abstraction λx. When replacing terms it is important to take care not to accidentally bind a free variable by mistake. That is why when replacing terms, if any free variables would become bound by the replacement, they must be renamed. This is called α-conversion. 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].

2.2.2 β-Reduction

β-reduction is a relation $\Lambda \times \Lambda$ based on the substitution rule introduced earlier:

$$\beta : (\lambda x.M)N \to M\{x := N\} \tag{2}$$

A *β-redex* is a term or subterm that is of the form $(\lambda x.M)N$, that is, a subterm that accepts aβ-reduction step. 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, this relation is not exact, and therefore, we cannot use β-reduction as a cost model for λ-calculus. An example, using the previous term:

$$(\lambda xy.yxz)MN \overset{1}{\to}_\beta (\lambda y.yMz)N \overset{1}{\to}_\beta NMz$$

2.3 Leftmost Outermost

When talking about reduction strategies, we refer to a map $F : \Lambda \to \Lambda$ such that for all terms $M \in \Lambda : M \overset{*}{\to} F(M)$. 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 $M \overset{n}{\to} F^n(M)$.
The reduction strategy that [1] uses is named Leftmost Outermost. This is a normalizing strategy where if M is not in normal form, and there exists such a 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 previously 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 Accattoli 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:

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

This means that every term $t \in \Lambda$ has at most one leftmost-outermost redex, and therefore every reduction sequence using this strategy is deterministic. The Leftmost Outermost strategy therefore reduces the leftmost outermost redex always, and it is a strongly normalizing strategy. An example:

$$(\lambda x.(\lambda y.yx)z)(yy(\lambda x.x)y) \xrightarrow[LO]{1} (\lambda y.y(yy(\lambda x.x)y))z$$

Let us introduce a more conceptual view of what these orderings represent: Imagine an infinite term t_∞. Imagine that this term contains every possible λ-term as a subterm. Viewing the term as a tree, at the root of the term, there will always be a single-hole context, $\langle\cdot\rangle$, in which the term could be plugged: $\langle t_\infty\rangle = t_\infty$. If we go a level deeper, we will encounter an application of 2 contexts, lets call them C_1 and C_2 (2 because the application of contexts is a binary relationship that can be extended to multiple terms by association). These two contexts will in turn have two contexts inside and so on and so forth. Since the term t_∞ has every subterm in it, we will have infinite contexts containing contexts before reaching an abstraction, in order to consider every infinite sized term, but we know that at the end of this nesting of contexts, we will most certainly encounter every possible variable, since there can be no applications or abstractions without variables. Thus, every variable $x \in \Lambda$ and every context surrounding a variable C_x, are at the end of the outside-in order with respect to **every other context**. In other words: $D \prec_O C_x, \forall D \in \mathcal{C}$ where $\mathcal{C}$ represents the set of all contexts.
Now, it is clear that for every context that is not surrounding a variable, except for the empty context $\langle\cdot\rangle$, there exists a context both above and below, so: $\forall C \in \mathcal{C}, \exists D_1, D_2 \ s.t \ D_1 \prec_O C \prec_O D_2$.
This context however, has no relation to contexts that either do not contain it, or are not contained in it. In order to relate contexts with no ordering based on $\prec_O$, we can use the (relative) position of contexts inside a common ancestor. Lets say at one point of t_∞ there is a context C_0 with 2 contexts inside of it, C_1, C_2. Lets also say that the application of

these 2 contexts is in the form $C_0 = C_1C_2$. We can then say that $C_1 \prec_L C_2$, or that C_1 is 'to the left' of C_2. Since every context except contexts surrounding variables is composed of exactly 2 contexts, this creates a relation between contexts which are not contained in one another.
Intuitively, it is easy to see how we have just defined the only two possible orderings on a binary tree, and so with both these orderings, there is no 2 terms that are on the same ouside-in position and left-to-right position.

2.4 Size-exploding family

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. While the invariance result of Accattoli and Dal Lago does not concern itself with space invariance, it is important to note that due to the design of Turing Machines, they have a conceptually different relation between space and time complexity as λ does. While the former has the space complexity be bound by the time complexity, that is, the number of used cells on the tape must be equal or less to the amount of times the head moves, the latter has the opposite. In λ-calculus, time complexity, or the amount of β-reduction steps, is bound by the size of the inital and final size. That means that to provide an accurate invariance result, we not only need to fix the inconsistencies with the number of reduction steps in λ, but we must also take care to show the normal form of a term in a way that is at least polynomially related to the size of the initial term, and the number of steps taken, closing the gap between the two ideas.
Consider the example provided by Accattoli 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.

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

Lemma 2.1. *Given a term of the form $(\lambda x.t_n)a$ for some $a \in \Lambda$:*

$$(\lambda x.t_n)a \xrightarrow[LO\beta]{1} (\lambda x.t_{n-1})yaa \tag{3}$$

Proof.

$$(\lambda x.t_n)a = (\lambda x.(\lambda x.t_{n-1})u)a \xrightarrow[LO\beta]{1} (\lambda x.t_{n-1})yxx\{x \leftarrow a\} = (\lambda x.t_{n-1})yaa \tag{4}$$

□

Lemma 2.2. *Given a term of the form $t_{n+1} = (\lambda x.t_n)u$:*

1. *Its β-normal form will be $r_{n+1} = yr_nr_n$*
2. *It will reach β-normal form in exactly $n+1$ steps.*

Proof. 1. By induction:

$$t_1 = (\lambda x.t_0)u \xrightarrow[LO\beta]{1} yuu = yr_0r_0 = r_1$$

$$t_{n+1} = (\lambda x.t_n)u = (\lambda x.(\lambda x.t_{n-1})u)u \xrightarrow[LO\beta]{1} (\lambda x.t_{n_1})(yuu) = \tag{5}$$

$$(\lambda x.t_{n-1})r1 \xrightarrow[LO\beta]{*} r_{n+1} = y(r_nr_n)$$

By Lemma 2.1, any term of the form $(\lambda x.t_n)a$ reduces in one step to $(\lambda x.t_{n-1})yaa$,and so any term $(\lambda x.t_n)u$ will evaluate to $(\lambda x.t_{n-1})r_1$, therefore, it is clear that the normal form of t_{n+1} is $r_{n+1} = y(r_n r_n)$

2. By induction: Assume that the number of steps from t_n to r_n,k for $\rho : t_n \xrightarrow[LO\beta]{k} r_n$, is $n-1$. Then, after $n-1$ LO β steps, we reach r_n, but by Lemma 2.2.1, and Lemma 2.1 we can see that the form of t_n after $n-1$ steps will be $(\lambda x.u)r_{n-1} \neq r_n$ Now assume that $k = n+1$. Then after n steps there should be another LO β step, from r_n, but by definition of the normal form, this is impossible.

□

Theorem 2.3. *Every λ-term $t \in \Lambda_{Exp}$ reduces in n LO β-reduction steps to a term r_n, with $|r_n| \in O(2^n)$.*

Proof. It follows from Lemma 2.1 and 2.2. □

As an example for the rest of the paper, we will use the term t_2 from this family. In order to define the ordering of this term, it must be first converted to a context based syntax:

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

An ordering on this term in LO would then be:

$$C_0 \prec_{LO} C_1 \prec_{LO} C_2 \prec_{LO} C_3 \prec_{LO} C_4 \prec_{LO} C_5 \tag{6}$$

by means of the contextual closure of LO.

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

3 Linear Substitution Calculus

In order to properly illustrate the properties of the LSC, we will use as an example the **size exploding** family that was introduced in 2. Any member of this family could be a good example, but for brevity, this paper will use the previously introduced

$$t_3 \equiv \lambda x.((\lambda x.(yxx))(yxx)(yxx))(yxx) \tag{7}$$

. 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:

$$t, u, r, p = x \mid \lambda x.t \mid tu \mid t[x \leftarrow u] \tag{8}$$

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):

$$\begin{aligned} S &= \langle\cdot\rangle \mid \lambda x.S \mid St \mid tS \mid S[x \leftarrow t] \\ L &= \langle\cdot\rangle \mid L[x \leftarrow t] \end{aligned} \tag{9}$$

The unfolding operation $t \downarrow$ is defined as:

$$\begin{aligned} & t \downarrow = t \\ & (\lambda x.t) \downarrow = \lambda x.t \downarrow \\ & (tu) \downarrow = t \downarrow u \downarrow \\ & t[x \leftarrow u] \downarrow = t \downarrow \{x \leftarrow u \downarrow\} \end{aligned} \tag{10}$$

and the contextual unfolding $t \downarrow_S$ as:

$$\begin{aligned} & t \downarrow_{\langle\cdot\rangle} = t \downarrow \\ & t \downarrow_{\lambda x.S} = t \downarrow_S \\ & t \downarrow_{Su} = t \downarrow_{uS} = t \downarrow_S \\ & t \downarrow_{S[x\leftarrow u]} = t \downarrow_S \{x \leftarrow u \downarrow\} \end{aligned} \tag{11}$$

Here are two examples to better illustrate the behavior:

$$\begin{aligned} & \lambda x.xyz[y \leftarrow u] \downarrow = \lambda x.(xuz)[y \leftarrow u] \\ & \lambda y.yx[x \leftarrow y][y \leftarrow z] \downarrow = \lambda y.yz[x \leftarrow y'][y' \leftarrow z] \end{aligned} \tag{12}$$

Notice the variable renaming on the second example in order to avoid variable capture. We introduce here the operational semantics of the LSC. There are two elementary reductions, required to provide the explicit substitutions up to an equivalence with the untyped λ-calculus.

$$\begin{aligned} & L\langle \lambda x.t\rangle u \rightarrow_{dB} L\langle t[x \leftarrow u]\rangle \\ & S\langle x\rangle[x \leftarrow u] \rightarrow_{ls} S\langle u\rangle[x \leftarrow u] \end{aligned} \tag{13}$$

It is important to note that we work modulo α-conversion on the dB steps, and that the linear substitution step is assumed not to capture variables. Based on this, [1] introduces a new reduction strategy, called **Leftmost-Outermost Useful** or **LOU** which we will use to show that the size explosion family does indeed not grow exponentially in the LSC under this strategy. It is important to note that while the Size-exploding family does not grow exponentially under the Linear Head Reduction of the LSC [2], it would never reach normal form if we included an application before the initial term, and so, an example of the use of LOU on it will show that even when completely reducing the term, the length increase will have a polynomial relation to the size of the initial term.

3.1 Leftmost-Outermost Useful

The definition of *usefulness* in [1] of a substitution step is defined in base to wether a redex is *useful* in its unfolding of the surrounding (shallow) context.

Useful Step A reduction step is useful if it is either a dB-step, or a ls-step $S\langle x\rangle \rightarrow_{ls} S\langle r\rangle$ such that the contextual unfolding $r \downarrow_S$:

1. Contains a β-redex,
2. Or is an abstraction and S is an applicative context, that is, a context of the form $A ::= S\langle Lt\rangle$. EXAMPLES

We now extend our definition of the complete LO order on redexes to the LSC by introducing a new rule:

$$\text{Substitution: If } C \prec_p t \text{ and } D \prec_p \text{ then } C[x \leftarrow u] \prec_L t[x \leftarrow D].$$

Therefore, we can define the **Leftmost Outermost Useful redex**:

A redex R of a term t is considered the leftmost outermost useful redex of t if $R \prec_{LO} Q$ for every other useful redex Q of t. We write $t \to_{LOU} u$ for the step reducing the leftmost outermost useful redex of t.

3.2 Size exploding family

In these examples we will forego the definition of the usual LO order on contexts and will focus solely on usefulness for the sake of convenience. Using the term t_2 as a first example, the first step is to split the term into (shallow) contexts, and order them according to the LO total order. In this term there is only one outermost context and it is a dB redex, which is inherently useful. Therefore, the first step will be:

$$(\lambda x.(\lambda x.yxx)(yxx))(yxx) \to_{dB} (\lambda x.yxx)(yxx)[x \leftarrow yx'x']$$

Note the α-conversion to avoid unwanted variable capture. We can then perform another dB step, arriving to the term

$$(yxx)[x \leftarrow yx'x'][x' \leftarrow yx''x'']$$

At this point, the polynomial algorithm introduced in [1] to find useful ls steps is necessary, although in this paper we will compute them by hand. There is obviously no possible β-redex of the unfolding of this term, and since there is no abstraction, we can safely say that this term is it dB-normal form and ls-normal form.

It is important to note that due to this example only requiring head substitution, the Linear Head Reduction introduced in [2] suffices, but the LOU strategy is necessary for reaching normal form in every case.

4 Generalizing Invariance

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. The LOU strategy is one strategy that has the properties required for invariance, but that does not mean that it is the only one. Accattoli and Dal Lago in [1] showed that any strategy having the following properties could be an adequate candidate for time invariance.

4.1 High level implementation systems

The purpose of the high level implementation system definition is to provide a rewriting system invariant to λ-calculus. This step is a bridge of sorts in between λ-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 λ-calculus. We want specifically termintation and polynomial overhead.

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

Furthermore,

$$t \rightsquigarrow_X^k u \text{ iff } t \rightsquigarrow^h u \downarrow \text{ with } O(h) \in O(k^2) \text{ for some } n \in \mathbb{N} \text{ and } n \leq 0$$

While there exists a proof for both of these statements in [1], we will not delve into it. We will instead, show that the implementation of addition and multiplication in the LSC follow these bounds.

5 Examples

There is an encoding from the natural numbers to the untyped λ-calculus, developed by Alonzo Church, called Church Numerals. We will denote church numerals using $\ulcorner n \urcorner$:

$$\begin{aligned} \ulcorner 0 \urcorner &= \lambda fx.x \\ \ulcorner n \urcorner &= \lambda fx.f^n(x) = \lambda fx.f_0(f_1(f_2...(f_n(x)...)) \end{aligned} \tag{14}$$

5.1 Addition

Addition using church numerals is defined by the term $add = \lambda mnfx.m(nf)x$, which applied to 2 other numerals will result the sum of them. An example: – todo –

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". In: (1936). DOI: `https://doi.org/10.1112/plms/s2-42.1.230`.