Beta Reduction is Invariant, Indeed

Beniamino Accattoli[*] Ugo Dal Lago[†]

Abstract

Slot and van Emde Boas' invariance thesis states that *reasonable* machines can simulate each other within a polynomially overhead in time. Is λ-calculus a reasonable machine? Is there a way to measure the computational complexity of a λ-term? This paper presents the first complete positive answer to this long-standing problem. Moreover, our answer is completely machine-independent and based over a standard notion in the theory of λ-calculus: the invariant cost model is the length of a leftmost-outermost derivation to normal form.

1 Introduction

When doing complexity analysis, one expects a neat relationship between primitives of the model and the way in which they are effectively implemented. In this respect, random access machines are often taken as the reference model, since their definition closely reflects the von Neumann architecture. The specifications of algorithms unfortunately lies on the other end of the spectrum, as one would like to be as machine-independent as possible. In this case the typical model is provided by programming languages. Functional programming languages, thanks to their higher-order nature, provide very concise and abstract specification models. Their strength is of course also their drawback: the abstraction from physical machines is pushed to a level where it is no longer clear how to measure the complexity of an algorithm. Is there a way in which such a tension can be solved?

The tools for stating the question formally are provided by complexity theory and by Slot and van Emde Boas invariance thesis [9]. Two computational models are *invariant* if they can simulate each other so as to make complexity classes independent from the model on top of which they are defined. Ideally, one would like only a linear overhead in the simulation. This is usually a too strong requirement, so that a polynomial invariance — that preserves the (non-)polynomial character of algorithms and computational problems — is instead considered. So a first refinement of our question is: are functional languages polynomially invariant with respect to standard models like random access machines or Turing machines? Such an invariance has to be proved via an appropriate measure of time complexity for programs, *i.e.* a *cost model.*

The natural answer is to take the *unitary* cost model, *i.e.* the number of evaluation steps. However, this is not well-defined. The evaluation of functional programs, indeed, depends very much on the evaluation strategy chosen to implement the language, as the λ-calculus — the reference model for functional languages — is so machine-independent that it does not even come with a deterministic evaluation strategy. And which strategy, if any, gives us the most natural, or *canonical* cost model (whatever that means)? These questions have received some attention in the last decades. The number of optimal parallel β-steps (in the sense of Lévy [8]) to normal form has been shown *not* to be a reasonable cost model: there exists a family of terms which reduces in a polynomial number of parallel β-steps, but whose complexity is non-elementary [7, 3]. If one considers the number of *sequential* β-steps (in a given strategy, for a given notion of reduction), the literature contains some partial positive results, all relying on the use of sharing (see below for more details). Some quite general results [6, 4] have been obtained through graph rewriting, itself a form of sharing, when only first order symbols are considered.

[*]Università di Bologna, `beniamino.accattoli@gmail.com`

[†]Università di Bologna & INRIA, `ugo.dallago@unibo.it`

Sharing is indeed a key ingredient, for one of the issues here is due to the *representation of terms*. But is appropriately managed sharing enough? The literature offers some positive, but partial, answers to this question. The number of steps is indeed known to be an invariant cost model for weak reduction [5, 6] and for head reduction [2].

If the problem at hand consists in computing the *normal form* of an arbitrary λ-term, however, no positive answer is known. We believe that it is embarrassing for the λ-calculus community not knowing whether the λ-calculus in its full generality is a reasonable machine. In addition, this problem is relevant in practice: proof assistants often need to check whether two terms are convertible, itself a problem that can be reduced to the one under consideration.

In this work, we give a positive answer to the question above, by showing that leftmost-outermost reduction *to normal form* indeed induces an invariant cost-model.

As in our previous work [2], we prove our result by means of the *linear substitution calculus*, a simple calculus of explicit substitutions based on a new *at a distance* approach, arising from linear logic and graphical syntaxes. Such a framework allows an easy management of sharing and, in contrast to previous approaches to explicit substitution, it admits a theory of standardization [1]. Indeed, the proof passes through a fine quantitative study of the relationship between standard reductions for the λ-calculus and some special standard reductions for the linear substitution calculus. Roughly, the latter avoids the size explosion problem while keeping a polynomial relationship with the former.

2 Why is The Problem Hard?

In principle, one may wonder why sharing is needed at all, or whether a relatively simple form of sharing suffices. In this section, we will show why this is *not* the case.

If we stick to explicit representations of terms, in which sharing is not allowed, couterexamples to invariance can be designed in a fairly easy way. Consider the sequence of λ-terms defined as follows, by induction on a natural number n (where u is the lambda term yxx): $t_0 = u$ and for every $n \in \mathbb{N}$, $t_{n+1} = (\lambda x.t_n)u$. The term t_n has size linear in n, and t_n rewrites to its normal form r_n in exactly n steps, following the leftmost-outermost reduction order; as an example:

$$\begin{aligned}
&t_0 \equiv u \equiv r_0;\\
&t_1 \to yuu \equiv yr_0r_0 \equiv r_1;\\
&t_2 \to (\lambda x.t_0)(yuu) \equiv (\lambda x.u)r_1 \to yr_1r_1 \equiv r_2.
\end{aligned}$$

For every n, however, r_{n+1} contains two copies of r_n, hence the size of r_n is *exponential* in n. As a consequence, the unitary cost model *is not* invariant: in a linear number of β-steps we reach an object which cannot even be written down in polynomial time.

The solution the authors proposed in [2] is based on explicit substitutions, and allows to tame the size explosion problem in a satisfactory way when *head*-reduction suffices. In particular, the head steps above become the following *linear* head steps:

$$\begin{aligned}
&t_0 \equiv u \equiv p_0;\\
&t_1 \to (yxx)[x/u] \equiv u[x/u] \equiv p_1;\\
&t_2 \to ((\lambda x.t_0)u)[x/u] \equiv ((\lambda x.u)u)[x/u] \to u[x/u][x/u] \equiv p_2.
\end{aligned}$$

As one can easily verify, the size of p_n is linear in n. More generally, linear head reduction has the *subterm property*, *i.e.* it only duplicates subterms of the initial term. This fact implies that the size of the result and the length of the derivation are linearly related. In other words, the size explosion problem has been solved. Of course one needs to show that 1) the compact results *unfold* to the expected result (that may be exponentially bigger), and 2) that compact representations can be managed efficiently (typically they can be tested for equality in time polynomial in the size of the compact representation), see [2] for more details.

It may seem that one is then forced to use ES to measure complexity. In [2] we also showed that LHR is at most quadratically longer than head reduction, so that the polynomial invariance of LHR lifts to head reduction. This is how we exploit sharing to circumvent the size explosion problem: we are allowed to take the length of the head derivation as a cost model, even if it suffers of the size explosion problem, because the actual implementation is meant to be via LHR.

There is a natural candidate for extending the approach to reduction to *normal form*: just iterate the (linear) head strategy on the arguments, obtaining the (linear) LO strategy, that does compute normal forms [1]. As we will show, for linear LO derivations the subterm property holds. The size of the output is still under control, being linearly related to the length of the LO derivation. Unfortunately, when computing normal forms this is not enough.

One of the key points in our previous work was that there is a notion of *linear head normal* form that is a compact representation for head normal forms. The generalisation of such an approach to normal forms has to face a fundamental problem: what is a *linear normal* form? Indeed, terms with and without ES share the same notion of normal form. Consider again the family of terms $\{t_n\}_{n\in\mathbb{N}}$: if we go on and unfold all ES in p_n, we end up in r_n. Thus, by the subterm property, the linear LO strategy takes an exponential number of steps, and so it cannot be polynomially invariant with respect to the LO strategy.

Summing up, we need a strategy that 1) implements the LO strategy, 2) has the subterm property and 3) never performs *useless* substitution steps, *i.e.* those steps whose role is simply to explicit the normal form, without contributing in any way to β-redexes. The main contribution of this work is the definition of such a linear useful strategy, and the proof that it is indeed polynomially invariant with respect to both the LO strategy and a concrete implementation model.

This is not a trivial task, actually. One may think that it is enough to evaluate a term t in a LO way, stopping as soon as the unfolding $u\!\downarrow$ of the current term u — *i.e.* the term obtained by expanding the ES of u — is a β-normal form. Unfortunately, this simple approach does not work, because the exponential blow-up may be caused by ESs *lying between* two β-redexes, so that proceeding in a LO way would execute the problematic ES anway.

Our notion of *useful* step will elaborate on this idea, by calculating *partial* unfoldings, to check if a substitution step contributes or will contribute to some β-redex. Of course, we will have to show that such tests do not cause another exponential blow up and that the notion of LO *useful* reduction retains all the good properties of LO reduction.

3 The Calculus

We assume familiarity with the λ-calculus. The language of the *linear substitution calculus* (LSC for short) is given by the following grammar for terms:

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

The constructor $t[x{\leftarrow}u]$ is called an *explicit substitution* (of u for x in t, the usual (implicit) substitution is instead noted $t\{x{\leftarrow}u\}$). Both $\lambda x.t$ and $t[x{\leftarrow}u]$ bind x in t.

The operational semantics of the LSC is defined using contexts. For the scope of the paper it is enough to use *shallow* contexts, simply referred as *context* and defined as (note the absence of the production $t[x{\leftarrow}S]$):

$$S, P, T \quad ::= \quad \langle\cdot\rangle \mid \lambda x.S \mid St \mid tS \mid S[x{\leftarrow}t]$$

A special class of contexts is that of *substitution contexts*, defined by $L ::= \langle\cdot\rangle \mid L[x{\leftarrow}t]$. The (shallow) rewriting rules $\rightarrow_{\mathtt{dB}}$ ($\mathtt{dB} = \beta$ *at a distance*) and $\rightarrow_{\mathtt{ls}}$ (linear substitution) are given by the closure by (shallow) contexts of the following rules:

$$L\langle\lambda x.t\rangle u \quad \mapsto_{\mathtt{dB}} \quad L\langle t[x{\leftarrow}u]\rangle; \qquad\qquad S\langle x\rangle[x{\leftarrow}u] \quad \mapsto_{\mathtt{ls}} \quad S\langle u\rangle[x{\leftarrow}u].$$

The union of $\rightarrow_{\mathtt{dB}}$ and $\rightarrow_{\mathtt{ls}}$ is simply noted $\rightarrow$. Taking the external context into account, a substitution step takes the following *explicit* form: $P\langle S\langle x\rangle[x{\leftarrow}u]\rangle \rightarrow_{\mathtt{ls}} P\langle S\langle u\rangle[x{\leftarrow}u]\rangle$. We shall often use a *compact* form, writing $T\langle x\rangle \rightarrow_{\mathtt{ls}} T\langle u\rangle$ where it is implicitly assumed that $T = P\langle S[x{\leftarrow}u]\rangle$. A *derivation* ρ is a finite sequence of reduction steps. We write $|t|$ for the size of t, $|t|_{[\cdot]}$ for the number of substitutions in t, $|\rho|$ for the length of ρ, and $|\rho|_{\mathtt{dB}}$ for the number of $\mathtt{dB}$-steps in ρ.

(Relative) unfolding. The unfolding $t\downarrow$ of a term t is the λ-term obtained from t by turning its explicit substitutions in implicit ones:

$$x\downarrow \;:=\; x; \qquad (tu)\downarrow \;:=\; t\downarrow u\downarrow; \qquad (\lambda x.t)\downarrow \;:=\; \lambda x.t\downarrow; \qquad (t[x{\leftarrow}u])\downarrow \;:=\; t\downarrow\{x{\leftarrow}u\downarrow\}.$$

We will also need a more general notion, the unfolding $t\downarrow_S$ of t in a context S:

$$t\downarrow_{\langle\cdot\rangle} := t\downarrow; \qquad t\downarrow_{uS} = t\downarrow_{\lambda x.S} = t\downarrow_{Su} := t\downarrow_S; \qquad t\downarrow_{S[x\leftarrow u]} := t\downarrow_S\{x{\leftarrow}u\downarrow\};$$

4 A Bird's Eye View on the Main Result

Our proof method can be described abstractly. We want to show that a certain strategy $\rightsquigarrow$ of λ-calculus (namely the leftmost-outermost strategy) provides a unitary and invariant cost model, *i.e.* that the number of $\rightsquigarrow$ steps is a measure polynomially related to the number of transition on Turing machines. This is done by going through an intermediary computational model, a calculus with explicit substitutions, the LSC, playing the role of a very abstract machine for λ-terms. We are looking for an appropriate strategy $\rightsquigarrow_X$ within the LSC s.t. it is invariant with respect to both $\rightsquigarrow$ and Turing machines. Our target result then decomposes in two parts:

1. *High-Level Implementation*: $\rightsquigarrow$ terminates iff $\rightsquigarrow_X$ terminates. Moreover, $\rightsquigarrow$ is implemented by $\rightsquigarrow_X$ with only a polynomial overhead. Namely, $t \rightsquigarrow_X^k u$ iff $t \rightsquigarrow^h u\downarrow$ with k polynomial in h (our actual bound will be quadratic);
2. *Low-Level Implementation*: $\rightsquigarrow_X$ is implemented on Turing machines with an overhead in time which is polynomial in both k and the size of t.

The high-level part relies on the following notion.

Definition 4.1. *Let $\rightsquigarrow$ be a deterministic strategy on λ-terms and $\rightsquigarrow_X$ a strategy on terms with ES. The pair $(\rightsquigarrow, \rightsquigarrow_X)$ is a* ***high-level implementation system*** *if whenever t is a λ-term and $\rho : t \rightsquigarrow_X^* u$ then:*

1. Normal Form*: if t is a $\rightsquigarrow_X$-normal form then $t\downarrow$ is a $\rightsquigarrow$-normal form.*
2. Projection*: $\rho\downarrow : t \rightsquigarrow^* u\downarrow$ and $|\rho\downarrow| = |\rho|_{\mathtt{dB}}$.*
3. Trace*: the number $|u|_{[\cdot]}$ of ES in u is exactly the number $|\rho|_{\mathtt{dB}}$ of $\mathtt{dB}$-steps in ρ;*
4. Syntactic Bound*: the length of a sequence of substitution steps from u is bounded by $|u|_{[\cdot]}$.*

Then we can prove abstractly the high-level implementation theorem.

Theorem 4.2 (High-Level Implementation)**.** *Let t be an ordinary λ-term and $(\rightsquigarrow, \rightsquigarrow_X)$ a high-level implementation system. Then:*

1. *t is $\rightsquigarrow$-normalising iff it is $\rightsquigarrow_X$-normalising.*
2. *If $\rho : t \rightsquigarrow_X^* u$ then $\rho\downarrow : t \rightsquigarrow^* u\downarrow$ and $|\rho| = O(|\rho\downarrow|^2)$.*

The low-level part, instead, follows from the following notion.

Definition 4.3. *A strategy $\rightsquigarrow_X$ on terms with explicit substitutions is* mechanisable *if for every derivation $\rho : t \rightsquigarrow_X^* u$:*

1. Subterm*: all the subterms duplicated along ρ are subterms of t.*
2. Selection*: the search of the next $\rightsquigarrow_X$ redex to reduce in u takes polynomial time in $|u|$.*

Let us point out that the subterm property implies the trace property in the definition of high-level implementation systems, when the starting term is a λ-term. Indeed, it implies that substitution steps do not change $|\cdot|_{[\cdot]}$, as all duplicated subterms are without explicit substitutions.

At first sight the *selection property* is always trivially verified: finding a redex in u is certainly linear in $|u|$. However — as it will be explained in a moment — our strategy for ES will reduce only redexes satisfying a side-condition whose naïve verification is exponential in $|u|$. Then one has to be sure that such a computation can be done in polynomial time.

Theorem 4.4 (Low-Level Implementation)**.** *Let $\rightsquigarrow_X$ be a mechanisable strategy and let $t \rightsquigarrow_X^k u$. Then there is an algorithm of input t and output u that is polynomial in k and $|t|$.*

Our strategy for β-reduction is the leftmost-outermost (LO) strategy $\rightarrow_{LO}$, that is standard. What is left to do, then, is to find the strategy $\rightsquigarrow_X$ for explicit substitutions, which is both mechanisable and a high-level implementation of $\rightarrow_{LO}$. Unfortunately, the linear LO strategy $\multimap_{LO}$ (defined in [1]) is mechanisable but the pair $(\rightarrow_{LO}, \multimap_{LO})$ does not form a high-level implementation system, since $\multimap_{LO}$ lacks the syntactic bound property. (Just consider the family $\{t_n\}_{n\in\mathbb{N}}$ of terms that evaluate in a linear number of $\multimap$ steps to an output u_n of size exponential in n.)

The key technical contribution of this work is the definition of a constrained, optimized notion of reduction, that will pave the way to the High-Level Implementation Theorem without sacrificing the Low-Level Implementation Theorem. The idea is that an optimised step takes place only if it contributes somehow to explicit a $\rightarrow_{\mathtt{dB}}$-redex. Let an *applicative context* be defined by $A ::= \langle\cdot\rangle u \mid S\langle A\rangle$ (remember that S denotes a shallow context). Then:

Definition 4.5 (Useful Steps and Derivations)**.** *An* useful *step is either a* dB*-step or a* ls*-step* $S\langle x\rangle \multimap_{\mathtt{ls}} S\langle r\rangle$ *(in compact form) s.t.* $r{\downarrow_S}$*:*
1. contains a β*-redex, or*
2. it is an abstraction and S *is an applicative context.*
A useful derivation *is a derivation whose steps are useful.*

The notion of small-step evaluation that we will use to implement LO β-reduction is the one of LO *useful* (LOU) derivation. This is defined by just imposing that at any step the fired redex is the LO among all *useful* redexes. In LO derivations, on the other hand, we are allowed to fire the LO redex, even if the latter is *not* useful in the sense of Definition 4.5.

We need to ensure that LOU derivations are mechanisable and form an high-level implementation system when paired with LO β-derivations. The proof of these two facts is a sophisticated detour in four steps:

1. The *subterm* and *trace properties*, by showing that:
   - standard derivations for the LSC (introduced in [1]) have the subterm property;
   - LOU derivations are standard;
2. The *normal form* and *projection properties*, by a careful study of unfoldings, LO, and LOU derivations;
3. The *syntactic bound property*, by:
   - introducing the new notion of *nested derivations* and show that they satisfy the property;
   - showing that LOU derivations are nested;
4. The *selection* property, by exhibiting an algorithm to test whether a redex is useful or not, which works in time polynomial on the size of the underlying (compressed) term.

Unfortunately, lack of space prevents us from giving any detail on the four steps above.

References

[1] B. Accattoli, E. Bonelli, D. Kesner, and C. Lombardi. A Nonstandard Standardization Theorem. In *POPL*, pages 659–670, 2014.

[2] B. Accattoli and U. Dal Lago. On the invariance of the unitary cost model for head reduction. In *RTA*, pages 22–37, 2012.

[3] A. Asperti and H. G. Mairson. Parallel beta reduction is not elementary recursive. In *POPL*, pages 303–315, 1998.

[4] M. Avanzini and G. Moser. Closing the gap between runtime complexity and polytime computability. In *RTA*, pages 33–48, 2010.

[5] U. Dal Lago and S. Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[6] U. Dal Lago and S. Martini. On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science*, 8(3), 2012.

[7] J. L. Lawall and H. G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? In *ICFP*, pages 92–101, 1996.

[8] J.-J. Lévy. Réductions correctes et optimales dans le lambda-calcul. Thése d'Etat, Univ. Paris VII, France, 1978.

[9] C. F. Slot and P. van Emde Boas. On tape versus core; an application of space efficient perfect hash functions to the invariance of space. In *STOC*, pages 391–400, 1984.