Beta Reduction is Invariant, Indeed

Beniamino Accattoli, Ugo Dal Lago

Beta Reduction is Invariant, Indeed

Beniamino Accattoli
Università di Bologna
beniamino.accattoli@gmail.com

Ugo Dal Lago
Università di Bologna & INRIA
ugo.dallago@unibo.it

Abstract

Slot and van Emde Boas' weak 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 length of a leftmost-outermost derivation to normal form is an invariant cost model. Such a theorem cannot be proved by directly relating λ-calculus with Turing machines or random access machines, because of the *size explosion problem*: there are terms that in a linear number of steps produce an exponentially long output. The first step towards the solution is to shift to a notion of evaluation for which the length and the size of the output are linearly related. This is done by adopting the linear substitution calculus (LSC), a calculus of explicit substitutions modelled after linear logic proof nets and admitting a decomposition of leftmost-outermost derivations with the desired property. Thus, the LSC is invariant with respect to, say, random access machines. The second step is to show that LSC is invariant with respect to the λ-calculus. The size explosion problem seems to imply that this is not possible: having the same notions of normal form, evaluation in the LSC is exponentially longer than in the λ-calculus. We solve such an *impasse* by introducing a new form of shared normal form and shared reduction, deemed *useful*. Useful evaluation avoids those steps that only unshare the output without contributing to β-redexes, *i.e.* the steps that cause the blow-up in size. The main technical contribution of the paper is indeed the definition of useful reductions and the thorough analysis of their properties.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages — Operational Semantics.; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic — Lambda Calculus and Related Systems.

General Terms Theory

Keywords λ-calculus, computational complexity, cost models, explicit substitutions, sharing

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2886-9…$15.00.
http://dx.doi.org/10.1145/2603088.2603105

1. Introduction

Theoretical computer science is built around algorithms, computational models, and machines: an algorithm describes a solution to a problem with respect to a fixed computational model, whose role is to provide a handy abstraction of concrete machines. The choice of the model reflects a tension between different needs. For complexity analysis, one expects a neat relationship between the 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 specification of algorithms unfortunately lies at the other end of the spectrum, as one would like them to be as machine-independent as possible. In this case programming languages are the typical model. Functional programming languages, thanks to their higher-order nature, provide very concise and abstract specifications. Their strength is also their weakness: 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 [25]:

Reasonable computational models simulate each other
with polynomially bounded overhead in time,
and constant factor overhead in space.

The *weak* invariance thesis is the variant where the requirement about space is dropped, and it is the one we will actually work with in this paper. The idea behind the thesis is that for reasonable models the definition of every polynomial or super-polynomial class such as $\mathbf{P}$ or $\mathbf{EXP}$ does not rely on the chosen model. On the other hand, it is well-known that sub-polynomial classes depend very much on the model, and thus it does not really make sense to pursue a linear rather than polynomial relationship.

A first refinement of our question then is: are functional languages 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 consider the *unitary* cost model, *i.e.* take the number of evaluation steps as the cost of the underlying term. 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 [20]) to normal form has been shown *not* to be a reasonable cost model: there exists a family of terms that reduces in a polynomial number of parallel β-steps, but whose complexity is non-elementary [7, 19]. If one considers the number of *sequential* β-steps (in a given strategy, for a given notion of reduction), the literature offers some partial positive results, all relying on the use of sharing (see below for more details). Some quite general results [8, 14] have been obtained through graph rewriting, itself a form of sharing, when only first order symbols are considered.

Sharing is indeed a key ingredient, for one of the issues here is due to the *representation of terms*. The ordinary way of representing terms indeed suffers from the *size explosion problem*: even for the most restrictive notions of reduction (e.g. Plotkin's weak reduction), there is a family of terms $\{t_n\}_{n\in\mathbb{N}}$ such that $|t_n|$ is linear in n, t_n evaluates to its normal form in n steps, but at i-th step a term of size 2^i is copied, producing a normal form of size exponential in n. Put differently, an evaluation sequence of linear length can possibly produce an output of exponential size. At first sight, then, there is no hope that evaluation lengths may provide an invariant cost model. The idea is that such an *impasse* can be avoided by sharing common sub-terms along the evaluation process, in order to keep the representation of the output compact, *i.e.* polynomially related to the number of evaluation steps. 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 [13, 14] and for head reduction [3].

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 not knowing whether the λ-calculus in its full generality is a reasonable machine is embarrassing for the λ-calculus community. 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 paper, we give a positive answer to the question above, by showing that leftmost-outermost (LO, for short) reduction *to normal form* indeed induces an invariant cost model. Such an evaluation strategy is *standard*, in the sense of the standardisation theorem, one of the central theorems in the theory of λ-calculus, first proved by Curry and Feys [12]. The relevance of our cost model is given by the fact that LO reduction is an abstract concept from rewriting theory which at first sight is totally unrelated to complexity analysis. In particular, our cost model is completely machine-independent.

Another view on this problem comes in fact from rewriting theory itself. It is common practice to specify the operational semantics of a language via a rewriting system, whose rules always employ some form of substitution, or at least of copying, of subterms. Unfortunately, this practice is very far away from the way languages are implemented. Indeed, actual interpreters perform copying in a very controlled way (see, *e.g.*, [23, 27]). This discrepancy induces serious doubts about the relevance of the computational model. Is there any theoretical justification for copy-based models, or more generally for rewriting theory as a modelling tool? In this paper we give a very precise answer, formulated within rewriting theory itself.

As in our previous work [3], we prove our result by means of the *linear substitution calculus* (see also [1, 6]), a simple calculus of explicit substitutions (ES, for short) arising from linear logic and graphical syntaxes and similar to calculi studied by De Bruijn [16], Nederpelt [22], and Milner [21]. A peculiar feature of the linear substitution calculus (LSC) is the use of rewriting rules *at a distance*, *i.e.* rules defined by means of contexts, that are used to closely mimic reduction in linear logic proof nets. Such a framework—whose use does not require any knowledge of these areas—allows an easy management of sharing and, in contrast to previous approaches to ES, admits a theory of standardisation and a notion of LO evaluation [6]. The proof of our result indeed is a *tour de force* based on a fine quantitative study of the relationship between LO derivations for the λ-calculus and a variation over LO derivations for the LSC. Roughly, the latter avoids the size explosion problem while keeping a polynomial relationship with the former.

Let us point out that invariance results usually have two directions, while we here study only one of them (namely that the λ-calculus can be efficiently simulated by, say, Turing machines). The missing half is a much simpler problem already solved in [3]: there is an encoding of Turing machines into λ-terms s.t. their execution is simulated by weak head β-reduction with only a linear overhead.

On Invariance and Complexity Analysis. Before proceeding, let us stress some crucial points:

1. *ES Are Only a Tool.* Although ES are an essential *tool* for the proof of our result, the *result itself* is about the usual, pure, λ-calculus. In particular, the invariance result can be used without any need to care about ES: we are allowed to measure the complexity of problems by simply bounding the *number* of LO β-steps taken by any λ-term solving the problem.
2. *Complexity Classes in the λ-Calculus.* The main consequence of our invariance result is that every polynomial or super-polynomial class, like $\mathbf{P}$ of $\mathbf{EXP}$, can be defined using λ-calculus (and LO β-reduction) instead of Turing machines.
3. *Our Cost Model is Unitary.* An important point is that our cost model is *unitary*, and thus attributes a constant cost to any LO step. One could argue that it is always possible to reduce λ-terms on abstract or concrete machines and take that number of steps as *the* cost model. First, such a measure of complexity would be very machine-dependent, against the very essence of λ-calculus. Second, these cost models invariably attribute a more-than-constant cost to any β-step, making the measure much harder to use and analyse. It is not evident that a computational model enjoys a unitary invariant cost model. As an example, if multiplication is a primitive operation, random access machines need to be endowed with a *logarithmic* cost model in order to obtain invariance.

The next section explains why the problem at hand is hard, and in particular why iterating our previous results on head reduction [3] does not provide a solution. An extended version of this paper with more details is available [2].

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 that sharing is unavoidable and that a new subtle notion of sharing is necessary.

If we stick to explicit representations of terms, in which sharing is not allowed, counterexamples to invariance can be designed in a fairly easy way. Let u be the lambda term yxx and consider the sequence $\{t_n\}_{n\in\mathbb{N}}$ of λ-terms defined as $t_0 = u$ and $t_{n+1} = (\lambda x.t_n)u$ for every $n \in \mathbb{N}$. 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 LO reduction order; as an example:

$$t_0 = u = r_0;$$
$$t_1 \to yuu = yr_0r_0 = r_1;$$
$$t_2 \to (\lambda x.t_0)(yuu) = (\lambda x.u)r_1 \to yr_1r_1 = r_2.$$

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 [3] is based on ES, 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 (where ES are denoted with $t[x{\leftarrow}u]$):

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

As one can easily verify, the size of p_n is linear in n. More generally, linear head reduction (LHR) 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 [3] or below for more details.

It may seem that one is then forced to use ES to measure complexity. In [3] 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 done via LHR and be only polynomially (actually quadratically) longer.

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 [6]. 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 substitutions 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 related 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 related 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 — 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 ES *lying between* two β-redexes, so that proceeding in a LO way would unfold the problematic substitutions anyway.

Our notion of *useful* step will elaborate on this idea, by computing *partial* unfoldings, to check if a substitution step contributes or will contribute to some future β-redex. Of course, we will have to show that such tests can be themselves performed in polynomial time, 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 (see [10]). The language of the *linear substitution calculus* (LSC for short) is given by the following grammar for terms:

$$t, u, r, p ::= x \;\big|\; \lambda x.t \;\big|\; tu \;\big|\; t[x{\leftarrow}u].$$

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, and we silently work modulo α-equivalence of these bound variables, *e.g.* $(xy)[y{\leftarrow}t]\{x{\leftarrow}y\} = (yz)[z{\leftarrow}t]$. We use $\mathtt{fv}(t)$ for the set of free variables of t.

Contexts. The operational semantics of the LSC is parametric in a notion of (one-hole) context. General contexts are defined by:

$$C ::= \langle\cdot\rangle \;\big|\; \lambda x.C \;\big|\; Ct \;\big|\; tC \;\big|\; C[x{\leftarrow}t] \;\big|\; t[x{\leftarrow}C],$$

and the plugging of a term t into a context C is defined as $\langle\cdot\rangle\langle t\rangle := t$, $(\lambda x.C)\langle t\rangle := \lambda x.(C\langle t\rangle)$, and so on. As usual, plugging in a context can capture variables, *e.g.* $((\langle\cdot\rangle y)[y{\leftarrow}t])\langle y\rangle = (yy)[y{\leftarrow}t]$. The plugging $C\langle D\rangle$ of a context D into a context C is defined analogously.

Along most of the paper, however, we will not need such a general notion of context. In fact, our study takes a simpler form if the operational semantics is defined with respect to **shallow** contexts, defined as (note the absence of the production $t[x{\leftarrow}S]$):

$$S, P, T, V ::= \langle\cdot\rangle \;\big|\; \lambda x.S \;\big|\; St \;\big|\; tS \;\big|\; S[x{\leftarrow}t].$$

In the following, whenever we refer to a *context* without further specification it is implicitly assumed that it is a *shallow* context. A special class of contexts is that of **substitution contexts**:

$$L ::= \langle\cdot\rangle \;\big|\; L[x{\leftarrow}t].$$

Operational Semantics. The (shallow) rewriting rules $\to_{\mathtt{dB}}$ ($\mathtt{dB}$ = *β at a distance*) and $\to_{\mathtt{ls}}$ (linear substitution) are given by the closure by (shallow) contexts of the following rules:

$$\begin{aligned} L\langle\lambda x.t\rangle u &\mapsto_{\mathtt{dB}} L\langle t[x{\leftarrow}u]\rangle; \\ S\langle x\rangle[x{\leftarrow}u] &\mapsto_{\mathtt{ls}} S\langle u\rangle[x{\leftarrow}u]. \end{aligned}$$

The union of $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ls}}$ is simply noted $\to$. The rewriting rules are assumed to use *on-the-fly* α-equivalence to avoid variable capture. For instance,

$$\begin{aligned} (\lambda x.t)[y{\leftarrow}u]y &\to_{\mathtt{dB}} t\{y{\leftarrow}z\}[x{\leftarrow}y][z{\leftarrow}u] \qquad \text{for } z \notin \mathtt{fv}(t); \\ (\lambda y.(xy))[x{\leftarrow}y] &\to_{\mathtt{ls}} (\lambda z.(yz))[x{\leftarrow}y]. \end{aligned}$$

Moreover, in rule $\mathtt{ls}$ the context S is assumed to not capture x, so that $(\lambda x.x)[x{\leftarrow}y] \not\to_{\mathtt{ls}} (\lambda x.y)[x{\leftarrow}y]$.

The just defined shallow fragment simply ignores garbage collection (that in the LSC can always be postponed [1]) and lacks some of the nice properties of the LSC (obtained simply by replacing shallow contexts by general contexts). Its relevance is the fact that it is the smallest fragment implementing linear LO reduction. The following are examples of shallow steps:

$$\begin{aligned} (\lambda x.x)y &\to_{\mathtt{dB}} x[x{\leftarrow}y]; \\ (xx)[x{\leftarrow}t] &\to_{\mathtt{ls}} (xt)[x{\leftarrow}t]; \end{aligned}$$

while the following steps are not

$$\begin{aligned} t[z{\leftarrow}(\lambda x.x)y] &\to_{\mathtt{dB}} t[z{\leftarrow}x[x{\leftarrow}y]]; \\ x[x{\leftarrow}y][y{\leftarrow}t] &\to_{\mathtt{ls}} x[x{\leftarrow}t][y{\leftarrow}t]. \end{aligned}$$

Taking the external context into account, a substitution step has the following *explicit* form: $P\langle S\langle x\rangle[x{\leftarrow}u]\rangle \to_{\mathtt{ls}} P\langle S\langle u\rangle[x{\leftarrow}u]\rangle$. We shall often use a *compact* form, writing $T\langle x\rangle \to_{\mathtt{ls}} T\langle u\rangle$ where it is implicitly assumed that $T = P\langle S[x{\leftarrow}u]\rangle$. We use R and Q

as metavariables for redexes. A **derivation** $\rho : t \to^k u$ is a finite sequence of reduction steps, sometimes given as $R_1; \ldots; R_k$, *i.e.* as the sequence of reduced redexes. We write $|t|$ for the size of t, $|t|_{[\cdot]}$ for the number of substitutions in t, $|\rho|$ for the length of ρ, and $|\rho|_{\mathtt{dB}}$ (resp. $|\rho|_{\mathtt{ls}}$) for the number of dB-steps (resp. ls-steps) in ρ.

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

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

We will also need a more general notion, the unfolding $t{\downarrow}_S$ of t in a context S:

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

For instance,

$$(x(yz))[y{\leftarrow}x][x{\leftarrow}z]{\downarrow} = z(zz);$$
$$(xy){\downarrow}_{(\langle\cdot\rangle[y\leftarrow x]t)[x\leftarrow\lambda z.(zz)]} = (\lambda z.(zz))\lambda z.(zz).$$

We extend implicit substitutions and unfoldings to contexts by setting $\langle\cdot\rangle\{x{\leftarrow}t\} := \langle\cdot\rangle$ and $\langle\cdot\rangle{\downarrow} := \langle\cdot\rangle$ (all other cases are defined as expected, *e.g.* $S[x{\leftarrow}t]{\downarrow} := S{\downarrow}\{x{\leftarrow}t{\downarrow}\}$). We also write $S \prec_p t$ if there is a term u s.t. $S\langle u\rangle = t$, call it the **prefix relation**. We have the following properties, that only hold because our contexts are shallow (implying that the hole cannot be duplicated during the unfolding).

Lemma 3.1. *Let S be a shallow context. Then:*
1. *$S{\downarrow}$ is a shallow context;*
2. *$S\langle t\rangle\{x{\leftarrow}u\} = S\{x{\leftarrow}u\}\langle t\{x{\leftarrow}u\}\rangle$;*
3. *$S\langle t\rangle{\downarrow} = S{\downarrow}\langle t{\downarrow}_S\rangle$, in particular if $S \prec_p t$ then $S{\downarrow} \prec_p t{\downarrow}$.*

Given a derivation $\rho : t \to^* u$ in the LSC, we often consider the β-derivation $\rho{\downarrow} : t{\downarrow} \to_\beta^* u{\downarrow}$ obtained by projecting ρ via unfolding.

Reduction Combinatorics. Given any calculus, a deterministic strategy $\to$ for it, and a term t, the expression $\#_\to(t)$ stands for the number of reduction steps necessary to reach the normal form of t along $\to$, or ∞ if t diverges. Similarly, given a natural number n, the expression $\to^n(t)$ stands for the term u such that $t \to^n u$, if $n \le \#_\to(t)$, or for the normal form of t otherwise.

4. The Proof, Made Abstract

Our proof method can be described abstractly. Such an approach both clarifies the structure of the proof and prepares the ground for possible generalisations to, *e.g.*, the call-by-value λ-calculus or calculi with additional features as pattern matching or control operators. We want to show that a certain strategy $\rightsquigarrow$ for the λ-calculus provides a unitary and invariant cost model, *i.e.* that the number of $\rightsquigarrow$ steps is a measure polynomially related to the number of transitions on Turing machines. As explained in the introduction, we pass through an intermediary computational model, a calculus with ES, the *linear substitution calculus*, playing the role of a very abstract machine for λ-terms.

We are looking for an appropriate strategy $\rightsquigarrow_X$ within the LSC which is invariant with respect to both $\rightsquigarrow$ and Turing machines. Then we need two theorems, which together form the main result of the paper:
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 of the LSC. 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 u is a $\rightsquigarrow_X$-normal form then $u{\downarrow}$ 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 dB-steps in ρ;*
4. Syntactic Bound*: the length of a sequence of substitution steps from u is bounded by $|u|_{[\cdot]}$.*

Concretely, the high-level implementation system at work in the paper will take as $\rightsquigarrow$ the LO strategy of the λ-calculus and as $\rightsquigarrow_X$ *a variant* of the linear LO strategy for the LSC. A variant is required because, as we will explain, the linear LO strategy of the LSC does not satisfy the syntactic bound property.

The normal form and projection properties address the *qualitative* part of the high-level implementation theorem, *i.e.* the part about termination. The normal form property guarantees that $\rightsquigarrow_X$ does not stop prematurely, so that when $\rightsquigarrow_X$ terminates $\rightsquigarrow$ cannot keep going. The projection property guarantees that termination of $\rightsquigarrow$ implies termination of $\rightsquigarrow_X$. The two properties actually state a stronger fact: *$\rightsquigarrow$ steps can be identified with the dB-steps of the $\rightsquigarrow_X$ strategy.*

The trace and syntactic bound properties are instead used for the *quantitative* part of the theorem, *i.e.* to provide the polynomial bound. The two properties together provide a bound the number of ls-steps in a $\rightsquigarrow_X$ derivation with respect to the number of dB-steps, that—by the identification of β and dB redexes—is exactly the length of the associated $\rightsquigarrow$ derivation.

The high-level part can now be proved abstractly.

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 $|\rho| = O(|\rho{\downarrow}|^2)$.*

Proof. 1. $\Leftarrow$) Suppose that t is $\rightsquigarrow_X$-normalisable and let $\rho : t \rightsquigarrow_X^* u$ a derivation to $\rightsquigarrow_X$-normal form. By the projection property there is a derivation $t \rightsquigarrow^* u{\downarrow}$. By the normal form property $u{\downarrow}$ is a $\rightsquigarrow$-normal form.

$\Rightarrow$) Suppose that t is $\rightsquigarrow$-normalisable and let $\tau : t \rightsquigarrow^k u$ be the derivation to $\rightsquigarrow$-normal form (unique by determinism of $\rightsquigarrow$). Assume, by contradiction, that t is not $\rightsquigarrow_X$-normalisable. Then there is a family of $\rightsquigarrow_X$-derivations $\rho_i : t \rightsquigarrow_X^i u_i$ with $i \in \mathbb{N}$, each one extending the previous one. By the syntactic bound property, $\rightsquigarrow_X$ can make only a finite number of ls steps (more generally, $\to_{\mathtt{ls}}$ is strongly normalising in the LSC). Then the sequence $\{|\rho_i|_{\mathtt{dB}}\}_{i\in\mathbb{N}}$ is non-decreasing and unbounded. By the projection property, the family $\{\rho_i\}_{i\in\mathbb{N}}$ unfolds to a family of $\rightsquigarrow$-derivations $\{\rho_i{\downarrow}\}_{i\in\mathbb{N}}$ of unbounded length (in particular greater than k), absurd.

2. By the projection property, it follows that $\rho{\downarrow} : t \rightsquigarrow^* u{\downarrow}$. Moreover, to show $|\rho| = O(|\rho{\downarrow}|^2)$ it is enough to show $|\rho| = O(|\rho|_{\mathtt{dB}}^2)$. Now, ρ has the shape:
$$t = r_1 \to_{\mathtt{dB}}^{a_1} p_1 \to_{\mathtt{ls}}^{b_1} r_2 \to_{\mathtt{dB}}^{a_2} p_2 \to_{\mathtt{ls}}^{b_2} \ldots r_k \to_{\mathtt{dB}}^{a_k} p_k \to_{\mathtt{ls}}^{b_k} u.$$
By the syntactic bound property, we obtain $b_i \le |p_i|_{[\cdot]}$. By the trace property we obtain $|p_i|_{[\cdot]} = \sum_{j=1}^{i} a_j$, and so $b_i \le \sum_{j=1}^{i} a_j$. Then:
$$|\rho|_{\mathtt{ls}} = \sum_{i=1}^{k} b_i \le \sum_{i=1}^{k}\sum_{j=1}^{i} a_j.$$
Note that $\sum_{j=1}^{i} a_j \le \sum_{j=1}^{k} a_j = |\rho|_{\mathtt{dB}}$ and $k \le |\rho|_{\mathtt{dB}}$. So

$$|\rho|_{\mathtt{ls}} \leq \sum_{i=1}^{k} \sum_{j=1}^{i} a_j \leq \sum_{i=1}^{k} |\rho|_{\mathtt{dB}} \leq |\rho|_{\mathtt{dB}}^2.$$

Finally, $|\rho| = |\rho|_{\mathtt{dB}} + |\rho|_{\mathtt{ls}} \leq |\rho|_{\mathtt{dB}} + |\rho|_{\mathtt{dB}}^2 = O(|\rho|_{\mathtt{dB}}^2)$. □

For the low-level part we rely on the following notion.

Definition 4.3. *A strategy $\rightsquigarrow_X$ on LSC terms is* ***mechanisable*** *if given a derivation $\rho : t \rightsquigarrow_X^* u$:*

1. Subterm*: the terms 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 $|u|$.*

The *subterm property*—essentially—guarantees that any step has a linear cost in the size of the initial term, the fundamental parameter for complexity; it will be discussed in more detail in Sect. 6. At first sight the *selection property* is always trivially verified: finding a redex in u takes time linear in $|u|$. However, 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. Then there is an algorithm that on input t and k outputs $\rightsquigarrow_X^k(t)$, and which works in time polynomial in k and $|t|$.*

Proof. By the subterm property, implementing one step takes time polynomial (if not linear) in $|t|$. An immediate consequence of the subterm property is the *no size explosion property*, *i.e.* that $|u| \leq (k+1) \cdot |t|$. By the selection property selecting the next redex takes time polynomial in $|u|$, that by the no size explosion property is polynomial in k and $|t|$. The composition of polynomials is again a polynomial, and so selecting the redex takes time polynomial in k and $|t|$. Hence, the reduction can be implemented in polynomial time. □

In [3], we proved that *head reduction* and *linear head reduction* form a high-level implementation system and that linear head reduction is mechanisable, even if we did not use such a terminology, nor were we aware of the presented abstract scheme. In order to extend such a result to normal forms we need to replace head reduction with a normalising strategy (*i.e.* a strategy reaching the β-normal form, if any).

One candidate for $\rightsquigarrow$ is the LO strategy $\rightarrow_{LO\beta}$. Such a choice is natural, as $\rightarrow_{LO\beta}$ is normalising, it produces standard derivations, and it is an iteration of head reduction. What is left to do, then, is to find a strategy $\rightsquigarrow_X$ for ES, which is both mechanisable and a high-level implementation of $\rightarrow_{LO\beta}$. Unfortunately, the linear LO strategy, noted $\rightarrow_{LO}$ and first defined in [6], is mechanisable but the pair $(\rightarrow_{LO\beta}, \rightarrow_{LO})$ is not a high-level implementation system.

In general, mechanisable strategies are not hard to find. As we will show in Sect. 6, the whole class of *standard* derivations for ES has the subterm property. In particular, the linear strategy $\rightarrow_{LO}$—which is standard—enjoys all the other properties *but* for the syntactic bound property.

Such a problem will be solved by LO *useful* derivations, to be introduced in Sect. 5, that will be shown to be both mechanisable and a high-level implementation of $\rightarrow_{LO\beta}$. Useful derivations avoid those substitution steps that only explicit the normal form without contributing to explicit $\beta/\mathtt{dB}$-redexes (that, by the projection property, can be identified). LO useful derivations will have all the nice properties of LO derivations and moreover will stop on shared, minimal representations of normal forms, solving the problem with linear LO derivations.

Let us point out that our analysis would be vacuous without evidence that useful normal forms are a reasonable representation of λ-terms. In other words, we must be sure that ES do not hide (too much of) the inherent difficulty of reducing λ-terms under the carpet of sharing. In [3], we solved this issue by providing an efficient algorithm for checking the equality of any two LSC terms—thus in particular of useful normal forms—without computing their unfoldings (that otherwise would reintroduce an exponential blow-up). Some further discussion can be found in Sect. 10.

5. Useful Derivations

In this section we define a constrained, optimised notion of reduction, that will be the key to the High-Level Implementation Theorem. The idea is that an optimised step takes place only if it somehow contributes to explicit a $\beta/\mathtt{dB}$-redex. Let an **applicative context** be defined by $A ::= S\langle Lt\rangle$, where S and L are a shallow and a substitution context, respectively (note that applicative contexts are *not* made out of applications only; for instance $t\lambda x.(\langle\cdot\rangle[y\leftarrow u]r)$ is an applicative context). Then:

Definition 5.1 (Useful/Useless Steps and Derivations)**.** *A* ***useful*** *step is either a $\mathtt{dB}$-step or a $\mathtt{ls}$-step $S\langle x\rangle \rightarrow_{\mathtt{ls}} S\langle r\rangle$ (in compact form) s.t. $r\downarrow_S$:*

1. *either contains a β-redex,*
2. *or is an abstraction and S is an applicative context.*

A ***useless*** *step is a $\mathtt{ls}$-step that is not useful. A* ***useful derivation*** *(resp.* ***useless derivation****) is a derivation whose steps are useful (resp. useless).*

Let us give some examples. The steps

$$(tx)[x\leftarrow(\lambda y.y)u] \rightarrow_{\mathtt{ls}} (t((\lambda y.y)u))[x\leftarrow(\lambda y.y)u];$$
$$(xt)[x\leftarrow\lambda y.y] \rightarrow_{\mathtt{ls}} ((\lambda y.y)t)[x\leftarrow\lambda y.y];$$

are useful because they move or create a $\beta/\mathtt{dB}$-redex (first and second case of the definition, respectively) while

$$(\lambda x.y)[y\leftarrow zz] \rightarrow_{\mathtt{ls}} (\lambda x.(zz))[y\leftarrow zz]$$

is useless. However, useful steps are subtler, for instance

$$(tx)[x\leftarrow zz][z\leftarrow\lambda y.y] \rightarrow_{\mathtt{ls}} (t(zz))[x\leftarrow zz][z\leftarrow\lambda y.y]$$

is useful also if it does not move or create $\beta/\mathtt{dB}$-redexes, because it does so up to relative unfolding, *i.e.* $(zz)\downarrow_{\langle\cdot\rangle[z\leftarrow\lambda y.y]} = (\lambda y.y)\lambda y.y$ that is a $\beta/\mathtt{dB}$-redex.

Note that useful steps concern future creations of β-redexes and yet their definition circumvents the explicit use of residuals, relying on relative unfoldings only.

Leftmost-Outermost Useful Derivations. The notion of small-step evaluation that we will use to implement LO β-reduction is the one of LO useful derivation. We need some preliminary definitions.

Let R be a redex. Its **position** is defined as follows:

1. If R is a $\mathtt{dB}$-redex $S\langle L\langle\lambda x.t\rangle u\rangle \rightarrow_{\mathtt{dB}} S\langle L\langle t[x\leftarrow u]\rangle\rangle$ then its position is given by the context S surrounding the changing expression; β-redexes are treated as $\mathtt{dB}$-redexes.
2. If R is a $\mathtt{ls}$-redex, expressed in compact form $S\langle x\rangle \rightarrow_{\mathtt{ls}} S\langle u\rangle$, then its position is the context S surrounding the variable occurrence to substitute.

The left-to-right outside-in order on redexes is expressed as an order on positions, *i.e.* contexts. Let us warn the reader about a possible source of confusion. The *left-to-right outside-in* order in the next definition is sometimes simply called *left-to-right* (or simply *left*) order. The former terminology is used when terms are seen as trees (where the left-to-right and the outside-in orders are disjoint), while the latter terminology is used when terms are seen as strings (where the left-to-right is a total order). While the study of standardisation for the LSC [6] uses the string approach (and thus only talks about the *left-to-right* order and the *leftmost* redex), here some of the proofs (see the long version [2]) require a delicate analysis of the relative positions of redexes and so we prefer the more informative tree approach and define the order formally.

Definition 5.2. *The following definitions are given with respect to general (not necessarily shallow) contexts, even if apart from the next section we will use them only for shallow contexts.*

1. *The* ***outside-in order****:*
   1. Root*: $\langle\cdot\rangle \prec_O C$ for every context $C \neq \langle\cdot\rangle$;*
   2. Contextual closure*: If $C \prec_O D$ then $E\langle C\rangle \prec_O E\langle D\rangle$ for any context E.*

   Note that $\prec_O$ can be seen as the prefix relation $\prec_p$ on contexts.
3. *The* ***left-to-right order****: $C \prec_L D$ is defined by:*
   1. Application*: If $C \prec_p t$ and $D \prec_p u$ then $Cu \prec_L tD$;*
   2. Substitution*: If $C \prec_p t$ and $D \prec_p u$ then $C[x{\leftarrow}u] \prec_L t[x{\leftarrow}D]$;*
   3. Contextual closure*: If $C \prec_L D$ then $E\langle C\rangle \prec_L E\langle D\rangle$ for any context E.*
4. *The* ***left-to-right outside-in order****: $C \prec_{LO} D$ if $C \prec_O D$ or $C \prec_L D$:*

The following are a few examples. For every context C, it holds that $\langle\cdot\rangle \not\prec_L C$. Moreover,

$$\begin{aligned}(\lambda x.\langle\cdot\rangle)t &\prec_O (\lambda x.(\langle\cdot\rangle[y{\leftarrow}u])r)t;\\ (\langle\cdot\rangle t)u &\prec_L (rt)\langle\cdot\rangle;\\ t[x{\leftarrow}\langle\cdot\rangle]u &\prec_L t[x{\leftarrow}r]\langle\cdot\rangle.\end{aligned}$$

The next lemma guarantees that we defined a total order.

Lemma 5.3 (Totality of $\prec_{LO}$)**.** *If $C \prec_p t$ and $D \prec_p t$ then either $C \prec_{LO} D$ or $D \prec_{LO} C$ or $C = D$.*

The orders above can be extended from contexts to redexes, in the expected way, *e.g.* for $\prec_{LO}$ given two redexes R and Q of positions S and P we write $R \prec_{LO} Q$ if $S \prec_{LO} P$. Now, we can define the notions of derivations we are interested in.

Definition 5.4 (Leftmost-Outermost (Useful) Redex)**.** *Let t be a term and R a redex of t. R is the* ***leftmost-outermost*** *(resp.* ***leftmost-outermost useful****, LOU for short) redex of t if $R \prec_{LO} Q$ for every other redex (resp. useful redex) Q of t. We write $t \to_{LO} u$ (resp. $t \to_{LOU} u$) if a step reduces the LO (resp. LOU) redex.*

We need to ensure that LOU derivations are mechanisable and form a high-level implementation system when paired with LO derivations. In particular, we will show:

1. the *subterm* and *trace properties*, by first showing that they hold for every standard derivation, in Sect. 6, and then showing that LOU derivations are standard, in Sect. 7;
2. the *normal form* and *projection properties*, by a careful study of unfoldings and LO/LOU derivations, in Sect. 8;
3. the *syntactic bound property*, passing through the abstract notion of *nested derivation*, in Sect. 9;
4. the *selection property*, by exhibiting a polynomial algorithm to test whether a redex is useful or not, in Sect. 10.

6. Standard Derivations

We need to show that LOU derivations have the subterm property. It could be done directly. However, we will proceed in an abstract way, by first showing that the subterm property is a property of standard derivations for the LSC, and then showing (in Sect. 7) that LOU derivations are standard. The detour has the purpose of shedding a new light on the notion of standard derivation, a classic concept in rewriting theory. For the sake of readability, we use the concept of residual without formally defining it (see [6] for details).

Definition 6.1 (Standard Derivation)**.** *A derivation $\rho : R_1; \ldots ; R_n$ is* ***standard*** *if R_i is not the residual of a redex $Q \prec_{LO} R_j$ for every $i \in \{2, \ldots , n\}$ and $j < i$.*

The same definition where terms are ordinary λ-terms gives the ordinary notion of standard derivation.

Note that any single reduction step is standard. Then, notice that standard derivations select redexes in a left-to-right and outside-in way, but they are not necessarily LO. For instance, the derivation

$$((\lambda x.y)y)[y{\leftarrow}z] \to_{\mathtt{ls}} ((\lambda x.z)y)[y{\leftarrow}z] \to_{\mathtt{ls}} ((\lambda x.z)z)[y{\leftarrow}z]$$

is standard even if the LO redex (*i.e.* the $\mathtt{dB}$-redex on x) is not reduced. The extension of the derivation with $((\lambda x.z)z)[y{\leftarrow}z] \to_{\mathtt{dB}} z[x{\leftarrow}z][y{\leftarrow}z]$ is not standard. Last, note that the position of a $\mathtt{ls}$-step is given by the substituted occurrence and not by the ES, that is $(xy)[x{\leftarrow}u][y{\leftarrow}t] \to_{\mathtt{ls}} (xt)[x{\leftarrow}u][y{\leftarrow}t] \to_{\mathtt{ls}} (ut)[x{\leftarrow}u][y{\leftarrow}t]$ is not standard.

In [6] it is showed that in the full LSC standard derivations are complete, *i.e.* that whenever $t \to^* u$ there is a standard derivation from t to u. The shallow fragment does not enjoy such a standardisation theorem, as the residuals of a shallow redex need not be shallow. This fact however does not clash with the technical treatment in this paper. The shallow restriction is indeed compatible with standardisation in the sense that:

1. *The linear LO strategy is shallow*: if the initial term is a λ-term then every redex reduced by the linear LO strategy is shallow (every non-shallow redex R is contained in a substitution, and every substitution is involved in an outer redex Q);
2. *$\prec_{LO}$-ordered shallow derivations are standard*: any strategy picking shallow redexes in a left-to-right and outside-in fashion does produce standard derivations (it follows from the easy fact that a shallow redex R cannot turn a non-shallow redex Q s.t. $Q \prec_{LO} R$ into a shallow redex).

Moreover, the only redex swaps we will consider (Lemma 7.1) will produce shallow residuals.

We are now going to show a fundamental property of standard derivations. The *subterm property* states that at any point of a derivation $\rho : t \to^* u$ only sub-terms of the initial term t are duplicated. It immediately implies that any rewriting step can be implemented in time polynomial in the size $|t|$ of t. A first consequence is the fact that $|u|$ is linear in the size of the starting term and the number of steps, that we call the *no size explosion* property.

These properties are based on a technical lemma relying on the notions of box context and box subterm, where a *box* is the argument of an application or the content of an explicit substitution, corresponding to explicit boxes for promotions in the proof nets representation of λ-terms with ES.

Definition 6.2 (Box Context, Box Subterm)**.** *Let t be a term.* ***Box contexts*** *(that are not necessarily shallow) are defined by the following grammar, where C is an generic context:*

$$B \quad ::= \quad t\langle\cdot\rangle \;\bigm|\; t[x{\leftarrow}\langle\cdot\rangle] \;\bigm|\; C\langle B\rangle.$$

A ***box subterm*** *of t is a term u s.t. $t = B\langle u\rangle$ for some box context B.*

We are now ready for the lemma stating the fundamental invariant of standard derivations.

Lemma 6.3 (Standard Derivations Preserve Boxes on Their Right)**.** *Let $\rho : t_0 \to^k t_k \to t_{k+1}$ be a standard derivation and let S be the position of the last contracted redex, $k \geq 0$, and $B \prec_p t_{k+1}$ be a box context s.t. $S \prec_{LO} B$. Then the box subterm u identified by B (*i.e. *s.t. $t_{k+1} = B\langle u\rangle$) is a box subterm of t_0.*

From the invariant, one easily obtains the subterm property, that in turn implies the no size explosion and the trace properties.

Corollary 6.4. *Let $\rho : t \to^k u$ be a standard derivation.*

1. Subterm*: every $\to_{\mathtt{ls}}$-step in ρ duplicates a subterm of t.*

2. No Size Explosion*:* $|u| \leq (k+1) \cdot |t|$.
3. Trace*: if t is an ordinary λ-term then $|u|_{[\cdot]} = |\rho|_{\mathtt{dB}}$.*

The subterm property of standard derivations is specific to evaluation in the LSC, and it is the crucial half of the notion of mechanisable strategy. It allows to see the standardisation theorem as the unveiling of a *very* abstract machine, hidden inside the calculus itself.

Let us conclude the section with a further invariant of standard derivations. It is not needed for the invariance result, but it sheds some light on the shallow subsystem under study. Let a term be **shallow** if its substitutions do not contain substitutions. The invariant is that if the initial term is a λ-term then standard shallow derivations involve only shallow terms. This fact is the only point of this section relying on the assumption that reduction is shallow (the standard hypothesis is also necessary, consider $(\lambda x.x)((\lambda y.y)z) \to_{\mathtt{dB}} (\lambda x.x)(y[y{\leftarrow}z]) \to_{\mathtt{dB}} x[x{\leftarrow}y[y{\leftarrow}z]]$).

Lemma 6.5 (Shallow Invariant)**.** *Let t be a λ-term and $\rho : t \to^k u$ be a standard derivation. Then u is a shallow term.*

7. The Subterm and Trace Properties, via Standard Derivations

While LO derivations are evidently standard, a priori LOU derivations may not be standard, if the reduction of a useful redex R could turn a useless redex $Q \prec_{LO} R$ into a useful redex. Luckily, this is not possible, *i.e.* uselessness is stable by reduction of $\prec_{LO}$-majorants, as proved by the next lemma.

Lemma 7.1 (Useless Persistence)**.** *Let $R : t \to_{\mathtt{ls}} u$ be a useless redex and $Q : t \to r$ be a useful redex s.t. $R \prec_{LO} Q$. The unique residual R' of R after Q is shallow and useless.*

Using the lemma above and a technical property of standard derivations (the *enclave axiom*, see [6]) we obtain:

Proposition 7.2 (LOU-Derivations Are Standard)**.** *Let ρ be a LOU derivation. Then ρ is a standard derivation.*

We conclude applying Corollary 6.4:

Corollary 7.3 (Subterm and Trace)**.** *LOU derivations have the subterm and the trace properties, and only involve shallow terms.*

8. The Normal Form and Projection Properties

For the normal form property it is necessary to show that the position of a redex in an unfolded term $t\downarrow$ can be traced back to the position of a useful redex in the original term t. Such a property requires a very detailed and technical study of unfoldings and position, and it is thus omitted (see [2]).

By induction on t and using the omitted property we obtain:

Proposition 8.1 (Normal Form)**.** *Let t be a LSC term in useful normal form. Then $t\downarrow$ is a β-normal form.*

The next lemma shows that useful reductions match their intended semantics, in the sense that every useful redex contributes somehow to a β-redex. It is not needed for the invariance result.

Lemma 8.2 (Inverse Normal Form)**.** *Let t be a LSC term s.t. $t\downarrow$ is a β-normal form. Then t is a useful normal form.*

For the projection property, we first show that the LO order is stable by unfolding, that in turn requires to show that it is stable by substitution. By induction on t:

Lemma 8.3 ($\prec_{LO}$ and Substitution)**.** *Let t be a λ-term, $S \prec_p t$ and $P \prec_p t$. If $S\{x{\leftarrow}u\} \prec_{LO} P\{x{\leftarrow}u\}$ then $S \prec_{LO} P$.*

Lemma 8.4 ($\prec_{LO}$ and Unfolding)**.** *Let t be a LSC term, $S \prec_p t$ and $P \prec_p t$. If $S\downarrow \prec_{LO} P\downarrow$ then $S \prec_{LO} P$.*

The next lemma deals with the hard part of the projection property. We use $\mapsto_\beta$ for β-reduction at top level.

Lemma 8.5 (LOU $\mathtt{dB}$-Step Projects on $\to_{LO\beta}$)**.** *Let t be a LSC term and $R : t = S\langle r\rangle \to_{\mathtt{dB}} S\langle p\rangle = u$ with $r \mapsto_{\mathtt{dB}} p$. Then:*
1. Projection*: $R\downarrow : t\downarrow = S\downarrow\langle r\downarrow_S\rangle \to_\beta S\downarrow\langle p\downarrow_S\rangle = u\downarrow$ with $r\downarrow_S \mapsto_\beta p\downarrow_S$;*
2. Minimality*: if moreover R is the LOU redex in t then $R\downarrow$ is the LO β-redex in $t\downarrow$.*

The first point is an ordinary projection of reductions. The second one is instead involved, as it requires to prove that if $R\downarrow$ is not LO then R is not LOU, *i.e.* to be able to somehow trace LO redexes back through unfoldings. The proof is by induction on S, that by hypothesis is the position of the LOU redex. The difficult case — not surprisingly — is when $S = P[x{\leftarrow}p]$, and where Lemma 8.3 is applied. The proof also uses the normal form property, when the position S is on the argument p of an application rp. Since R is LOU, r is useful-normal. To prove that $R\downarrow$ is the LO β redex in $(rp)\downarrow = r\downarrow p\downarrow$ we use the fact that $r\downarrow$ is normal.

Projection of derivations now follows as an easy induction:

Theorem 8.6 (Projection)**.** *Let t be a LSC term and $\rho : t \to^*_{LOU} u$. Then there is a LO β-derivation $\rho\downarrow : t\downarrow \to^*_\beta u\downarrow$ s.t. $|\rho\downarrow| = |\rho|_{\mathtt{dB}}$.*

9. The Syntactic Bound Property, via Nested Derivations

In this section we show that LOU derivations have the syntactic bound property. Instead of proving this fact directly, we introduce an abstract property, the notion of *nested derivation* and then prove that 1) nested derivations ensure the syntactic bound property, and 2) LOU derivations are nested. Such an approach helps to understand both LOU derivations and the syntactic bound property.

Definition 9.1 (Nested Derivation)**.** *Two $\mathtt{ls}$-steps $t \to_{\mathtt{ls}} u \to_{\mathtt{ls}} r$ are **nested** if the second one substitutes on the subterm substituted by the first one,* i.e. *if exist S and P s.t. the two steps have the compact form $S\langle x\rangle \to_{\mathtt{ls}} S\langle P\langle y\rangle\rangle \to_{\mathtt{ls}} S\langle P\langle u\rangle\rangle$. A derivation is nested if any two consecutive substitution steps are nested.*

For instance, the first of the following two sequences of steps is nested while the second is not:

$$\begin{aligned}(xy)[x{\leftarrow}yt][y{\leftarrow}u] &\to_{\mathtt{ls}} ((yt)y)[x{\leftarrow}yt][y{\leftarrow}u]\\ &\to_{\mathtt{ls}} ((ut)y)[x{\leftarrow}yt][y{\leftarrow}u];\\ (xy)[x{\leftarrow}yt][y{\leftarrow}u] &\to_{\mathtt{ls}} ((yt)y)[x{\leftarrow}yt][y{\leftarrow}u]\\ &\to_{\mathtt{ls}} ((yt)u)[x{\leftarrow}yt][y{\leftarrow}u].\end{aligned}$$

The idea is that nested derivations ensure the syntactic bound property because no substitution can be used twice in a nested sequence $u \to^k_{\mathtt{ls}} r$, and so k is necessarily bounded by $|u|_{[\cdot]}$.

Lemma 9.2 (Nested + Subterm = Syntactic Bound)**.** *Let t be a λ-term, $\rho : t \to^n u \to^k_{\mathtt{ls}} r$ be a derivation having the subterm property and whose suffix $u \to^k_{\mathtt{ls}} r$ is nested. Then $k \leq |u|_{[\cdot]}$.*

We are left to show that our small-step implementation of β — LOU derivations — indeed are nested derivations with the subterm property. We already know that they have the subterm property (Corollary 7.3), so we only need to show that they are nested. Using an omitted technical lemma, a case analysis on why a given substitution step is LOU proves:

Proposition 9.3. *LOU derivations are nested, and so they have the syntactic bound property.*

At this point, we proved all the abstract properties implying the high-level implementation theorem.

10. The Selection Property, or Computing Functions in Compact Form

This section proves the selection property for LOU derivations, which is the missing half of the proof that they are mechanisable, *i.e.* that they enjoy the low-level implementation theorem. The proof consists in providing a polynomial algorithm for testing the usefulness of a substitution step. The subtlety is that the test has to check whether a term in the form $t{\downarrow_S}$ contains a β-redex, or whether it is an abstraction, without explicitly computing $t{\downarrow_S}$ (which, of course, takes exponential time in the worst case). If one does not prove that this can be done in time polynomial in (the size of) t and S, then firing *each* reduction step can cause an exponential blowup!

Our algorithm consists in the simultaneous computation of 4 correlated functions on terms in compact form, two of which will provide the answer to our problem. We need some abstract preliminaries about computing functions in compact form.

A function f from n-uples of λ-terms to a set A is said to have *arity* n, and we write $f : n \to A$ in this case. The function f is said to be:

- *Efficiently computable* if there is a polynomial time algorithm $\mathcal{A}$ such that for every n-uple of λ-terms $(t_1, \ldots, t_n)$, the result of $\mathcal{A}(t_1, \ldots, t_n)$ is precisely $f(t_1, \ldots, t_n)$.
- *Efficiently computable in compact form* if there is a polynomial time algorithm $\mathcal{A}$ such that for every n-uple of LSC terms $(t_1, \ldots, t_n)$, the result of $\mathcal{A}(t_1, \ldots, t_n)$ is precisely $f(t_1{\downarrow}, \ldots, t_n{\downarrow})$.
- *Efficiently computable in compact form relatively to a context* if there is a polynomial time algorithm $\mathcal{A}$ such that for every n-uple of pairs of LSC terms and contexts $((t_1, S_1), \ldots, (t_n, S_n))$, the result of $\mathcal{A}((t_1, S_1), \ldots, (t_n, S_n)))$ is precisely $f(t_1{\downarrow_{S_1}}, \ldots, t_n{\downarrow_{S_n}})$.

An example of function is $alpha : 2 \to \mathbb{B}$, which given two λ-terms t and u, returns true if t and u are α-equivalent and false otherwise. In [3], $alpha$ is shown to be efficiently computable in compact form, via a dynamic programming algorithm $\mathcal{B}_{alpha}$ taking in input two LSC terms and computing, for every pair of their subterms, whether the (unfoldings) are α-equivalent or not. Proceeding bottom-up, as usual in dynamic programming, allows to avoid the costly task of computing unfoldings explicitly, which takes exponential time in the worst-case. More details about $\mathcal{B}_{alpha}$ can be found in [3].

Each one of the functions of our interest take values in one of the following sets:

$$\begin{aligned} \mathcal{VARS} &= \text{ the set of finite sets of variables} \\ \mathbb{B} &= \{\mathsf{true}, \mathsf{false}\} \\ \mathbb{T} &= \{\mathsf{var}(x) \mid x \text{ is a variable}\} \cup \{\mathsf{lam}, \mathsf{app}\} \end{aligned}$$

Elements of $\mathbb{T}$ represent the *nature* of a term. The functions are:

- $nature : 1 \to \mathbb{T}$, which returns the nature of the input term;
- $redex : 1 \to \mathbb{B}$, which returns true if the input term contains a redex and false otherwise;
- $apvars : 1 \to \mathcal{VARS}$, which returns the set of variables occurring in applicative position in the input term;
- $freevars : 1 \to \mathcal{VARS}$, which returns the set of free variables occurring in the input term.

Note that they all have arity 1 and that showing $redex$ and $nature$ to be *efficiently computable in compact form relatively to a context* is precisely what is required to prove the efficiency of useful reduction.

The four functions above can all be proved to be efficiently computable (in the three meanings). It is convenient to do so by giving an algorithm computing the product function $nature \times redex \times apvars \times freevars : 1 \to \mathbb{T} \times \mathbb{B} \times \mathcal{VARS} \times \mathcal{VARS}$ (which we call g) compositionally, on the structure of the input term, because the four function are correlated (for example, tu has a redex, *i.e.* $redex(tu) = \mathsf{true}$, if t is an abstraction, *i.e.* if $nature(t) = \mathsf{lam}$). The algorithm computing g on terms is $\mathcal{A}_g$ and is defined in Figure 1.

The interesting case in the algorithms for the two compact cases is the one for ES, that makes use of a special notation: given two sets of variables V, W and a variable x, $V \Downarrow_{x,W}$ is defined to be V if $x \in W$ and the empty set $\emptyset$ otherwise. The algorithm $\mathcal{B}_g$ computing g on LSC terms is defined in Figure 2. The algorithm computing g on pairs in the form (t, S) (where t is a LSC term and S is a shallow context) is defined in Figure 3.

First of all, we need to convince ourselves about the *correctness* of the proposed algorithms: do they really compute the function g? Actually, the way the algorithms are defined, namely by primitive recursion on the input terms, helps very much here: a simple induction suffices to prove the following:

Proposition 10.1. *The algorithms $\mathcal{A}_g$,$\mathcal{B}_g$,$\mathcal{C}_g$ are all correct, namely for every λ-term t, for every term u and for every context S, it holds that*

$$\mathcal{A}_g(t) = g(t); \qquad \mathcal{B}_g(u) = g(u{\downarrow}); \qquad \mathcal{C}_g(u, S) = g(u{\downarrow_S}).$$

The way the algorithms above have been defined also helps while proving that they work in bounded time, e.g., the number of recursive calls triggered by $\mathcal{A}_g(t)$ is linear in $|t|$ and each of them takes polynomial time. As a consequence, we can also easily bound the complexity of the three algorithms at hand.

Proposition 10.2. *The algorithms $\mathcal{A}_g$,$\mathcal{B}_g$,$\mathcal{C}_g$ all work in polynomial time. Thus LOU derivations are mechanisable.*

11. Summing Up

The various ingredients from the previous sections can be combined together so as to obtain the following result:

Theorem 11.1 (Invariance)**.** *There is an algorithm which takes in input a λ-term t and which, in time polynomial in $\#_{\to_{LO\beta}}(t)$ and $|t|$, outputs an LSC term u such that $u{\downarrow}$ is the normal form of t.*

As we have already mentioned, the algorithm witnessing the invariance of λ-calculus does *not* produce in output a λ-term, but a compact representation in the form of a term with ES. Theorem 11.1, together with the fact that equality of terms can be checked efficiently *in compact form*, entail the following formulation of invariance, akin in spirit to, *e.g.*, Statman's Theorem [26]:

Corollary 11.2. *There is an algorithm which takes in input two λ-terms t and u and checks whether t and u have the same normal form in time polynomial in $\#_{\to_{LO\beta}}(t)$, $\#_{\to_{LO\beta}}(u)$, $|t|$, and $|u|$.*

If one instantiates Corollary 11.2 to the case in which u is a normal form, one obtains that checking whether the normal form of any term t is equal to u can be done in time polynomial in $\#_{\to_{LO\beta}}(t)$, $|t|$, and $|u|$. This is particularly relevant when the size of u is constant, *e.g.*, when the λ-calculus computes decision problems and the relevant results are truth values.

Please observe that whenever one (or both) of the involved terms are *not* normalisable, the algorithms above (correctly) diverge.

12. Discussion

Here we further discuss invariance and some potential optimisations, that, however, are outside the scope of this work (which only deals with asymptotical bounds and is thus foundational in spirit).

$$\begin{aligned}
\mathcal{A}_g(x) &= (\mathsf{var}(x), \mathsf{false}, \emptyset, \{x\});\\
\mathcal{A}_g(\lambda x.t) &= (\mathsf{lam}, b_t, V_t - \{x\}, W_t - \{x\})\\
&\quad \text{where } \mathcal{A}_g(t) = (n_t, b_t, V_t, W_t);\\
\mathcal{A}_g(tu) &= (\mathsf{app}, b_t \vee b_u \vee (n_t = \mathsf{lam}), V_t \cup V_u \cup \{x \mid n_t = \mathsf{var}(x)\}, W_t \cup W_u)\\
&\quad \text{where } \mathcal{A}_g(t) = (n_t, b_t, V_t, W_t) \text{ and } \mathcal{A}_g(u) = (n_u, b_u, V_u, W_u);
\end{aligned}$$

Figure 1. Computing g in explicit form.

$$\begin{aligned}
\mathcal{B}_g(x) &= (\mathsf{var}(x), \mathsf{false}, \emptyset, \{x\});\\
\mathcal{B}_g(\lambda x.t) &= (\mathsf{lam}, b_t, V_t - \{x\}, W_t - \{x\})\\
&\quad \text{where } \mathcal{B}_g(t) = (n_t, b_t, V_t, W_t);\\
\mathcal{B}_g(tu) &= (\mathsf{app}, b_t \vee b_u \vee (n_t = \mathsf{lam}), V_t \cup V_u \cup \{x \mid n_t = \mathsf{var}(x)\}, W_t \cup W_u)\\
&\quad \text{where } \mathcal{B}_g(t) = (n_t, b_t, V_t, W_t) \text{ and } \mathcal{B}_g(u) = (n_u, b_u, V_u, W_u);\\
\mathcal{B}_g(t[x\leftarrow u]) &= (n, b, V, W)\\
&\quad \text{where } \mathcal{B}_g(t) = (n_t, b_t, V_t, W_t) \text{ and } \mathcal{B}_g(u) = (n_u, b_u, V_u, W_u) \text{ and:}\\
&\qquad n_t = \mathsf{var}(x) \Rightarrow n = n_u; \quad n_t = \mathsf{var}(y) \Rightarrow n = \mathsf{var}(y); \quad n_t = \mathsf{lam} \Rightarrow n = \mathsf{lam}; \quad n_t = \mathsf{app} \Rightarrow n = \mathsf{app};\\
&\qquad b = b_t \vee (b_u \wedge x \in W_t) \vee ((n_u = \mathsf{lam}) \wedge (x \in V_u));\\
&\qquad V = (V_t - \{x\}) \cup V_u \Downarrow_{x,W_t} \cup \{y \mid n_u = \mathsf{var}(y) \wedge x \in V_t\};\\
&\qquad W = (W_t - \{x\}) \cup W_u \Downarrow_{x,W_t}
\end{aligned}$$

Figure 2. Computing g in implicit form.

$$\begin{aligned}
\mathcal{C}_g(t, \langle\cdot\rangle) &= \mathcal{B}_g(t);\\
\mathcal{C}_g(t, \lambda x.S) &= \mathcal{C}_g(t, S);\\
\mathcal{C}_g(t, Su) &= \mathcal{C}_g(t, S);\\
\mathcal{C}_g(t, uS) &= \mathcal{C}_g(t, S);\\
\mathcal{C}_g(t, S[x\leftarrow u]) &= (n, b, V, W)\\
&\quad \text{where } \mathcal{C}_g(t, S) = (n_{t,S}, b_{t,S}, V_{t,S}, W_{t,S}) \text{ and } \mathcal{B}_g(u) = (n_u, b_u, V_u, W_u) \text{ and:}\\
&\qquad n_t = \mathsf{var}(x) \Rightarrow n = n_u; \quad n_t = \mathsf{var}(y) \Rightarrow n = \mathsf{var}(y); \quad n_t = \mathsf{lam} \Rightarrow n = \mathsf{lam}; \quad n_t = \mathsf{app} \Rightarrow n = \mathsf{app};\\
&\qquad b = b_t \vee (b_u \wedge x \in W_{t,S}) \vee ((n_u = \mathsf{lam}) \wedge (x \in V_u));\\
&\qquad V = (V_{t,S} - \{x\}) \cup V_u \Downarrow_{x,W_{t,S}} \cup \{y \mid n_u = \mathsf{var}(y) \wedge x \in V_t\};\\
&\qquad W = (W_{t,S} - \{x\}) \cup W_u \Downarrow_{x,W_{t,S}}
\end{aligned}$$

Figure 3. Computing g in implicit form, relative to a context

Mechanisability vs Efficiency. Let us stress that the study of invariance is about *mechanisability* rather than *efficiency*. One is not looking for the smartest or shortest evaluation strategy. But rather, for one that does not hide the complexity of its implementation in the cleverness of its definition, as it is the case for Lévy's optimal evaluation. Indeed, an optimal derivation can be even shorter then the shortest sequential strategy, but—as shown by Asperti and Mairson [7]—its definition hides hyper-exponential computations, and consequently optimal derivations do not provide an invariant cost model. The leftmost-outermost strategy, is a sort of *maximally unshared* normalising strategy, where redexes are duplicated whenever possible (and unneeded redexes are never reduced), somehow dually with respect to optimal derivations. It is exactly this *inefficiency* that induces the subterm property, the key point for its mechanisability. It is important to not confuse two different levels of sharing: our LOU derivations share *subterms*, but not *computations*, while Lévy's optimal derivations do the opposite. By sharing computations, they collapse the complexity of many steps into a single one, making the number of steps an unreliable measure.

Call-by-Value and Call-by-Need. Call-by-name evaluation is in many cases less efficient than call-by-value or call-by-need evaluation. Since we follow the call-by-name policy, the same kind of inefficiency shows up here. However, as already said, invariance is not about absolute efficiency: call-by-name and call-by-value are incomparable — sometimes the former can even be exponentially faster than the latter, sometimes the other way around—but this fact does not forbid both to be invariant, *i.e.* reasonably mechanisable.

We did not prove call-by-value/need invariance. Nonetheless, we strove to provide an abstract view of both the problem and of the architecture of our solution, having already in mind the adaptation to call-by-value/need λ-calculi. Recently, the first author and Sacerdoti Coen show [4] that (in the much simpler weak case) these policies provide an improved high-level implementation theorem, where evaluation in the LSC has a *linear* overhead, rather than quadratic.

Usefulness. Another source of inefficiency is the fact that at each reduction step we need to check whether the LO redex is useful before firing it, and this potentially amounts to doing a global analysis of the term. One could imagine decorating terms with additional tags in such a way that the check for usefulness becomes local *and* updating tags is not too costly, so that useful reduction may be implemented more efficiently. In particular, building on the already established relationships between the LSC and abstract machines [5], we expect to be able to design an abstract machine implementing LOU evaluation and testing for usefulness in time linear in the size of the starting term.

13. Conclusions

This work is the last tale in the long quest for an invariant cost model for the λ-calculus. In the last ten years, the authors have been involved in various works in which *parsimonious* time cost models have been shown to be invariant for more and more general notions of reduction, progressively relaxing the conditions on the use of sharing [3, 13, 14]. None of the results in the literature, however, concerns reduction to normal form as instead we do here.

By means of explicit substitutions—our tool for sharing—we provided the first full answer to this long-standing open problem: we proved that the λ-calculus is indeed a reasonable machine, by showing that the length of the leftmost-outermost derivation to normal form is an invariant cost model.

The solution required the development of a whole new toolbox: an abstract deconstruction of the problem, a detailed study of unfoldings, a theory of useful derivations, and a general view of functions efficiently computable in compact form. Along the way, we showed that standard derivations for explicit substitutions enjoy the crucial *subterm property*. Essentially, it ensures that standard derivations are mechanisable, unveiling a very abstract notion of machine hidden deep inside the λ-calculus itself, and also a surprising perspective on the standardisation theorem, a classic result apparently unrelated to the complexity of evaluation.

Among the downfalls of our results, one can of course mention that proving systems to characterise time complexity classes equal or larger than $\mathbf{P}$ can now be done merely by deriving bounds on the *number* of leftmost-outermost reduction steps to normal form. This could be useful, e.g., in the context of *light logics* [9, 11, 17]. The kind of bounds we obtain here are however more *general* than those obtained in implicit computational complexity (since we deal with a universal model of computation).

While there is room for finer analyses (*e.g.* studying call-by-value or call-by-need evaluation), we consider the understanding of time invariance essentially achieved. However, the study of complexity measures for λ-terms is far from being over. Indeed, the study of space complexity for functional programs has only made its very first steps [15, 18, 24], and not much is known about invariant *space* cost models.

Acknowledgments

The second author is supported by the project ANR-12IS02001 "PACE".

References

[1] B. Accattoli. An abstract factorization theorem for explicit substitutions. In *RTA*, pages 6–21, 2012.

[2] B. Accattoli and U. Dal Lago. Beta reduction is invariant, indeed (long version). Available at `http://eternal.cs.unibo.it/brii.pdf`.

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

[4] B. Accattoli and C. Sacerdoti Coen. On the Value of Variables. Accepted to WoLLIC 2014, 2014.

[5] B. Accattoli, P. Barenbaum, and D. Mazza. Distilling Abstract Machines. Accepted to ICFP 2014, 2014.

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

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

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

[9] P. Baillot and K. Terui. Light types for polynomial time computation in lambda calculus. *Inf. Comput.*, 207(1):41–62, 2009.

[10] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103. North-Holland, 1984.

[11] P. Coppola, U. Dal Lago, and S. Ronchi Della Rocca. Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science*, 4(4), 2008.

[12] H. Curry and R. Feys. *Combinatory Logic*. Studies in logic and the foundations of mathematics. North-Holland Publishing Company, 1958.

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

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

[15] U. Dal Lago and U. Schöpp. Functional programming in sublinear space. In *ESOP*, pages 205–225, 2010.

[16] N. G. de Bruijn. Generalizing Automath by Means of a Lambda-Typed Lambda Calculus. In *Mathematical Logic and Theoretical Computer Science*, number 106 in Lecture Notes in Pure and Applied Mathematics, pages 71–92. Marcel Dekker, 1987.

[17] M. Gaboardi and S. Ronchi Della Rocca. A soft type assignment system for λ-calculus. In *CSL*, pages 253–267, 2007.

[18] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In *POPL*, pages 121–131, 2008.

[19] 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.

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

[21] R. Milner. Local bigraphs and confluence: Two conjectures. *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007.

[22] R. P. Nederpelt. The fine-structure of lambda calculus. Technical Report CSN 92/07, Eindhoven Univ. of Technology, 1992.

[23] S. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[24] U. Schöpp. Stratified Bounded Affine Logic for Logarithmic Space. In *LICS*, pages 411–420, 2007.

[25] 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.

[26] R. Statman. The typed lambda-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9:73–81, 1979.

[27] C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. PhD Thesis, Oxford, 1971. Chapter 4.