HAL
archives-ouvertes.fr

(Leftmost-outermost) beta reduction is invariant, indeed

Beniamino Accattoli, Ugo Dal Lago

▶ **To cite this version:**

Beniamino Accattoli, Ugo Dal Lago. (Leftmost-outermost) beta reduction is invariant, indeed. Logical Methods in Computer Science, Logical Methods in Computer Science Association, 2016, <10.2168/LMCS-12(1:4)2016>. <hal-01337712>

HAL Id: hal-01337712
https://hal.inria.fr/hal-01337712

Submitted on 27 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Logical Methods in Computer Science
Vol. 12(1:4)2016, pp. 1–46
www.lmcs-online.org

Submitted Feb. 20, 2015
Published Mar. 9, 2016

(LEFTMOST-OUTERMOST) BETA REDUCTION IS INVARIANT, INDEED

BENIAMINO ACCATTOLI [a] AND UGO DAL LAGO [b]

[a] INRIA & LIX École Polytechnique
e-mail address: beniamino.accattoli@inria.fr

[b] Dipartimento di Informatica - Scienza e Ingegneria Università degli Studi di Bologna
e-mail address: dallago@cs.unibo.it

Abstract. Slot and van Emde Boas' weak invariance thesis states that *reasonable* machines can simulate each other within a polynomial 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 on a standard notion in the theory of λ-calculus: the length of a leftmost-outermost derivation to normal form is an invariant, *i.e.* reasonable, 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 large 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 modeled 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 the 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, called *useful*. Useful evaluation produces a compact, shared representation of the normal form, by avoiding 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.

2012 ACM CCS: [**Theory of computation**]: Models of computation—Computability—Lambda calculus; Models of computation—Abstact machines; Logic—Proof theory / Linear logic / Equational logic and rewriting; Computational complexity and cryptography—Complexity theory and logic.

Key words and phrases: lambda calculus, cost models, linear logic, invariance, standardization, sharing, operational semantics.

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-12(1:4)2016

© B. Accattoli and U. Dal Lago
Creative Commons

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 resolved?

The tools for stating the question formally are provided by complexity theory and by Slot and van Emde Boas' invariance thesis [SvEB84], which stipulates when any Turing complete computational model can be considered reasonable:

> *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 (alternatively called *extended*, *efficient*, *modern*, or *complexity-theoretic* Church(-Turing) thesis). The idea behind the thesis is that for reasonable models the definition of every polynomial or super-polynomial class such as **P** or **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. A first refinement of our question then is: are functional languages invariant with respect to standard models like random access machines or Turing machines?

Invariance results have to be proved via an appropriate measure of time complexity for programs, *i.e.* a *cost model*. The natural measure for functional languages is the *unitary* cost model, *i.e.* the number of evaluation steps. There is, however, a subtlety. The evaluation of functional programs, in fact, depends very much on the evaluation strategy chosen to implement the language, while the reference model for functional languages, the λ-calculus, 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 [Lév78]) 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 intrinsic complexity is non-elementary [LM96, AM98]. 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).

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 the 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 subterms 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 answer is positive, at least for certain restricted forms of reduction: the number of steps is already known to be an invariant cost model for weak reduction [BG95, SGM02, DLM08, DLM12] and for head reduction [ADL12].

If the problem at hand consists in computing the *normal form* of an arbitrary λ-term, however, no positive answer was known, to the best of our knowledge, before our result. 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 usually reduced to the computation of normal forms.

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 standardization theorem, one of the central theorems in the theory of λ-calculus, first proved by Curry and Feys [CF58]. 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. Moreover, the underlying computational model is very far from traditional, machine-based models like Turing machines and RAMs.

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, as actual interpreters perform copying in a very controlled way (see, *e.g.*, [Wad71, PJ87]). 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 modeling tool? In this paper we give a very precise answer, formulated within rewriting theory itself. A second contribution of the paper, indeed, is a rewriting analysis of the technique used to prove the invariance result.

As in our previous work [ADL12], we prove our result by means of the *linear substitution calculus* (see also [Acc12, ABKL14]), a simple calculus of explicit substitutions (ES, for short) introduced by Accattoli and Kesner, that arises from linear logic and graphical syntaxes and it is similar to calculi studied by de Bruijn [dB87], Nederpelt [Ned92], and Milner [Mil07]. 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 standardization and a notion of LO evaluation [ABKL14]. The proof of our result is 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.

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 [ADL12]: there is an encoding of Turing machines into λ-terms such that their execution is simulated by weak head β-reduction with only a linear overhead. The result on tree Turing machines from [FS91] is not immediately applicable here, being formulated on a different, more parsimonious, cost model.

On Invariance, Complexity Analysis, and Some Technical Choices. 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 **P** or **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 analyze. 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.
(4) *LSC vs. Graph-Reduction vs. Abstract Machines.* The LSC has been designed as a graph-free formulation of the representation of λ-calculus into linear logic proof nets. As such, it can be seen as equivalent to a graph-rewriting formalism. While employing graphs may slightly help in presenting some intuitions, terms are much simpler to define, manipulate, and formally reason about. In particular the detailed technical development we provide would simply be out of scope if we were using a graphical formalism. Abstract machines are yet another formalism that could have been employed, that also has a tight relationship with the LSC, as shown by Accattoli, Berenbaum, and Mazza in [ABM14]. We chose to work with the LSC because it is more abstract than abstract machines and both more apt to formal reasoning and closer to the λ-calculus (no translation is required) than graph-rewriting.
(5) *Proof Strategy.* While the main focus of the paper is the invariance result, we also spend much time providing an abstract decomposition of the problem and a more general rewriting analysis of the LSC and of useful sharing. Therefore, the proof presented here is not the simplest possible one. We believe, however, that our study is considerably more informative than the shortest proof.

The next section explains why the problem at hand is hard, and in particular why iterating our previous results on head reduction [ADL12] does not provide a solution.

Related Work. In the literature invariance results for the weak call-by-value λ-calculus have been proved three times, independently. First, by Blelloch and Greiner [BG95], while studying cost models for parallel evaluation. Then by Sands, Gustavsson and Moran [SGM02], while studying speedups for functional languages, and finally by Martini and the second author [DLM08], who addressed the invariance thesis for the λ-calculus. The latter also proved invariance for the weak call-by-name λ-calculus [DLM12]. Invariance of head reduction has been shown by the present authors, in previous work [ADL12]. The problem of an invariant cost model for the ordinary λ-calculus is discussed by Frandsen and Sturtivant [FS91], and then by Lawall and Mairson [LM96]. Frandsen and Sturtivant's proposal consists in taking the number of *parallel* β-steps to normal form as the cost of reducing any term. A negative result about the invariance of such cost model has been proved by Asperti and Mairson [AM98]. When only first order symbols are considered, Dal Lago and Martini, and independently Avanzini and Moser, proved some quite general results through graph rewriting [DLM12, AM10], itself a form of sharing.

This paper is a revised and extended version of [ADL14a], to which it adds explanations and the proofs that were omitted. It differs considerably with respect to both [ADL14a] and the associated technical report [ADL14b], as proofs and definitions have been improved and simplified, partially building on the recent work by Accattoli and Sacerdoti Coen in [ASC15], where useful sharing is studied in a call-by-value scenario.

After the introduction, in Sect. 1 we explain why the problem is hard by discussing the size-explosion problem. An abstract view of the solution is given in Sect. 7. The sections in between (2-6) provide the background, *i.e.* definitions and basic results, up to the introduction of useful reduction—at a first reading we suggest to skip them. After the abstract view, in Sect. 8 we explain how the various abstract requirements are actually proved in the remaining sections (9-14), where the proofs are. We put everything together in Sect. 15, and discuss optimizations in Sect. 16.

1. 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. The problem is *size-explosion*, or the existence of terms of size n that in $O(n)$ steps produce an output of size $O(2^n)$, and affects the λ-calculus as well as its weak and head variants. The explosion is due to iterated *useless* duplications of subterms that are normal and whose substitution does not create new redexes. For simple cases as weak or head reduction, turning to shared representations of λ-terms and micro-step substitutions (*i.e.* one occurrence at the time) is enough to avoid size-explosion. For micro-steps, in fact, the length of evaluation and the size of the output are linearly related. A key point is that both micro-step weak and head reduction stop on a compact representation of the weak or head normal form.

In the ordinary λ-calculus, a very natural notion of evaluation to normal form is LO reduction. Unfortunately, turning to sharing and micro-step LO evaluation is not enough, because such a micro-step simulation of β-reduction computes ordinary normal forms, *i.e.* it does not produce a compact representation, but the usual one, whose size is sometimes exponential. In other words, size-explosion reappears disguised as *length-explosion*: for the

size-exploding family, indeed, micro-step evaluation to normal form is necessarily exponential in n, because its length is linear in the size of the output. Thus, the number of β-steps cannot be shown to be invariant using such a simple form of sharing.

The problem is that evaluation should stop on a compact—*i.e.* not exponential—representation of the normal form, as in the simpler cases, but there is no such notion. Our way out is the definition of a variant of micro-step LO evaluation that stops on a minimal *useful normal form*, that is a term with ES t such that unfolding all the substitutions in t produces a normal form, *i.e.* such that the duplications left to do are useless. In Sect. 6, we will define *useful* reduction, that will stop on minimal useful normal forms and for which we will show invariance with respect to both β-reduction and random access machines.

In the rest of the section we discuss in detail the size-explosion problem, recall the solution for the head case, and explain the problem for the general case. Last, we discuss the role of standard derivations.

1.1. **A Size-Exploding Family.** The typical example of a term that is useless to duplicate is a free variable[1], as it is normal and its substitution cannot create redexes. Note that the same is true for the application xx of a free variable x to itself, and, iterating, for $(xx)(xx)$, and so on. We can easily build a term u_n of size $|u_n| = O(n)$ that takes a free variable x, and puts its self application xx as argument of a new redex, that does the same, *i.e.* it puts the self application $(xx)(xx)$ as argument of a new redex, and so on, for n times normalizing in n steps to a complete binary tree of height n and size $O(2^n)$, whose internal nodes are applications and whose 2^n leaves are all occurrences of x. Let us formalize this notion of *variable tree* $x^{@n}$ of height n:

$$\begin{array}{lcl} x^{@0} & := & x; \\ x^{@(n+1)} & := & x^{@n}x^{@n}. \end{array}$$

Clearly, the size of variable trees is exponential in n, a routine induction indeed shows $|x^{@n}| = 2^{n+1} - 1 = O(2^n)$. Now let us define the family of terms $\{t_n\}_{n\geq 1}$ that in only n LO steps blows up x into the tree $x^{@n}$:

$$\begin{array}{lcll} t_1 & := & \lambda y_1.(y_1 y_1) & = \lambda y_1.y_1^{@1}; \\ t_{n+1} & := & \lambda y_{n+1}.(t_n(y_{n+1}y_{n+1})) & = \lambda y_{n+1}.(t_n y_{n+1}^{@1}). \end{array}$$

Note that the size $|t_n|$ of t_n is $O(n)$. Leftmost-outermost β-reduction is noted $\to_{\mathrm{LO}\beta}$. The next proposition proves size-explosion, *i.e.* $t_n x = t_n x^{@0} \to^n_{\mathrm{LO}\beta} x^{@n}$ (with $|t_n x| = O(n)$ and $|x^{@n}| = O(2^n)$ giving the explosion). The statement is slightly generalized, in order to express it as a nice property over variable trees.

Proposition 1.1 (Size-Explosion)**.** $t_n x^{@m} \to^n_{\mathrm{LO}\beta} x^{@(n+m)}$.

Proof. By induction on n. Cases:
(1) *Base Case*, *i.e.* $n = 1$:

$$t_1 x^{@m} \quad = \quad (\lambda y_1.(y_1 y_1))x^{@m} \quad \to_{\mathrm{LO}\beta} \quad x^{@m}x^{@m} \quad = \quad x^{@(m+1)}.$$

[1] *On open terms*: in the λ-calculus free variables are unavoidable because reduction takes place under abstractions. Even if one considers only globally closed terms, variable occurrences may look free *locally*, as y in $\lambda y.((\lambda x.(xx))y) \to_\beta \lambda y.(yy)$. This is why for studying the strong λ-calculus it is common practice to work with possibly open terms.

(2) *Induction Step*:

$$\begin{array}{rcll} t_{n+1}x^{@m} & = & (\lambda y_{n+1}.(t_n(y_{n+1}y_{n+1})))x^{@m} & \to_{\mathrm{LO}\beta} \quad t_n(x^{@m}x^{@m}) \\ & & & = \quad t_n x^{@(m+1)} \\ & & & \to^n_{\mathrm{LO}\beta} \quad x^{@(n+m+1)} \qquad (i.h.). \end{array}$$ □

It seems that the unitary cost model—*i.e.* the number of LO β-steps—*is not* invariant: in a linear number of β-steps we reach an object which cannot even be written down in polynomial time.

1.2. **The Head Case.** The solution the authors proposed in [ADL12] tames size-explosion in a satisfactory way when *head* reduction is the evaluation strategy (note that β-steps in Proposition 1.1 are in particular head steps). It uses sharing under the form of explicit substitutions (ES), that amounts to extend the language with an additional constructor noted $t[x{\leftarrow}u]$, that is an avatar of `let`-expressions, to be thought of as a sharing annotation of u for x in t, or as a term notation for the DAGs used in the graph-rewriting of λ-calculus (see [AG09]). The usual, capture-avoiding, and meta-level notion of substitution is instead noted $t\{x{\leftarrow}u\}$.

Let us give a sketch of how ES work for the head case. Formal details about ES and the more general LO case will be given in the Sect. 4. First of all, a term with sharing, *i.e.* with ES, can always be unshared, or unfolded, obtaining an ordinary λ-term $t{\downarrow}$.

Definition 1.2 (Unfolding)**.** The *unfolding* $t{\downarrow}$ of a term with ES t is given by:

$$\begin{array}{rlrl} x{\downarrow} & := x; & (tu){\downarrow} & := t{\downarrow}u{\downarrow}; \\ (\lambda x.t){\downarrow} & := \lambda x.t{\downarrow}; & (t[x{\leftarrow}u]){\downarrow} & := t{\downarrow}\{x{\leftarrow}u{\downarrow}\}. \end{array}$$

Head β-reduction is β-reduction in a head context, *i.e.* out of all arguments, possibly under abstraction (and thus involving open terms). A head step $(\lambda x.t)u \to_\beta t\{x{\leftarrow}u\}$ is simulated by

(1) *Delaying Substitutions*: the substitution $\{x{\leftarrow}u\}$ is delayed with a rule $(\lambda x.t)u \to_{\mathtt{dB}} t[x{\leftarrow}u]$ that introduces an explicit substitution. The name `dB` stays for *distant* β or β *at a distance*, actually denoting a slightly more general rule to be discussed in the next section[2].

(2) *Linear Head Substitutions*: linear substitution $\to_{\mathtt{ls}}$ replaces a single variable occurrence with the associated shared subterm. Linear *head* substitution $\to_{\mathtt{lhs}}$ is the variant that replaces only the head variable, for instance $(x(yx))[x{\leftarrow}t][y{\leftarrow}u] \to_{\mathtt{lhs}} (t(yx))[x{\leftarrow}t][y{\leftarrow}u]$. Linear substitution can be seen as a reformulation with ES of de Bruijn's *local* β-*reduction* [dB87], and similarly its head variant is a reformulation of Danos and Regnier's presentation of linear head reduction [DR04].

In particular, the size-exploding family $t_n x$ is evaluated by the following *linear* head steps. For $n = 1$ we have

$$t_1 x \quad = \quad (\lambda y_1.(y_1y_1))x \quad \to_{\mathtt{dB}} \quad (y_1y_1)[y_1{\leftarrow}x] \quad \to_{\mathtt{lhs}} \quad (xy_1)[y_1{\leftarrow}x]$$

[2]A more accurate explanation of the terminology: in the literature on ES the rewriting rule $(\lambda x.t)u \to t[x{\leftarrow}u]$ (that is the explicit variant of β) is often called `B` to distinguish it from β, and `dB`—that will be formally defined in Sect. 4—stays for *distant* `B` (or `B` *at a distance*) rather than *distant* β.

Note that only the head variable has been replaced, and that evaluation requires one $\to_{\mathtt{dB}}$ step and one $\to_{\mathtt{lhs}}$ step. For $n = 2$,

$$\begin{array}{rll} t_2 x & = & (\lambda y_2.((\lambda y_1.(y_1 y_1))(y_2 y_2)))x \\ & \to_{\mathtt{dB}} & ((\lambda y_1.(y_1 y_1))(y_2 y_2))[y_2 \leftarrow x] \\ & \to_{\mathtt{dB}} & (y_1 y_1)[y_1 \leftarrow y_2 y_2][y_2 \leftarrow x] \\ & \to_{\mathtt{lhs}} & ((y_2 y_2) y_1)[y_1 \leftarrow y_2 y_2][y_2 \leftarrow x] \\ & \to_{\mathtt{lhs}} & ((x y_2) y_1)[y_1 \leftarrow y_2 y_2][y_2 \leftarrow x]. \end{array}$$

It is easily seen that

$$t_n x \to_{\mathtt{dB}}^{n} \to_{\mathtt{lhs}}^{n} ((\dots(x y_n)\dots y_2) y_1)[y_1 \leftarrow y_2 y_2][y_2 \leftarrow y_3 y_3]\dots[y_n \leftarrow x] \quad =: r_n$$

As one can easily verify, the size of the linear head normal form r_n is linear in n, so that there is no size-explosion (the number of steps is also linear in n). Moreover, the unfolding $r_n{\downarrow}$ of r_n is exactly $x^{@n}$, so that the linear head normal form r_n is a compact representation of the head normal form, *i.e.* the expected result. Morally, in r_n only the left branch of the complete binary tree $x^{@n}$ has been unfolded, while the rest of the tree is kept shared via explicit substitutions. Size-explosion is avoided by not substituting in arguments at all.

Invariance of head reduction via LHR is obtained in [ADL12] by proving that LHR correctly implements head reduction up to unfolding within—crucially—a quadratic overhead. This is how sharing is exploited to circumvent the size-explosion problem: the length of head derivations is a reasonable cost model even if head reduction suffers of size-explosion, because the actual implementation is meant to be done via LHR and be only polynomially (actually quadratically) longer. Note that—a posteriori—we are allowed to forget about ES. They are an essential tool for the proof of invariance. But once invariance is established, one can provide reasonable complexity bounds by simply counting β-steps in the λ-calculus, with no need to deal with ES.

Of course, one needs to show that turning to shared representations is a reasonable choice, *i.e.* that using a term with ES outside the evaluation process does not hide an exponential overhead. Shared terms can in fact be managed efficiently, typically tested for *equality of their unfoldings* in time polynomial (actually quadratic [ADL12], or quasi-linear [GR14]) in the size of the *shared terms*. In Sect. 14, we will discuss another kind of test on shared representations.

1.3. **Length-Explosion and Usefulness.** It is clear that the computation of the full normal form $x^{@n}$ of $t_n x$, requires exponential work, so that the general case seems to be hopeless. In fact, there is a notion of linear LO reduction $\to_{\mathrm{LO}}$ [ABKL14], obtained by iterating LHR on the arguments, that computes normal forms and it is linearly related to the size of the output. However, $\to_{\mathrm{LO}}$ cannot be polynomially related to the LO strategy $\to_{\mathrm{LO}\beta}$, because it produces an exponential output, and so it necessarily takes an exponential number of steps. In other words, size-explosion disguises itself as *length-explosion*. With respect to our example, $\to_{\mathrm{LO}}$ extends LHR evaluation by unfolding the whole variable tree in a LO way,

$$t_n x \to_{\mathtt{dB}}^{n} \to_{\mathtt{ls}}^{O(2^n)} x^{@n}[y_1 \leftarrow y_2 y_2][y_2 \leftarrow y_3 y_3]\dots[y_n \leftarrow x]$$

and leaving garbage $[y_1 \leftarrow y_2 y_2][y_2 \leftarrow y_3 y_3]\dots[y_n \leftarrow x]$ that may eventually be collected. Note the exponential number of steps.

Getting out of this *cul-de-sac* requires to avoid useless duplication. Essentially, only substitution steps that contribute to eventually obtain an unshared β-redex have to be done.

The other substitution steps, that only unfold parts of the normal form, have to be avoided. Such a process then produces a minimal shared term whose unfolding is an ordinary normal form. The tricky point is how to define, and then select in reasonable time, those steps that *contribute to eventually obtain an unshared β-redex*. The definition of useful reduction relies on tests of certain partial unfoldings that have a inherent global nature, what in a graphical formalism can be thought of as the unfolding of the sub-DAG rooted in a given sharing node. Of course, computing unfoldings takes in general exponential time, so that an efficient way of performing such tests has to be found.

The proper definition of useful reduction is postponed to Sect. 6, but we discuss here how it circumvents size-explosion. With respect to the example, useful reduction evaluates $t_n x$ to the useful normal form

$$(y_1 y_1)[y_1 \leftarrow y_2 y_2][y_2 \leftarrow y_3 y_3] \dots [y_n \leftarrow x],$$

that unfolds to the exponentially bigger result $x^{@n}$. In particular, our example of size-exploding family will be evaluated without performing *any duplication at all*, because the duplications needed to compute the normal form are all useless.

Defining and reasoning about useful reduction requires some care. At first sight, one may think that it is enough to evaluate a term t in a LO way, stopping as soon as a useful normal form is reached. Unfortunately, this simple approach does not work, because size-explosion may be caused by ES *lying in between* two β-redexes, so that LO evaluation would unfold the exploding substitutions anyway.

Moreover, it is not possible to simply define useless terms and avoid their reduction. The reason is that usefulness and uselessness are properties of substitution steps, not of subterms. Said differently, whether a subterm is useful depends crucially on the context in which it occurs. An apparently useless argument may become useful if plugged into the right context. Indeed, consider the term $u_n := (\lambda x.(t_n x))I$, obtained by plugging the size-exploding family in the context $(\lambda x.\langle\cdot\rangle)I$, that abstracts x and applies to the identity $I := \lambda z.z$. By delaying β-redexes we obtain:

$$u_n \to_{\mathtt{dB}}^{n+1} (y_1 y_1)[y_1 \leftarrow y_2 y_2][y_2 \leftarrow y_3 y_3] \dots [y_n \leftarrow x][x \leftarrow I]$$

Now—in contrast to the size-explosion case—it is useful to unfold the whole variable tree $x^{@n}$, because the obtained copies of x will be substituted by I, generating exponentially many β steps, that compensate the explosion in size. Our notion of *useful* step will elaborate on this idea, by computing *contextual* 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.

It is also worth mentioning that the contextual nature of useful substitution implies that—as a rewriting rule—it is inherently *global*: it cannot be first defined at top level (*i.e.* locally) and then extended via a closure by evaluation contexts, because the evaluation context has to be taken into account in the definition of the rule itself. Therefore, the study of useful reduction is delicate at the technical level, as proofs by naïve induction on evaluation contexts usually do not work.

1.4. **The Role of Standard Derivations.** Apart from the main result, we also connect the classic rewriting concept of standard derivation with the problem under study. Let us stress that such a connection is a *plus*, as it is not needed to prove the invariance theorem.

We use it in the proof, but only to shed a new light on a well-established rewriting concept, and not because it is necessary.

The role of standard derivations is in fact twofold. On the one hand, LO β-derivations are standard, and thus our invariant cost model is justified by a classic notion of evaluation, *internal* to the theory of the λ-calculus and not *ad-hoc*. On the other hand, the linear useful strategy is shown to be standard for the LSC. Therefore, this notion, at first defined *ad-hoc* to solve the problem, turns out to fit the theory.

The paper contains also a general result about standard derivations for the LSC. We show they have the *subterm property*, *i.e.* every single step of a standard derivation $\rho : t \to^* u$ is implementable in time linear in the size $|t|$ of the input. It follows that the size of the output is linear in the length of the derivation, and so there is no size-explosion. Such a connection between standardization and complexity analysis is quite surprising, and it is one of the signs that a new complexity-aware rewriting theory of β-reduction is emerging.

At a first reading, we suggest to read Sect. 7, where an abstract view of the solution is provided, right after this section. In between (*i.e.* sections 2-6), there is the necessary long sequence of preliminary definitions and results. In particular, Sect. 6, will define useful reduction.

2. Rewriting

For us, an *(abstract) reduction system* is a pair $(T, \to_{\mathcal{T}})$ consisting of a set T of terms and a binary relation $\to_{\mathcal{T}}$ on T called a *reduction (relation)*. When $(t, u) \in \to_{\mathcal{T}}$ we write $t \to_{\mathcal{T}} u$ and we say that t *$\mathcal{T}$-reduces* to u. The reflexive and transitive closure of $\to_{\mathcal{T}}$ is written $\to^*_{\mathcal{T}}$. Composition of relations is denoted by juxtaposition. Given $k \geq 0$, we write $a \to^k_{\mathcal{T}} b$ iff a is $\mathcal{T}$-related to b in k steps, *i.e.* $a \to^0_{\mathcal{T}} b$ if $a = b$ and $a \to^{k+1}_{\mathcal{T}} b$ if $\exists\, c$ such that $a \to_{\mathcal{T}} c$ and $c \to^k_{\mathcal{T}} b$.

A term $t \in R$ is a *$\mathcal{T}$-normal form* if there is no $u \in R$ such that $t \to_{\mathcal{T}} u$. Given a deterministic reduction system $(T, \to_{\mathcal{T}})$, and a term $t \in T$, the expression $\#_{\to_{\mathcal{T}}}(t)$ stands for the number of reduction steps necessary to reach the $\to_{\mathcal{T}}$-normal form of t along $\to_{\mathcal{T}}$, or ∞ if t diverges. Similarly, given a natural number n, the expression $\to^n_{\mathcal{T}}(t)$ stands for the term u such that $t \to^n_{\mathcal{T}} u$, if $n \leq \#_{\to_{\mathcal{T}}}(t)$, or for the normal form of t otherwise.

3. λ-Calculus

3.1. **Statics.** The syntax of the λ-calculus is given by the following grammar for terms:

$$t, u, r, p ::= x \;\mid\; \lambda x.t \;\mid\; tu.$$

We use $t\{x \leftarrow u\}$ for the usual (meta-level) notion of substitution. An abstraction $\lambda x.t$ binds x in t, and we silently work modulo α-equivalence of these bound variables, *e.g.* $(\lambda y.(xy))\{x \leftarrow y\} = \lambda z.(yz)$. We use $\mathtt{fv}(t)$ for the set of free variables of t.

Contexts. One-hole contexts are defined by:

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

and the plugging of a term t into a context C is defined by

$$\begin{array}{rclcrcl} \langle\cdot\rangle\langle t\rangle & := & t & \qquad & (Cu)\langle t\rangle & := & C\langle t\rangle u \\ (\lambda x.C)\langle t\rangle & := & \lambda x.C\langle t\rangle & & (uC)\langle t\rangle & := & uC\langle t\rangle \end{array}$$

As usual, plugging in a context can capture variables, *e.g.* $(\lambda y.(\langle\cdot\rangle y))\langle y\rangle = \lambda y.(yy)$. The plugging $C\langle D\rangle$ of a context D into a context C is defined analogously. Plugging will be implicitly extended to all notions of contexts in the paper, always in the expected way.

3.2. **Dynamics.** We define β-reduction $\to_\beta$ as follows:

Rule at Top Level	Contextual closure
$(\lambda x.t)u \mapsto_\beta t\{x \leftarrow u\}$	$C\langle t\rangle \to_\beta C\langle u\rangle$ if $t \mapsto_\beta u$

The *position* of a β-redex $C\langle t\rangle \to_\beta C\langle u\rangle$ is the context C in which it takes place. To ease the language, we will identify a redex with its position. A *derivation* $\rho : t \to^k u$ is a finite, possibly empty, sequence of reduction steps, sometimes given as $C_1; \ldots; C_k$, *i.e.* as the sequence of positions of reduced redexes. We write $|t|$ for the size of t and $|\rho|$ for the length of ρ.

Leftmost-Outermost Derivations. 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 left-to-right is a total order). While the study of standardization for the LSC [ABKL14] uses the string approach (and thus only talks about the *left-to-right* order and the *leftmost* redex), here some of the proofs 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 3.1.

(1) The *outside-in order*:
   (a) *Root*: $\langle\cdot\rangle \prec_O C$ for every context $C \neq \langle\cdot\rangle$;
   (b) *Contextual closure*: If $C \prec_O D$ then $E\langle C\rangle \prec_O E\langle D\rangle$ for any context E.
(2) The *left-to-right order*: $C \prec_L D$ is defined by:
   (a) *Application*: If $C \prec_p t$ and $D \prec_p u$ then $Cu \prec_L tD$;
   (b) *Contextual closure*: If $C \prec_L D$ then $E\langle C\rangle \prec_L E\langle D\rangle$ for any context E.
(3) 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,

$$(\lambda x.\langle\cdot\rangle)t \prec_O (\lambda x.(\langle\cdot\rangle u))t;$$
$$(\langle\cdot\rangle t)u \prec_L (rt)\langle\cdot\rangle.$$

Definition 3.2 (LO β-Reduction)**.** Let t be a λ-term and C a redex of t. C is the *leftmost-outermost* β-redex (LOβ for short) of t if $C \prec_{LO} D$ for every other β-redex D of t. We write $t \to_{\mathrm{LO}\beta} u$ if a step reduces the LO β-redex.

3.3. **Inductive LOβ Contexts.** It is useful to have an inductive characterization of the contexts in which $\to_{\mathrm{LO}\beta}$ takes place. We use the following terminology: a term is *neutral* if it is β-normal and it is not of the form $\lambda x.u$, *i.e.* it is not an abstraction.

Definition 3.3 (iLOβ Context)**.** Inductive LOβ (or iLOβ) contexts are defined by induction as follows:

$$\frac{}{\langle\cdot\rangle \text{ is iLO}\beta}\ (\text{ax-iLO}\beta) \qquad \frac{C \text{ is iLO}\beta \qquad C \neq \lambda x.D}{Ct \text{ is iLO}\beta}\ (@\text{l-iLO}\beta)$$

$$\frac{C \text{ is iLO}\beta}{\lambda x.C \text{ is iLO}\beta}\ (\lambda\text{-iLO}\beta) \qquad \frac{t \text{ is neutral} \qquad C \text{ is iLO}\beta}{tC \text{ is iLO}\beta}\ (@\text{r-iLO}\beta)$$

As expected,

Lemma 3.4 ($\to_{\mathrm{LO}\beta}$-steps are Inductively LOβ)**.** Let t be a λ-term and C a redex in t. C is the LOβ redex in t iff C is iLOβ.

Proof. The left-to-right implication is by induction on C. The right-to-left implication is by induction on the definition of C *is iLO*β. □

4. The Shallow Linear Substitution Calculus

4.1. **Statics.** The language of the *linear substitution calculus* (LSC for short) is given by the following grammar for terms:

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

The constructor $t[x{\leftarrow}u]$ is called an *explicit substitution* (of u for x in t). Both $\lambda x.t$ and $t[x{\leftarrow}u]$ bind x in t. In general, we assume a strong form of Barendregt's convention: any two bound or free variables have distinct names. We also silently work modulo α-equivalence of bound variables to preserve the convention, *e.g.* $(xy)[y{\leftarrow}t]\{x{\leftarrow}y\} = (yz)[z{\leftarrow}t\{x{\leftarrow}y\}]$ and $(xy)[y{\leftarrow}t]\{y{\leftarrow}u\} = (xz)[z{\leftarrow}t\{y{\leftarrow}u\}]$ where z is fresh.

The operational semantics of the LSC is parametric in a notion of (one-hole) context. General contexts simply extend the contexts for λ-terms with the two cases for explicit substitutions:

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

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 $t[x{\leftarrow}S]$):

$$S, S', S'', S''' ::= \langle\cdot\rangle \ \mid\ \lambda x.S \ \mid\ St \ \mid\ tS \ \mid\ S[x{\leftarrow}t].$$

In the following, whenever we refer to a *context* without further specification, it is implicitly assumed that it is a *shallow* context. We write $S \prec_p t$ if there is a term u such that $S\langle u\rangle = t$, and call it the *prefix relation*.

A special class of contexts is that of *substitution contexts*:

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

Remark 4.1 (α-Equivalence for Contexts)**.** While Barendregt's convention can always be achieved for terms, for contexts the question is subtler. Plugging in a context S, indeed, is *not* a capture-avoiding operation, so it is not stable by α-renaming S, as renaming can change the set of variables captured by S (if the hole of the context appears in the scope of the binder). Nonetheless, taking into account both the context S and the term t to be plugged into S, one can always rename both the bound variable in S and its free occurrences in t and satisfy the convention. Said differently, the contexts we consider are always obtained by splitting a term t as a subterm u and a context S such that $S\langle u\rangle = t$, so we assume that t has been renamed before splitting it into S and u, guaranteeing that S respects the convention. In particular, we shall freely assume that in $t[x\leftarrow u]$ and $S[x\leftarrow u]$ there are no free occurrences of x in u, as this can always be obtained by an appropriate α-conversion.

4.2. **Dynamics.** The (shallow) rewriting rules $\to_{\mathtt{dB}}$ ($\mathtt{dB} = \beta$ *at a distance*) and $\to_{\mathtt{ls}}$ ($\mathtt{ls}$ = linear substitution) are given by:

Rule at Top Level	(Shallow) Contextual closure
$L\langle\lambda x.t\rangle u \mapsto_{\mathtt{dB}} L\langle t[x\leftarrow u]\rangle$	$S\langle t\rangle \to_{\mathtt{dB}} S\langle u\rangle$ if $t \mapsto_{\mathtt{dB}} u$
$S\langle x\rangle[x\leftarrow u] \mapsto_{\mathtt{ls}} S\langle u\rangle[x\leftarrow u]$	$S\langle t\rangle \to_{\mathtt{ls}} S\langle u\rangle$ if $t \mapsto_{\mathtt{ls}} u$

and the union of $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ls}}$ is simply noted $\to$.

Let us point out a slight formal abuse of our system: rule $\to_{\mathtt{ls}}$ does not preserve Barendregt's convention (shortened BC), as it duplicates the bound names in u, so BC is not stable by reduction. To preserve BC it would be enough to replace the target term with $S\langle u^\alpha\rangle[x\leftarrow u]$, where u^α is an α-equivalent copy of u such that all bound names in u have been replaced by fresh and distinct names. Such a renaming can be done while copying u and thus does not affect the complexity of implementing $\to_{\mathtt{ls}}$. In order to lighten this already technically demanding paper, however, we decided to drop an explicit and detailed treatment of α-equivalence, and so we simply stick to $S\langle u\rangle[x\leftarrow u]$, letting the renaming implicit.

The implicit use of BC also rules out a few degenerate rewriting sequences. For instance, the following degenerated behavior is *not* allowed

$$x[x\leftarrow xx] \quad \to_{\mathtt{ls}} \quad (xx)[x\leftarrow xx] \quad \to_{\mathtt{ls}} \quad ((xx)x)[x\leftarrow xx] \to_{\mathtt{ls}} \dots$$

because the initial term does not respect BC. By α-equivalence we rather have the following evaluation sequence, ending on a normal form

$$x[x\leftarrow xx] \quad \equiv_\alpha \quad y[y\leftarrow xx] \quad \to_{\mathtt{ls}} \quad (xx)[y\leftarrow xx]$$

Finally, BC implies that in $\to_{\mathtt{ls}}$ 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 [Acc12]) and lacks some of the nice properties of the LSC (obtained simply by replacing shallow contexts by general contexts). Its relevance lies in the fact that it is the smallest fragment implementing linear LO reduction (see forthcoming Definition 4.5). The following are examples of shallow steps:

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

while the following are not

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

With respect to the literature on the LSC we slightly abuse the notation, as $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ls}}$ are usually used for the unrestricted versions, while here we adopt them for their shallow variants. Let us also warn the reader of a possible source of confusion: in the literature there exists an alternative notation and terminology in use for the LSC, stressing the linear logic interpretation, for which $\to_{\mathtt{dB}}$ is noted $\to_{\mathtt{m}}$ and called *multiplicative* (cut-elimination rule) and $\to_{\mathtt{ls}}$ is noted $\to_{\mathtt{e}}$ and called *exponential*.

Taking the external context into account, a substitution step has the following *explicit* form: $S'\langle S\langle x\rangle[x{\leftarrow}u]\rangle \to_{\mathtt{ls}} S'\langle S\langle u\rangle[x{\leftarrow}u]\rangle$. We shall often use a *compact* form:

Linear Substitution in Compact Form
$$S''\langle x\rangle \to_{\mathtt{ls}} S''\langle u\rangle \qquad \text{if } S'' = S'\langle S[x{\leftarrow}u]\rangle$$

Since every $\to_{\mathtt{ls}}$ step has a unique compact form, and a shallow context is the compact form of at most one $\to_{\mathtt{ls}}$ step, it is natural to use the compact context of a $\to_{\mathtt{ls}}$ step as its position.

Definition 4.2. Given a $\to_{\mathtt{dB}}$-redex $S\langle t\rangle \to_{\mathtt{dB}} S\langle u\rangle$ with $t \mapsto_{\mathtt{dB}} u$ or a compact $\to_{\mathtt{ls}}$-redex $S\langle x\rangle \to_{\mathtt{ls}} S\langle t\rangle$, the *position* of the redex is the context S.

As for λ-calculus, we identify a redex with its position, thus using S, S', S'' for redexes, and use $\rho : t \to^k u$ for (possibly empty) derivations. We write $|t|_{[\cdot]}$ for the number of substitutions in t and $|\rho|_{\mathtt{dB}}$ for the number of $\mathtt{dB}$-steps in ρ.

4.3. **Linear LO Reduction.** We redefine the LO order on contexts to accommodate ES.

Definition 4.3. The following definitions are given with respect to general (not necessarily shallow) contexts, even if apart from Sect. 11 we will use them only for shallow contexts.

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

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

Some examples:

$$\begin{array}{lll} (\lambda x.\langle\cdot\rangle)t & \prec_O (\lambda x.(\langle\cdot\rangle[y{\leftarrow}u]))t; & \\ t[x{\leftarrow}\langle\cdot\rangle] & \prec_O t[x{\leftarrow}uS]; & \\ t[x{\leftarrow}S]u & \prec_L t[x{\leftarrow}r]\langle\cdot\rangle & \text{if } S \prec_p r. \end{array}$$

Note that the outside-in order $\prec_O$ can be seen as the prefix relation $\prec_p$ on contexts.

The next lemma guarantees that we defined a total order.

Lemma 4.4 (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 $C = D$.

Proof. By induction on t. □

Remember that we identify redexes with their position context and write $S \prec_{LO} S'$. We can now define LO reduction in the LSC, first considered in [ABKL14].

Definition 4.5 (LO Linear Reduction $\to_{\mathrm{LO}}$)**.** Let t be a term and S a redex of t. S is the *leftmost-outermost* (LO for short) redex of t if $S \prec_{LO} S'$ for every other redex S' of t. We write $t \to_{\mathrm{LO}} u$ if a step reduces the LO redex.

Technical Remark. Note that one cannot define $\to_{\mathrm{LO}}$ as the union of the two natural rules $\to_{LO\mathtt{dB}}$ and $\to_{LO\mathtt{ls}}$, reducing the LO `dB` and `ls` redexes, respectively. For example, if $I := \lambda z.z$ then $(xy)[x{\leftarrow}I](Iy) \to_{LO\mathtt{dB}} (xy)[x{\leftarrow}I](z[z{\leftarrow}y])$, while we have $(xy)[x{\leftarrow}I](Iy) \to_{\mathrm{LO}} (Iy)[x{\leftarrow}I](Iy)$, because the LO redex has to be chosen among both `dB` and `ls` redexes. Therefore, we will for instance say *given a* $\to_{LO}$ `dB`*-step* and not *given the LO* `dB`*-step*.

5. Unfoldings

In Sect. 1, we defined the unfolding $t{\downarrow}$ of a term t (Definition 1.2, page 7). Here we extend it in various ways. We first define context unfoldings, then we generalize the unfolding (of both terms and contexts) relatively to a context, and finally we unfold shallow derivations.

5.1. **Unfolding Contexts.** Shallowness is crucial here: the unfolding of a shallow context is still a context, because the hole cannot be duplicated by unfolding, being out of all ES. First of all, we define *substitution on (general) contexts*:

$$\begin{array}{rcl@{\qquad}rcl}
\langle\cdot\rangle\{x{\leftarrow}u\} & := & \langle\cdot\rangle & & & \\
(\lambda y.C)\{x{\leftarrow}u\} & := & \lambda y.C\{x{\leftarrow}u\} & (\lambda x.C)\{x{\leftarrow}u\} & := & \lambda x.C \\
(tC)\{x{\leftarrow}u\} & := & t\{x{\leftarrow}u\}C\{x{\leftarrow}u\} & (Ct)\{x{\leftarrow}u\} & := & C\{x{\leftarrow}u\}t\{x{\leftarrow}u\} \\
C[y{\leftarrow}t]\{x{\leftarrow}u\} & := & C\{x{\leftarrow}u\}[y{\leftarrow}t\{x{\leftarrow}u\}] & C[x{\leftarrow}t]\{x{\leftarrow}u\} & := & C[x{\leftarrow}t\{x{\leftarrow}u\}] \\
t[y{\leftarrow}C]\{x{\leftarrow}u\} & := & t\{x{\leftarrow}u\}[y{\leftarrow}C\{x{\leftarrow}u\}] & t[x{\leftarrow}C]\{x{\leftarrow}u\} & := & t[x{\leftarrow}C\{x{\leftarrow}u\}]
\end{array}$$

Note that the definition of $S\{x{\leftarrow}u\}$ assumes that the free variables of u are not captured by S (that means that for instance $y \notin \mathtt{fv}(u)$ in $(\lambda y.C)\{x{\leftarrow}u\}$). This can always be achieved by α-renaming S (according to Remark 4.1).

And then define *context unfolding* $S{\downarrow}$ as:

$$\begin{array}{rcl@{\qquad}rcl}
\langle\cdot\rangle{\downarrow} & := & \langle\cdot\rangle & (\lambda x.S){\downarrow} & := & \lambda x.S{\downarrow} \\
(tS){\downarrow} & := & t{\downarrow}S{\downarrow} & (St){\downarrow} & := & S{\downarrow}t{\downarrow} \\
S[x{\leftarrow}t]{\downarrow} & := & S{\downarrow}\{x{\leftarrow}t{\downarrow}\} & & &
\end{array}$$

We have the following properties.

Lemma 5.1. Let S be a shallow contexts. Then:
(1) $S{\downarrow}$ is a context;
(2) $S\langle t\rangle\{x{\leftarrow}u\} = S\{x{\leftarrow}u\}\langle t\{x{\leftarrow}u\}\rangle$;

Proof. By induction on S. □

An important notion of context will be that of applicative context, *i.e.* of context whose hole is applied to an argument, and that if plugged with an abstraction provides a dB-redex.

Definition 5.2 (Applicative Context)**.** An *applicative context* is a context $A ::= S\langle Lt\rangle$, where S and L are a shallow and a substitution context, respectively.

Note that applicative contexts are *not* made out of applications only: $t(\lambda x.(\langle\cdot\rangle[y{\leftarrow}u]r))$ is an applicative context.

Lemma 5.3. Let S be a context. Then,
(1) S is applicative iff $S\{x{\leftarrow}t\}$ is applicative;
(2) S is applicative iff $S{\downarrow}$ is applicative.

5.2. **Relative Unfoldings.** Useful reduction will require a more general notion of unfolding and context unfolding. The usefulness of a redex, in fact, will depend crucially on the context in which it takes place. More precisely, it will depend on the unfolding of the term extended with the substitutions that the surrounding context can provide—this is the unfolding of a term *relative to* a context. Moreover, relative unfoldings will also be needed for contexts.

Definition 5.4 (Relative Unfolding)**.** Let S be a (shallow) context (verifying, as usual, Barendregt's convention—see also the remark after this definition). The *unfolding $t{\downarrow_S}$ of a term t relative to S* and the *unfolding $S'{\downarrow_S}$ of a (shallow) context S' relative to S* are defined by:

$$\begin{array}{lcllcl}
t{\downarrow_{\langle\cdot\rangle}} & := & t{\downarrow} & S'{\downarrow_{\langle\cdot\rangle}} & := & S'{\downarrow}\\
t{\downarrow_{\lambda x.S}} & := & t{\downarrow_S} & S'{\downarrow_{\lambda x.S}} & := & S'{\downarrow_S}\\
t{\downarrow_{uS}} & := & t{\downarrow_S} & S'{\downarrow_{uS}} & := & S'{\downarrow_S}\\
t{\downarrow_{Su}} & := & t{\downarrow_S} & S'{\downarrow_{Su}} & := & S'{\downarrow_S}\\
t{\downarrow_{S[x\leftarrow u]}} & := & t{\downarrow_S}\{x{\leftarrow}u{\downarrow}\} & S'{\downarrow_{S[x\leftarrow u]}} & := & S'{\downarrow_S}\{x{\leftarrow}u{\downarrow}\}
\end{array}$$

For instance, $(xy){\downarrow_{(\langle\cdot\rangle[y\leftarrow x]t)[x\leftarrow\lambda z.(zz)]}} = (\lambda z.(zz))(\lambda z.(zz))$.

Remark 5.5 (Relative Unfoldings and Barendregt's Convention)**.** Let us point out that the definition of relative unfolding $t{\downarrow_S}$ relies crucially on the use of Barendregt's convention for contexts (according to Remark 4.1). For contexts not respecting the convention, in fact, the definition does not give the intended result. For instance,

$$x{\downarrow_{(\lambda x.\langle\cdot\rangle)[x\leftarrow y]}} = x{\downarrow_{(\lambda x.\langle\cdot\rangle)}}\{x{\leftarrow}y\} = x{\downarrow_{\langle\cdot\rangle}}\{x{\leftarrow}y\} = x\{x{\leftarrow}y\} = y$$

while the result should clearly be x.

We also state some further properties of relative unfolding, to be used in the proofs, and proved by easy inductions.

Lemma 5.6 (Properties of Relative Unfoldings)**.** Let t and u be terms and S and S' be shallow contexts.
(1) *Well-Definedness*: $S{\downarrow_{S'}}$ is a context.
(2) *Commutation*: the following equalities hold (and in those on the right S is assumed to not capture x)

$$\begin{array}{rclrcl}
(tu){\downarrow_S} & = & t{\downarrow_S}u{\downarrow_S} & (\lambda x.t){\downarrow_S} & = & \lambda x.t{\downarrow_S}\\
t\{x{\leftarrow}u\}{\downarrow_S} & = & t[x{\leftarrow}u]{\downarrow_S} & t\{x{\leftarrow}u\}{\downarrow_S} & = & t{\downarrow_S}\{x{\leftarrow}u{\downarrow_S}\}\\
S{\downarrow}\{x{\leftarrow}t{\downarrow}\} & = & S\{x{\leftarrow}t{\downarrow}\}{\downarrow} & t{\downarrow_{S[x\leftarrow u]}} & = & t\{x{\leftarrow}u{\downarrow}\}{\downarrow_{S\{x\leftarrow u{\downarrow}\}}}
\end{array}$$

(3) *Freedom*: if S does not capture any free variable of t then $t{\downarrow_S} = t{\downarrow}$.
(4) *Relativity*: if $t{\downarrow} = u{\downarrow}$ then $t{\downarrow_S} = u{\downarrow_S}$ and if $t{\downarrow}$ has a β-redex then so does $t{\downarrow_S}$.
(5) *Splitting*: $t{\downarrow_{S\langle S'\rangle}} = t{\downarrow_{S'}}{\downarrow_S}$.
(6) *Factorization*: $S'\langle t\rangle{\downarrow_S} = S'{\downarrow_S}\langle t{\downarrow_{S\langle S'\rangle}}\rangle$, and in particular
   (a) $S\langle t\rangle{\downarrow} = S{\downarrow}\langle t{\downarrow_S}\rangle$,
   (b) $L\langle t\rangle{\downarrow_S} = t{\downarrow_{S\langle L\rangle}}$, and
   (c) if $S' \prec_p u$ then $S'{\downarrow} \prec_p u{\downarrow}$.

5.3. **Unfolding Derivations.** 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. It is built in two steps.

As expected, linear substitution steps do not modify the unfolding, as the next lemma shows. Its proof is a nice application of the properties of contexts and (relative) unfoldings, allowed by the restriction to shallow contexts (the property is valid more generally for unrestricted `ls` steps, but we will need it only for the shallow ones).

Lemma 5.7 ($\to_{\texttt{ls}}$ Projects on $=$)**.** If $t \to_{\texttt{ls}} u$ then $t{\downarrow} = u{\downarrow}$.

Proof. If $t \to_{\texttt{ls}} u$ is a root step, *i.e.* $t = S\langle x\rangle[x{\leftarrow}r] \mapsto_{\texttt{ls}} S\langle r\rangle[x{\leftarrow}r] = u$ then

$$\begin{array}{rll} S\langle x\rangle[x{\leftarrow}r]{\downarrow} & = & S\langle x\rangle{\downarrow}\{x{\leftarrow}r{\downarrow}\} \\ & =_{5.6.6a} & S{\downarrow}\langle x{\downarrow_S}\rangle\{x{\leftarrow}r{\downarrow}\} \\ & =_{5.6.3} & S{\downarrow}\langle x\rangle\{x{\leftarrow}r{\downarrow}\} \\ & =_{5.1.2} & S{\downarrow}\{x{\leftarrow}r{\downarrow}\}\langle r{\downarrow}\rangle \\ & = & S{\downarrow}\{x{\leftarrow}r{\downarrow}\}\langle r{\downarrow}\{x{\leftarrow}r{\downarrow}\}\rangle \\ & =_{5.1.2} & S{\downarrow}\langle r{\downarrow}\rangle\{x{\leftarrow}r{\downarrow}\} \\ & =_{5.6.3} & S{\downarrow}\langle r{\downarrow_S}\rangle\{x{\leftarrow}r{\downarrow}\} \\ & =_{5.6.6a} & S\langle r\rangle{\downarrow}\{x{\leftarrow}r{\downarrow}\} \\ & = & S\langle r\rangle[x{\leftarrow}r]{\downarrow} \end{array}$$

Suppose instead $t \to_{\texttt{ls}} u$ because $t = S'\langle r\rangle \to_{\texttt{ls}} S'\langle p\rangle = u$ with $r \mapsto_{\texttt{ls}} p$. By what we just proved we obtain $r{\downarrow} = p{\downarrow}$, that gives $r{\downarrow_{S'}} = p{\downarrow_{S'}}$ by Lemma 5.6.4. Then $S'\langle r\rangle{\downarrow} =_{5.6.6a} = S'{\downarrow}\langle r{\downarrow_{S'}}\rangle = S'{\downarrow}\langle p{\downarrow_{S'}}\rangle =_{5.6.6a} = S'\langle p\rangle{\downarrow}$. □

Instead, `dB`-steps project to β-steps. Because of shallowness, we actually obtain a strong form of projection, as every `dB`-step projects on a single β-step. We are then allowed to identify `dB` and β-redexes.

Lemma 5.8 ($\to_{\texttt{dB}}$ Strongly Projects on $\to_\beta$)**.** Let t be a LSC term and $S : t \to_{\texttt{dB}} u$ a `dB`-redex in t. Then $S{\downarrow} : t{\downarrow} \to_\beta u{\downarrow}$.

Proof. Let $t = S\langle r\rangle \to_{\texttt{dB}} S\langle p\rangle = u$ with $r \mapsto_{\texttt{dB}} p$. We show that $S{\downarrow} : t{\downarrow} = S{\downarrow}\langle r{\downarrow_S}\rangle \to_\beta S{\downarrow}\langle p{\downarrow_S}\rangle = u{\downarrow}$ with $r{\downarrow_S} \mapsto_\beta p{\downarrow_S}$ ($\mapsto_{\texttt{dB}}$ and $\mapsto_\beta$ stands for `dB`/β-reduction at top level). By induction on S. Cases:
(1) *Empty context* $S = \langle\cdot\rangle$. Notation: given $L = [x_1{\leftarrow}r_1]\dots[x_n{\leftarrow}r_n]$ we denote with $\hat{L}$ the list of implicit substitutions $\{x_1{\leftarrow}r_1\}\dots\{x_n{\leftarrow}r_n\}$. Then $t = r = L\langle\lambda x.q\rangle s$, $t{\downarrow} = r{\downarrow_S} = r{\downarrow}$, $u = p$, $u{\downarrow} = p{\downarrow_S} = p{\downarrow}$, and

$$\begin{array}{ccc} t = L\langle\lambda x.q\rangle s & \xrightarrow{\texttt{dB}} & L\langle q[x{\leftarrow}s]\rangle = u \\ \big\downarrow\downarrow & & \big\downarrow\downarrow \\ t\downarrow = (\lambda x.q\downarrow\hat{L})(s\downarrow) & \overset{\beta}{\to} & q\downarrow\hat{L}\{x{\leftarrow}s\downarrow\} = q\downarrow\{x{\leftarrow}s\downarrow\}\hat{L} = u\downarrow \end{array}$$

where the first equality in the South-East corner is given by the fact that x does not occur in L and the variables on which $\hat{L}$ substitutes do not occur in s, as is easily seen by looking at the starting term. Thus the implicit substitutions $\hat{L}$ and $\{x{\leftarrow}s\downarrow\}$ commute.

(2) *Abstraction* $S = \lambda x.S'$. It follows immediately by the *i.h.*

(3) *Left of an application* $S = S'q$. By Lemma 5.6.2 we know that $t\downarrow = S'\downarrow\langle r\downarrow_{S'}\rangle q\downarrow$. Using the *i.h.* we derive the following diagram:

$$\begin{array}{ccc} S'\langle r\rangle q & \xrightarrow{\texttt{dB}} & S'\langle p\rangle q \\ \big\downarrow\downarrow & & \big\downarrow\downarrow \\ S'\downarrow\langle r\downarrow_{S'}\rangle q\downarrow & \xrightarrow{\beta} & S'\downarrow\langle p\downarrow_{S'}\rangle q\downarrow \end{array}$$

(4) *Right of an application* $S = qS'$. By Lemma 5.6.2 we know that $t\downarrow = q\downarrow S'\downarrow\langle r\downarrow_{S'}\rangle$. Using the *i.h.* we derive the following diagram:

$$\begin{array}{ccc} qS'\langle r\rangle & \xrightarrow{\texttt{dB}} & qS'\langle p\rangle \\ \big\downarrow\downarrow & & \big\downarrow\downarrow \\ q\downarrow S'\downarrow\langle r\downarrow_{S'}\rangle & \xrightarrow{\beta} & q\downarrow S'\downarrow\langle p\downarrow_{S'}\rangle \end{array}$$

(5) *Substitution* $S = S'[x{\leftarrow}q]$. By *i.h.* $S'\downarrow\langle r\downarrow_{S'}\rangle \to_\beta S'\downarrow\langle p\downarrow_{S'}\rangle$ with $r\downarrow_{S'} \mapsto_\beta p\downarrow_{S'}$. In general, from the definition of substitution it follows $s \mapsto_\beta s'$ implies $s\{x{\leftarrow}s''\} \mapsto_\beta s'\{x{\leftarrow}s''\}$. Therefore, $r\downarrow_{S'}\{x{\leftarrow}q\downarrow\} \mapsto_\beta p\downarrow_{S'}\{x{\leftarrow}q\downarrow\}$. The following calculation concludes the proof:

$$\begin{array}{lll} t\downarrow & = & S'\downarrow\langle r\downarrow_{S'}\rangle\{x{\leftarrow}q\downarrow\} \\ & =_{5.1.2} & S'\downarrow\{x{\leftarrow}q\downarrow\}\langle r\downarrow_{S'}\{x{\leftarrow}q\downarrow\}\rangle \\ & \to_\beta & S'\downarrow\{x{\leftarrow}q\downarrow\}\langle p\downarrow_{S'}\{x{\leftarrow}q\downarrow\}\rangle \\ & =_{5.1.2} & S'\downarrow\langle p\downarrow_{S'}\rangle\{x{\leftarrow}q\downarrow\}u\downarrow \end{array}$$

□

6. Useful Derivations

In this section we define useful reduction, a constrained, optimized reduction, that will be the key to the invariance theorem. The idea is that an optimized substitution step $S\langle x\rangle[x{\leftarrow}u] \to_{\texttt{ls}} S\langle u\rangle[x{\leftarrow}u]$ takes place only if it contributes to eventually obtain an unshared (*i.e.* shallow) $\beta/\texttt{dB}$-redex. *Absolute* usefulness can be of two kinds.

(1) *Duplication*: a step can *duplicate* dB-redexes, as in

$$S\langle x\rangle[x{\leftarrow}(\lambda y.r)p] \to_{\texttt{ls}} S\langle(\lambda y.r)p\rangle[x{\leftarrow}(\lambda y.r)p]$$

(2) *Creation*: it can *create* a new dB-redex with its context, if it substitutes an abstraction in an applicative context, as in

$$S\langle L\langle x\rangle u\rangle[x{\leftarrow}\lambda y.t] \to_{\texttt{ls}} S\langle L\langle\lambda y.t\rangle u\rangle[x{\leftarrow}\lambda y.t]$$

There is also a subtler *relative* usefulness to dB-redexes. A substitution step may indeed put just a piece of what later, with further substitutions, will become a dB-redex. Accommodating relative usefulness requires to generalize the duplication and the creation cases to contexts and relative unfoldings.

Let us give some examples. The following step

$$(tx)[x\leftarrow yy] \to_{\mathtt{ls}} (t(yy))[x\leftarrow yy]$$

is useless. However, in an appropriate context it is relatively useful. For instance, let us put it in $\langle\cdot\rangle[y\leftarrow\lambda z.z]$, obtaining a case of *relatively useful duplication*,

$$(tx)[x\leftarrow yy][y\leftarrow\lambda z.z] \to_{\mathtt{ls}} (t(yy))[x\leftarrow yy][y\leftarrow\lambda z.z] \tag{6.1}$$

Note that the step, as before, does not duplicate a dB-redex. Now, however, evaluation will continue and turn the substituted copy of yy into a dB-redex, as follows

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

We consider the step in (6.1) a case of relative duplication because yy contains a β-redex up to relative unfolding in its context, as we have $(yy)\downarrow_{\langle\cdot\rangle[y\leftarrow\lambda z.z]} = (\lambda z.z)(\lambda z.z)$, and thus duplicating yy duplicates a β-redex, up to unfolding.

Similarly, a case of *relatively useful creation* is given by:

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

Again, the step itself does not create a dB-redex, but—up to unfolding—it substistutes an abstraction, because $y\downarrow_{\langle\cdot\rangle[y\leftarrow\lambda z.z]} = \lambda z.z$, and the context is *applicative* (note that a context is applicative iff it is applicative up to unfolding, by Lemma 5.3).

The actual definition of useful reduction captures at the same time absolute and relative cases by means of relative unfoldings.

Definition 6.1 (Useful/Useless Steps and Derivations)**.** A *useful* step is either a dB-step or a ls-step $S\langle x\rangle \to_{\mathtt{ls}} S\langle r\rangle$ (in compact form) such that:
(1) *Relative Duplication*: either $r\downarrow_S$ contains a β-redex, or
(2) *Relative Creation*: $r\downarrow_S$ is an abstraction and S is applicative.
A *useless* step is a ls-step that is not useful. A *useful derivation* (resp. *useless derivation*) is a derivation whose steps are useful (resp. useless).

Note that a useful normal form, *i.e.* a term that is normal for useful reduction, is not necessarily a normal form. For instance, the reader can now verify that the compact normal form we discussed in Sect. 1, namely

$$(y_1y_1)[y_1\leftarrow y_2y_2][y_2\leftarrow y_3y_3]\dots[y_n\leftarrow x],$$

is a useful normal form, but not a normal form.

As a first sanity check for useful reduction, we show that as long as there are useful substitutions steps to do, the unfolding is not $\to_\beta$-normal.

Lemma 6.2. If $S\langle x\rangle \to_{\mathtt{ls}} S\langle t\rangle$ is useful then $S\langle x\rangle\downarrow = S\langle t\rangle\downarrow$ has a β-redex.

Proof. The equality $S\langle x\rangle\downarrow = S\langle t\rangle\downarrow$ holds in general for $\to_{\mathtt{ls}}$-steps (Lemma 5.7). For the existence of a β-redex, note that $S\langle t\rangle\downarrow = S\downarrow\langle t\downarrow_S\rangle$ by Lemma 5.6.6a, and that $S\downarrow$ applicative iff S is applicative by Lemma 5.3.2. Then by relative duplication or relative creation there is a $\to_\beta$-redex in $S\langle x\rangle\downarrow$. □

We can finally define the strategy that will be shown to implement LO β-reduction within a polynomial overhead.

Definition 6.3 (LO Useful Reduction $\to_{\mathrm{LOU}}$)**.** Let t be a term and S a redex of t. S is the *leftmost-outermost useful* (LOU for short) redex of t if $S \prec_{LO} S'$ for every other useful redex S' of t. We write $t \to_{\mathrm{LOU}} u$ if a step reduces the LOU redex.

6.1. **On Defining Usefulness via Residuals.** Note that useful steps concern future creations of β-redexes and yet circumvent the explicit use of residuals, relying on relative unfoldings only. It would be interesting, however, to have a characterization based on residuals. We actually spent time investigating such a characterization, but we decide to leave it to future work. We think that it is informative to know the reasons, that are the following:

(1) a definition based on residuals is not required for the final result of this paper;
(2) the definition based on relative unfoldings is preferable, as it allows the complexity analysis required for the final result;
(3) we believe that the case studied in this paper, while certainly relevant, is not enough to develop a general, abstract theory of usefulness. We feel that more concrete examples should first be developed, for instance in call-by-value and call-by-need scenarios, and comparing weak and strong variants, extending the language with continuations or pattern matching, and so on. The complementary study in [ASC15], indeed, showed that the weak call-by-value case already provides different insights, and that useful sharing as studied here is only an instance of a more general concept;
(4) we have a candidate characterization of useful reduction using residuals, for which however one needs sophisticated rewriting theory. It probably deserves to be studied in another paper. Our candidate characterization relies on a less rigid order between redexes of the LSC than the total order $\prec_{LO}$ considered here, namely the partial *box order* $\prec_{\mathtt{box}}$ studied in [ABKL14]. Our conjecture is that an $\to_{\mathtt{ls}}$ redex S is useful iff it is shallow and
   (a) there is a (not necessarily shallow) $\to_{\mathtt{dB}}$ redex C such that $S \prec_{\mathtt{box}} C$, or
   (b) S creates a shallow $\to_{\mathtt{dB}}$ redex, or
   (c) there is a (not necessarily shallow) $\to_{\mathtt{ls}}$ redex C such that $S \prec_{\mathtt{box}} C$ and there exists a residual D of C after S that is useful.

   Coming back to the previous point, we feel that such an abstract characterization—assuming it holds—is not really satisfying, as it relies too much on the concrete notion of shallow redex. It is probably necessary to abstract away from a few cases in order to find the right notion. An obstacle, however, is that the rewriting theory developed in [ABKL14] has yet to be adapted to call-by-value and call-by-need.

To conclude, while having a residual theory of useful sharing is certainly both interesting and challenging, it is also certainly not necessary in order to begin a theory of cost models for the λ-calculus.

7. The Proof, Made Abstract

Here we describe the architecture of our proof, decomposing it, and proving the implementation theorem from a few abstract properties. The aim is to provide a tentative recipe for a general proof of invariance for functional languages.

We want to show that a certain *abstract* 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 a Turing machine or a RAM.

In our case, $\rightsquigarrow$ will be LO β-reduction $\to_{\mathrm{LO}\beta}$. Such a choice is natural, as $\to_{\mathrm{LO}\beta}$ is normalizing, it produces standard derivations, and it is an iteration of head reduction.

Because of size-explosion in the λ-calculus, we have to add sharing, and our framework for sharing is the *(shallow) linear substitution calculus*, that plays the role of a very abstract intermediate machine between λ-terms and Turing machines. Our encoding will rather address an informal notion of an algorithm rather than Turing machines. The algorithms will be clearly implementable with polynomial overhead but details of the implementation will not be discussed (see however Sect. 16).

In the LSC, $\to_{\mathrm{LO}\beta}$ is implemented by LO useful reduction $\to_{\mathrm{LOU}}$. We say that $\to_{\mathrm{LOU}}$ is a *partial strategy* of the LSC, because the useful restriction forces it to stop on compact normal forms, that in general are not normal forms of the LSC. Let us be abstract, and replace $\to_{\mathrm{LOU}}$ with a general partial strategy $\rightsquigarrow_X$ within the LSC. We want to show that $\rightsquigarrow_X$ is invariant with respect to both $\rightsquigarrow$ and RAM. Then we need two theorems, which together—when instantiated to the strategies $\to_{\mathrm{LO}\beta}$ and $\to_{\mathrm{LOU}}$—yield 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 a RAM with an overhead in time which is polynomial in both k and the size of t.

7.1. **High-Level Implementation.** The high-level half relies on the following notion.

Definition 7.1 (High-Level Implementation System)**.** Let $\rightsquigarrow$ be a deterministic strategy on λ-terms and $\rightsquigarrow_X$ a partial strategy of the shallow LSC. The pair $(\rightsquigarrow, \rightsquigarrow_X)$ is a *high-level implementation system* if whenever t is a LSC term it holds that:

(1) *Normal Form*: if t is a $\rightsquigarrow_X$-normal form then $t{\downarrow}$ is a $\rightsquigarrow$-normal form.
(2) *Strong Projection*: if $t \rightsquigarrow_X u$ is a $\to_{\mathtt{dB}}$ step then $t{\downarrow} \rightsquigarrow u{\downarrow}$.

Moreover, it is *locally bounded* if whenever t is a λ-term and $\rho : t \rightsquigarrow_X^* u$ then the length of a sequence of `ls`-steps from u is linear in the number $|\rho|_{\mathtt{dB}}$ of (the past) `dB`-steps in ρ.

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 $\rightsquigarrow_X$. It also states a stronger fact: $\rightsquigarrow$ *steps can be identified with the* `dB`*-steps of the* $\rightsquigarrow_X$ *strategy.* Note that the fact that one $\to_{\mathtt{dB}}$ step projects on exactly one $\to_\beta$-step is a general property of the shallow LSC, given by Lemma 5.8. The projection property then requires that the steps selected by the two strategies coincide up to unfolding.

The *local boundedness* property is instead used for the *quantitative* part of the theorem, *i.e.* to provide the polynomial bound. A simple argument indeed bounds the *global* number of `ls`-steps in $\rightsquigarrow_X$ 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.

Theorem 7.2 (High-Level Implementation)**.** Let t be an ordinary λ-term and $(\rightsquigarrow, \rightsquigarrow_X)$ a high-level implementation system. Then,

(1) *Normalization*: t is $\rightsquigarrow$-normalizing iff it is $\rightsquigarrow_X$-normalizing,
(2) *Projection*: if $\rho : t \rightsquigarrow_X^* u$ then $\rho{\downarrow} : t \rightsquigarrow^* u{\downarrow}$,
(3) *Overhead*: if the system is locally bounded, then ρ is at most quadratically longer than $\rho{\downarrow}$, *i.e.* $|\rho| = O(|\rho{\downarrow}|^2)$.

Proof.
(1) $\Leftarrow$) Suppose that t is $\rightsquigarrow_X$-normalizable 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 $\rightsquigarrow$-normalizable 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 $\rightsquigarrow_X$-normalizable. 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 local boundedness, $\rightsquigarrow_X$ can make only a finite number of $\mathtt{ls}$ steps (more generally, $\rightarrow_{\mathtt{ls}}$ is strongly normalizing 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) From Lemma 5.7 ($\rightarrow_{\mathtt{ls}}$ projects on $=$) and Lemma 5.8 (a single shallow $\rightarrow_{\mathtt{dB}}$ projects on a single $\rightarrow_\beta$ step) we obtain $\rho{\downarrow} : t{\downarrow} \rightarrow_\beta^* u{\downarrow}$ with $|\rho{\downarrow}| = |\rho|_{\mathtt{dB}}$, and by the projection property the steps of $\rho{\downarrow}$ are $\rightsquigarrow$ steps.
(3) To show $|\rho| = O(|\rho{\downarrow}|^2)$ it is enough to show $|\rho| = O(|\rho|_{\mathtt{dB}}^2)$. Now, ρ has the shape:

$$t = r_1 \rightarrow_{\mathtt{dB}}^{a_1} p_1 \rightarrow_{\mathtt{ls}}^{b_1} r_2 \rightarrow_{\mathtt{dB}}^{a_2} p_2 \rightarrow_{\mathtt{ls}}^{b_2} \dots r_k \rightarrow_{\mathtt{dB}}^{a_k} p_k \rightarrow_{\mathtt{ls}}^{b_k} u$$

By the local boundedness, we obtain $b_i \leq c \cdot \sum_{j=1}^{i} a_j$ for some constant c. Then:

$$|\rho|_{\mathtt{ls}} = \sum_{i=1}^{k} b_i \leq \sum_{i=1}^{k} (c \cdot \sum_{j=1}^{i} a_j) = c \cdot \sum_{i=1}^{k} \sum_{j=1}^{i} a_j$$

Note that $\sum_{j=1}^{i} a_j \leq \sum_{j=1}^{k} a_j = |\rho|_{\mathtt{dB}}$ and $k \leq |\rho|_{\mathtt{dB}}$. So

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

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

Note that the properties of the implementation hold for *all* derivations (and not only for those reaching normal forms). In fact, they even hold for derivations in strongly diverging terms. In this sense, our cost model is robust.

7.2. **Low-Level Implementation.** For the low-level part we define three basic requirements.

Definition 7.3. A partial strategy $\rightsquigarrow_X$ on LSC terms is *efficiently mechanizable* if given a derivation $\rho : t \rightsquigarrow_X^* u$:
(1) *No Size-Explosion*: $|u|$ is polynomial in $|t|$ and $|\rho|$;
(2) *Step*: every redex of u can be implemented in time polynomial in $|u|$;
(3) *Selection*: the search for the next $\rightsquigarrow_X$ redex to reduce in u takes polynomial time in $|u|$.

The first two properties are natural. 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 takes exponential time in $|u|$. Then one has to be sure that such a computation can be done in polynomial time.

Theorem 7.4 (Low-Level Implementation)**.** Let $\rightsquigarrow_X$ be an efficiently mechanizable strategy, t a λ-term, and k a number of steps. Then there is an algorithm that outputs $\rightsquigarrow_X^k(t)$, and which works in time polynomial in k and $|t|$.

Proof. The algorithm for computing $\rightsquigarrow_X^k(t)$ is obtained by iteratively searching for the next $\rightsquigarrow_X$ redex to reduce and then reducing it, by using the algorithms given by the step and selection property. The complexity is obtained by summing the polynomials given by the step and selection property, that are in turn composed with the polynomial of the no size-explosion property. Since polynomials are closed by sum and composition, the algorithm works in polynomial time. □

In [ADL12], we proved that *head reduction* and *linear head reduction* form a locally bounded high-level implementation system and that linear head reduction is efficiently mechanizable (but note that [ADL12] does not employ the terminology we use here).

Taking $\rightsquigarrow_X$ as the LO strategy $\to_{\mathrm{LO}}$ of the LSC, almost does the job. Indeed, $\to_{\mathrm{LO}}$ is efficiently mechanizable and $(\to_{\mathrm{LO}\beta}, \to_{\mathrm{LO}})$ is a high-level implementation system. Unfortunately, it is not a locally bounded implementation, because of the *length-explosion* example given in Sect. 1, and thus invariance does not hold. This is why useful reduction is required.

7.3. **Efficient Mechanizability and Subterms.** We have been very lax in the definition of efficiently mechanizable strategies. The strategy that we will consider has the following additional property.

Definition 7.5 (Subterm)**.** A partial strategy $\rightsquigarrow_X$ on LSC terms has the *subterm property* if given a derivation $\rho : t \rightsquigarrow_X^* u$ the terms duplicated along ρ are subterms of t.

The subterm property in fact enforces *linearity* in the no size-explosion and step properties, as the following immediate lemma shows.

Lemma 7.6 (Subterm ⇒ Linear No Size-Explosion + Linear Step)**.** If $\rho : t \rightsquigarrow_X^* u$ has the subterm property then
(1) *Linear Size*: $|u| \leq (|\rho| + 1) \cdot |t|$.
(2) *Linear Step*: every redex of u can be implemented in time linear in $|t|$;

The subterm property is fundamental and common to most implementations of functional languages [Jon96, SGM02, ADL12, ABM14], and for implementations and their complexity analysis it plays the role of the subformula property in sequent calculus. It is sometimes called *semi-compositionality* [Jon96]. We will show that every standard derivation for the LSC has the subterm property. To the best of our knowledge, instead, no strategy of the λ-calculus has the subterm property. We are not aware of a proof of the nonexistence of strategies for β-reduction with the subterm property, though. For a fixed strategy, however, it is easy to build a counterexample, as β-reduction substitutes everywhere, in particular in the argument of applications that can later become a redex. The reason why the subterm

property holds for many micro-step strategies, indeed, is precisely the fact that they do not substitute in such arguments, see also Sect. 12.

The reader may wonder, then, why we did not ask the subterm property of an efficiently mechanizable strategy. The reason is that we want to provide a general abstract theory, and there may very well be strategies that are efficiently mechanizable but that do not satisfy the subterm property, or, rather, that satisfy only some relaxed form of it.

Let us also point out that in the subterm property the word *subterm* conveys the good intuition but is slightly abused: since evaluation is up to α-equivalence, a subterm of t is actually a subterm up to variable names, both free and bound. More precisely: define r^- as r in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. Then, we will consider u to be a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\underline{u}|$ of $\underline{u}$ is bounded by $|\underline{t}|$, which is what is actually relevant for the complexity analysis.

8. Road Map

We need to ensure that LOU derivations are efficiently mechanizable and form a high-level implementation system when paired with LOβ derivations. These are non-trivial properties, with subtle proofs in the following sections. The following schema is designed to help the reader to follow the master plan:

(1) we will show that $(\to_{\mathrm{LO}\beta}, \to_{\mathrm{LOU}})$ is a high-level implementation system, by showing
   (a) the *normal form* property in Sect. 9
   (b) the *projection property* in Sect. 10, by introducing LOU contexts;
(2) we will prove the *local boundedness* property of $(\to_{\mathrm{LO}\beta}, \to_{\mathrm{LOU}})$ through a detour via standard derivations. The detour is in three steps:
   (a) the introduction of *standard derivations* in Sect. 11, that are shown to have the *subterm property*;
   (b) the proof that *LOU derivations are standard* in Sect. 12, and thus have the subterm property;
   (c) the proof that LOU derivations have the *local boundedness property*, that relies on the subterm and normal form properties;
(3) we will prove that LOU derivations are efficiently mechanizable by showing:
   (a) the no size-explosion and step properties, that at this point are actually already known to hold, because they follow from the subterm property (Lemma 7.6);
   (b) the *selection property*, by exhibiting a polynomial algorithm to test whether a redex is useful or not, in Sect. 14.

In Sect. 15, we will put everything together, obtaining an implementation of the λ-calculus with a polynomial overhead, from which invariance follows.

9. The Normal Form Property

To show the normal form property we first have to generalize it to the relative unfolding in a context, in the next lemma, and then obtain it as a corollary.

The statement of the lemma essentially says that for a useful normal form u in a context S the unfolding $u{\downarrow_S}$ either unfolds to a $\to_\beta$-normal form or it has a useful substitution redex provided by S. The second case is stated in a more technical way, spelling out the components of the redex, and will be used twice later on, in Lemma 10.2 in Sect. 10 (page

26) and Proposition 13.3 in Sect. 13 (page 35). To have a simpler look at the statement we suggest to ignore cases 2(b)i and 2(b)ii at a first reading.

Lemma 9.1 (Contextual Normal Form Property)**.** Let $S\langle u\rangle$ be such that u is a useful normal form and S is a shallow context. Then either

(1) $u{\downarrow_S}$ is a β-normal form, or
(2) $S \neq \langle\cdot\rangle$ and there exists a shallow context S' such that
   (a) $u = S'\langle x\rangle$, and
   (b) $S\langle S'\rangle$ is the position of a useful $\mathtt{ls}$-redex of $S\langle u\rangle$, namely
      (i) *Relative Duplication*: $x{\downarrow_{S\langle S'\rangle}}$ has a β-redex, or
      (ii) *Relative Creation*: S' is applicative and $x{\downarrow_{S\langle S'\rangle}}$ is an abstraction.

Proof. Note that:

- If there exists S' as in case 2 then $S \neq \langle\cdot\rangle$, otherwise case 2(b)i or case 2(b)ii would imply a useful substitution redex in u, while u is a useful normal form by hypothesis.
- Cases 1 and 2 are mutually exclusive: if 2(b)i or 2(b)ii holds clearly $u{\downarrow_S}$ has a β-redex. We are only left to prove that one of the two always holds.

By induction on u. Cases:

(1) *Variable* $u = x$. If $x{\downarrow_S}$ is a β-normal form nothing remains to be shown. Otherwise, $x{\downarrow_S}$ has a β-redex and we are in case 2(b)i, setting $S' := \langle\cdot\rangle$ (giving $x{\downarrow_{S\langle S'\rangle}} = x{\downarrow_S}$).
(2) *Abstraction* $u = \lambda y.r$. Follows from the *i.h.* applied to r and $\lambda x.\langle\cdot\rangle$.
(3) *Application* $u = rp$. Note that $u{\downarrow_S} =_{5.6.2} r{\downarrow_S}p{\downarrow_S}$ and that r is a useful normal form, since u is. We can then apply the *i.h.* to r and $S\langle\langle\cdot\rangle p\rangle$. There are two cases:
   (a) $r{\downarrow_{S\langle\langle\cdot\rangle p\rangle}} = r{\downarrow_S}$ *is a normal form.* Sub-cases:
      (i) $r{\downarrow_S}$ *is an abstraction* $\lambda y.q$. This is the interesting inductive case, because it is the only one where the *i.h.* provides case 1 for r but the case ends proving case 2, actually 2(b)ii, for u. In the other cases of the proof (3(a)ii, 3(b), and 4), instead, the case of the statement provided by the *i.h.* is simply propagated *mutatis mutandis*.
      Note that r cannot have the form $L\langle\lambda y.s\rangle$, because otherwise u would not be $\to_{\mathrm{LOU}}$ normal. Then it follows that $r = L\langle x\rangle$ (as r cannot be an application). For the same reason, $r{\downarrow}$ cannot have the form $\lambda y.s$. Then $r{\downarrow} = z$ for some variable z (possibly $z = x$). Now take $S' := Lp$. Note that S' is applicative and that $x{\downarrow_{S\langle Lp\rangle}} =_{5.6.6} L\langle x\rangle{\downarrow_{S\langle\langle\cdot\rangle p\rangle}} = L\langle x\rangle{\downarrow_S} = r{\downarrow_S} = \lambda y.q$. So we are in case 2(b)ii and there is a useful $\to_{\mathtt{ls}}$-redex of position $S\langle S'\rangle$.
      (ii) r *is not an abstraction.* Note that $r{\downarrow_S}$ is neutral. Then the statement follows from the *i.h.* applied to p and $S\langle r\langle\cdot\rangle\rangle$. Indeed, if $p{\downarrow_{S\langle r\langle\cdot\rangle\rangle}} = p{\downarrow_S}$ is a β-normal form then $u{\downarrow_S} = r{\downarrow_S}p{\downarrow_S}$ is a β-normal form and we are in case 1. While if exists S'' such that $p = S''\langle x\rangle$ verifies case 2 (with respect to $S\langle r\langle\cdot\rangle\rangle$) then $S' := rS''$ verifies case 2 with respect to S.
   (b) *exists* S'' *such that* $r = S''\langle x\rangle$ *verifying case 2 of the statement with respect to* $S\langle\langle\cdot\rangle p\rangle$. Then case 2 holds for $S' := S''p$ with respect to S.
(4) *Substitution* $u = r[y{\leftarrow}p]$. Note that $u{\downarrow_S} = r[y{\leftarrow}p]{\downarrow_S} =_{5.6.6} r{\downarrow_{S\langle\langle\cdot\rangle[y\leftarrow p]\rangle}}$. So we can apply the *i.h.* to r and $S\langle\langle\cdot\rangle[y{\leftarrow}p]\rangle$, from which the statement follows.

□

Corollary 9.2 (Normal Form Property)**.** Let t be a useful normal form. Then $t{\downarrow}$ is a β-normal form.

Proof. Let us apply Lemma 9.1 to $S := \langle\cdot\rangle$ and $u := t$. Since $S = \langle\cdot\rangle$, case 2 cannot hold, so that $t{\downarrow_S} = t{\downarrow}$ is a β-normal form. □

At the end of the next section we will obtain the converse implication (Corollary 10.7.1), as a corollary of the strong projection property.

Let us close this section with a comment. The auxiliary lemma for the normal form property is of a technical nature. Actually, it is a strong improvement (inspired by [ASC15]) over the sequence of lemmas we provided in the technical report [ADL14b], that followed a different (worse) proof strategy. At a first reading the lemma looks very complex and it is legitimate to suspect that we did not fully understand the property we proved. We doubt, however, the existence of a much simpler proof, and believe that—despite the technical nature—our proof is compact. There seems to be an inherent difficulty given by the fact that useful reduction is a global notion, in the sense that it is not a rewriting rule closed by evaluation contexts, but it is defined by looking at the whole term at once. Its global character seems to prevent a simpler proof.

10. The Projection Property

We now turn to the proof of the projection property. We already know that a single shallow dB-step projects to a single β-step (Lemma 5.8). Therefore it remains to be shown that $\to_{\mathrm{LOU}}$ dB-steps project on LOβ steps. We do it contextually, in three steps:

(1) giving a (non-inductive) notion of LOU *context*, and proving that if a redex S is a $\to_{\mathrm{LOU}}$ redex then S is a LOU context.
(2) providing that LOU contexts admit a *inductive* formulation.
(3) proving that *inductive* LOU contexts unfold to inductive LOβ contexts, that is where LOβ steps take place.

As for the normal form property, the proof strategy is inspired by [ASC15], and improves the previous proof in the technical report [ADL14b].

10.1. LOU Contexts.

Remember that a term is *neutral* if it is β-normal and is not of the form $\lambda x.u$ (*i.e.* it is not an abstraction).

Definition 10.1 (LOU Contexts)**.** A context S is LOU if

(1) *Leftmost*: whenever $S = S'\langle tS''\rangle$ then $t{\downarrow_{S'}}$ is neutral, and
(2) *Outermost*: whenever $S = S'\langle \lambda x.S''\rangle$ then S' is not applicative.

The next lemma shows that $\to_{\mathrm{LOU}}$ redexes take place in LOU contexts. In the last sub-case the proof uses the generalized form of the normal form property (Lemma 9.1).

Lemma 10.2 (The Position of a $\to_{\mathrm{LOU}}$ Step is a LOU Context)**.** Let $S : t \to u$ be a useful step. If S is a $\to_{\mathrm{LOU}}$ step then S is LOU.

Proof. Properties in the definition of LOU contexts:

(1) *Outermost*: if $S = S'\langle\lambda x.S''\rangle$ then clearly S' is not applicative, otherwise there is a useful redex (the $\to_{\mathtt{dB}}$ redex involving $\lambda x.S''$) containing S, *i.e.* S is not the LOU redex, that is absurd.
(2) *Leftmost*: suppose that the leftmost property of LOU contexts is violated for S, and let $S = S'\langle rS''\rangle$ be such that $r{\downarrow_{S'}}$ is not neutral. We have that r is $\to_{\mathrm{LOU}}$-normal. Two cases:

(a) $r\downarrow_{S'}$ *is an abstraction.* Then S' is the position of a useful redex and S is not the LOU step, absurd.
(b) $r\downarrow_{S'}$ *contains a* β*-redex.* By the contextual normal form property (Lemma 9.1) there is a useful redex in t having its position in r, and so S is not the position of the $\to_{\mathrm{LOU}}$-redex, absurd. □

10.2. **Inductive LOU Contexts.** We introduce the alternative characterization of LOU contexts, avoiding relative unfoldings. We call it *inductive* because it follows the structure of the context S, even if in the last clause the hypothesis might be syntactically bigger than the conclusion. Something, however, always decreases: in the last clause it is the number of ES. The lemma that follows is indeed proved by induction over the number of ES in S and S itself.

Definition 10.3 (Inductive LOU Contexts)**.** A context S is *inductively LOU* (or iLOU), if a judgment about it can be derived by using the following inductive rules:

$$\frac{}{\langle\cdot\rangle \text{ is iLOU}}\ (\text{ax-iLOU}) \qquad \frac{S \text{ is iLOU} \qquad S \neq L\langle\lambda x.S'\rangle}{St \text{ is iLOU}}\ (@\text{l-iLOU})$$

$$\frac{S \text{ is iLOU}}{\lambda x.S \text{ is iLOU}}\ (\lambda\text{-iLOU}) \qquad \frac{t\downarrow \text{ is neutral} \qquad S \text{ is iLOU}}{tS \text{ is iLOU}}\ (@\text{r-iLOU})$$

$$\frac{S\{x\leftarrow t\downarrow\} \text{ is iLOU}}{S[x\leftarrow t] \text{ is iLOU}}\ (\text{ES-iLOU})$$

Let us show that LOU contexts are inductive LOU contexts.

Lemma 10.4 (LOU Contexts are iLOU)**.** Let S be a context. If S is LOU then S is inductively LOU.

Proof. By lexicographic induction on $(|S|_{[\cdot]}, S)$, where $|S|_{[\cdot]}$ is the number of ES in S. Cases of S:
(1) *Empty* $S = \langle\cdot\rangle$. Immediate.
(2) *Abstraction* $S = \lambda x.S'$. By *i.h.* S' is iLOU. Then S is iLOU by rule (λ-iLOU).
(3) *Left Application* $S = S'u$. By *i.h.*, S' is iLOU. Note that $S' \neq L\langle\lambda x.S''\rangle$ otherwise the outermost property of LOU contexts would be violated, since S' appears in an applicative context. Then S is iLOU by rule (@l-iLOU).
(4) *Right Application* $S = uS'$. Then S' is LOU and so S' is iLOU by *i.h.*. By the leftmost property of LOU contexts $u\downarrow = u\downarrow_{\langle\cdot\rangle}$ is neutral. Then S is iLOU by rule (@r-iLOU).
(5) *Substitution* $S = S'[x\leftarrow u]$. We prove that $S'\{x\leftarrow u\downarrow\}$ is LOU and obtain that S is iLOU by applying the *i.h.* (first component decreases) and rule (ES-iLOU). There are two conditions to check:
   (a) *Outermost*: consider $S'\{x\leftarrow u\downarrow\} = S''\langle\lambda y.S'''\rangle$. Note that the abstraction λy comes from an abstraction of S' to which $\{x\leftarrow u\downarrow\}$ has been applied, because $u\downarrow$ cannot contain a context hole. Then $S' = S_2\langle\lambda y.S_3\rangle$ with $S_2\{x\leftarrow u\downarrow\} = S''$ and $S_3\{x\leftarrow u\downarrow\} = S'''$. By hypothesis S is LOU and so $S_2[x\leftarrow u\downarrow]$ is not applicative. Then S_2 and thus $S_2\{x\leftarrow u\downarrow\}$ are not applicative.
   (b) *Leftmost*: consider $S'\{x\leftarrow u\downarrow\} = S''\langle rS'''\rangle$. Note that the application rS''' comes from an application of S' to which $\{x\leftarrow u\downarrow\}$ has been applied, because $u\downarrow$ cannot

contain a context hole. Then $S' = S_2\langle r' S_3\rangle$ with $S_2\{x{\leftarrow}u{\downarrow}\} = S''$, $r'\{x{\leftarrow}u{\downarrow}\} = r$, and $S_3\{x{\leftarrow}u{\downarrow}\} = S'''$. We have to show that $r{\downarrow_{S''}}$ is neutral. Note that

$$r{\downarrow_{S''}} = r'\{x{\leftarrow}u{\downarrow}\}{\downarrow_{S_2\{x\leftarrow u\downarrow\}}} =_{5.6.6} r'{\downarrow_{S_2[x\leftarrow u]}}$$

Since $S = S_2\langle r' S_3\rangle[x{\leftarrow}u]$ is LOU by hypothesis, we obtain that $r'{\downarrow_{S_2[x\leftarrow u]}}$ is neutral. □

10.3. **Unfolding LOU steps.** Finally, we show that inductive LOU contexts unfold to inductive LOβ contexts (Definition 3.3, page 12).

Lemma 10.5 (iLOU unfolds to iLOβ)**.** If S is an iLOU context then $S{\downarrow}$ is a iLOβ context.

Proof. By lexicographic induction on $(|S|_{[\cdot]}, S)$. Cases of S.

(1) *Empty* $S = \langle\cdot\rangle$. Then $S{\downarrow} = \langle\cdot\rangle{\downarrow} = \langle\cdot\rangle$ is iLOβ.
(2) *Abstraction* $S = \lambda x.S'$. By definition of unfolding $S{\downarrow} = \lambda x.S'{\downarrow}$, and by the iLOU hypothesis S' is iLOU. Then by *i.h.* (second component), $S'{\downarrow}$ is iLOβ, and so $S{\downarrow}$ is iLOβ by rule (λ-iLOβ).
(3) *Left Application* $S = S't$. By definition of unfolding, $S{\downarrow} = S'{\downarrow}t{\downarrow}$, and by the iLOU hypothesis S' is iLOU. Then by *i.h.* (second component), $S'{\downarrow}$ is iLOβ. Moreover, $S' \neq L\langle\lambda x.S''\rangle$, that implies $S' = L\langle S''u\rangle$, $S' = L\langle uS''\rangle$, or $S' = L$. In all such cases we obtain $S'{\downarrow} \neq \lambda x.C$. Therefore, $S{\downarrow}$ is iLOβ by rule (@l-iLOβ).
(4) *Right Application* $S = tS'$. By definition of unfolding $S{\downarrow} = t{\downarrow}S'{\downarrow}$, and by the iLOU hypothesis S' is iLOU and $t{\downarrow}$ is neutral. Then by *i.h.* (second component), S' is iLOβ, and so S is iLOβ by rule (@r-iLOβ).
(5) *Substitution* $S = S'[x{\leftarrow}t]$. By definition of unfolding $S{\downarrow} = S'{\downarrow}\{x{\leftarrow}t{\downarrow}\}$, and by the iLOU hypothesis $S'\{x{\leftarrow}t{\downarrow}\}$ iLOU. Then by *i.h.* (first component) $S'\{x{\leftarrow}t{\downarrow}\}{\downarrow}$ is iLOβ, that is equivalent to the statement because $S'{\downarrow}\{x{\leftarrow}t{\downarrow}\} =_{5.6.2} S'\{x{\leftarrow}t{\downarrow}\}{\downarrow}$. □

The projection property of LOU $\to_{\mathtt{dB}}$ steps now follows easily:

Theorem 10.6 (Strong Projection Property)**.** Let t be a LSC term and $t \to_{\mathrm{LOU}} u$ a $\to_{\mathtt{dB}}$ step. Then $t{\downarrow} \to_{\mathrm{LO}\beta} u{\downarrow}$.

Proof. $\to_{\mathrm{LOU}}$-steps take place in LOU contexts (Lemma 10.2), LOU contexts are inductive LOU contexts (Lemma 10.4), that unfold to iLOβ contexts (Lemma 10.5), which is where $\to_{\mathrm{LO}\beta}$ steps take place (Lemma 3.4). □

The following corollary shows that in fact for our high-level implementation system the converse statements of the normal form and projection properties are also valid, even if they are not needed for the invariance result.

Corollary 10.7.

(1) *Converse Normal Form*: if $t{\downarrow}$ is a β-normal form then t is a useful normal form.
(2) *Converse Projection*: if $t{\downarrow} \to_{\mathrm{LO}\beta} u$ then there exists r such that $t \to_{\mathrm{LOU}} r$ with $r{\downarrow} = t{\downarrow}$ or $r{\downarrow} = u$.

Proof.

(1) Suppose that t has a useful redex. By the strong projection property (Theorem 10.6) the $\to_{\mathrm{LOU}}$ redex in t cannot be a $\mathtt{dB}$-redex, otherwise $t{\downarrow}$ is not β-normal. Similarly, by Lemma 6.2 it cannot be a $\mathtt{ls}$-redex. Absurd.

(2) By the normal form property (Corollary 9.2) t has a useful redex, otherwise $t_{\downarrow}$ is β-normal, that is absurd. Then the statement follows from the strong projection property (Theorem 10.6) or from Lemma 5.7. □

For the sake of completeness, let us also point out that the converse statements of Lemma 10.2 (*i.e.* useful steps taking place in LOU contexts are $\to_{\mathrm{LOU}}$ steps) and Lemma 10.5 (inductive LOU contexts are LOU contexts) are provable. We omitted them to lighten the technical development of the paper.

11. Standard Derivations and the Subterm Property

Here we introduce standard derivations and show that they have the subterm property.

11.1. **Standard derivation.** They are defined on top of the concept of residual of a redex. For the sake of readability, we use the concept of residual without formally defining it (see [ABKL14] for details).

Definition 11.1 (Standard Derivation)**.** A derivation $\rho : S_1; \ldots; S_n$ is *standard* if S_i is not the residual of a LSC redex $S' \prec_{LO} S_j$ for every $i \in \{2, \ldots, n\}$ and $j < i$.

The same definition where terms are ordinary λ-terms and redexes are β-redexes gives the ordinary notion of standard derivation in the theory of λ-calculus.

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 (Definition 4.2, page 14) 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. We have the following expected result.

Theorem 11.2 ([ABKL14])**.** LO derivations (of the LSC) are standard.

The *subterm property* states that at any point of a derivation $\rho : t \to^* u$ only subterms of the initial term t are duplicated. Duplicable subterms are identified by *boxes*, and we need a technical lemma about them.

11.2. **Boxes, Invariants, and Subterms.** A *box* is the argument of an application or the content of an explicit substitution. In the graphical representation of λ-terms xwith ES, our boxes correspond to explicit boxes for promotions.

Definition 11.3 (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 a general context (*i.e.* not necessarily shallow):

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

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

In the simple case of linear head reduction [ADL12], the subterm property follows from the invariant that evaluation never substitutes in box-subterms, and so ES—that are boxes—contain and thus substitute only subterms of the initial term. For linear LO reduction, and more generally for standard derivations in the LSC, the property is a bit more subtle to establish. Substitutions can in fact act inside boxes, but a more general invariant still holds: whenever a standard derivation substitutes/evaluates in a box subterm u, then u will no longer be substituted. This is a consequence of selecting redexes according to $\prec_{LO}$. Actually, the invariant is stated the other way around, saying that the boxes on the right—that are those involved in later duplications—are subterms of the initial term.

Lemma 11.4 (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_k be the last contracted redex, $k \geq 0$, and $B \prec_p t_{k+1}$ be a box context such that $S_k \prec_{LO} B$. Then the box subterm u identified by B (*i.e.* such that $t_{k+1} = B\langle u\rangle$) is a box subterm of t_0.

Proof. By induction on k. If $k = 0$ the statement trivially holds. Otherwise, consider the redex $S_{k-1} : t_{k-1} \to t_k$. By *i.h.* the statement holds wrt box contexts $B \prec_p t_k$ such that $S_{k-1} \prec_{LO} B$. The proof analyses the position of S_k with respect to the position S_{k-1}, often distinguishing between the left-to-right $\prec_L$ and the outside-in $\prec_O$ suborders of $\prec_{LO}$. Cases:

(1) S_k *is equal to* S_{k-1}. Clearly if $S_{k-1} = S_k \prec_L B$ then the statement holds because of the *i.h.* (reduction does not affect boxes on the right of the hole of the position). We need to check the box contexts such that $S_k \prec_O B$. Note that S_{k-1} cannot be a $\to_{\mathtt{dB}}$ redex, because if $t_{k-1} = S_{k-1}\langle L\langle \lambda x.r\rangle p\rangle \to_{\mathtt{dB}} S_{k-1}\langle L\langle r[x\leftarrow p]\rangle\rangle = t_k$ then $S_{k-1} = S_k$ is not the position of any redex in t_k. Hence, S_{k-1} is a $\to_{\mathtt{ls}}$ step and there are two cases. If it substitutes a:
   (a) *Variable*, *i.e.* the sequence of steps $S_{k-1}; S_k$ is
   $$t_{k-1} = S_{k-1}\langle x\rangle \to_{\mathtt{ls}} S_{k-1}\langle y\rangle \to_{\mathtt{ls}} S_{k-1}\langle r\rangle = t_{k+1}$$
   Then all box subterms of r trace back to box subterms that also appear on the right of S_{k-1} in t_k and so they are box subterms of t_0 by *i.h.*
   (b) $\mathtt{dB}$-*redex*, *i.e.* the sequence of steps $S_{k-1}; S_k$ is
   $$t_{k-1} = S_{k-1}\langle x\rangle \to_{\mathtt{ls}} S_{k-1}\langle L\langle \lambda y.r\rangle p\rangle \to_{\mathtt{ls}} S_{k-1}\langle L\langle r[y\leftarrow p]\rangle\rangle = t_{k+1}$$
   Then all box subterms of $L\langle r[y\leftarrow p]\rangle$ trace back to box subterms of $L\langle \lambda y.r\rangle p$, hence they are in t_k, and so they are box subterms of t_0 by *i.h.*

(2) S_{k-1} *is internal to* S_k, *i.e.* $S_k \prec_O S_{k-1}$ and $S_k \neq S_{k-1}$. This case is only possible if S_k has been created *upwards* by S_{k-1}, otherwise the derivation would not be $\prec_{LO}$-standard. There are only two possible cases of creations *upwards*:
   (a) $\mathtt{dB}$ creates $\mathtt{dB}$, *i.e.* S_{k-1} is
   $$t_{k-1} = S_{k-1}\langle L\langle \lambda y.L'\langle \lambda z.r\rangle\rangle p\rangle \to_{\mathtt{dB}} S_{k-1}\langle L\langle L'\langle \lambda z.r\rangle[y\leftarrow p]\rangle\rangle = t_k$$
   and S_{k-1} is applicative, that is $S_{k-1} = S_k\langle L''\langle\cdot\rangle q\rangle$, so that S_k is
   $$t_k = S_k\langle L''\langle L\langle L'\langle \lambda z.r\rangle[y\leftarrow p]\rangle\rangle q\rangle \to_{\mathtt{dB}} S_k\langle L''\langle L\langle L'\langle r[z\leftarrow q]\rangle[y\leftarrow p]\rangle\rangle\rangle = t_{k+1}$$
   The box subterms of L'' and q (including q itself) are box subterms of t_k with box context B such that $S_{k-1} \prec_L B$ and so they are box subterms of t_0 by *i.h.*The other box subterms of $L''\langle L\langle L'\langle r[z\leftarrow q]\rangle[y\leftarrow p]\rangle\rangle$ are instead box subterms of $L\langle \lambda y.L'\langle \lambda z.r\rangle\rangle p$, *i.e.* of box context B such that $S_{k-1} \prec_O B$, and so they are box subterms of t_0 by *i.h.*
   (b) $\mathtt{ls}$ creates $\mathtt{dB}$, *i.e.* S_{k-1} is
   $$t_{k-1} = S_{k-1}\langle x\rangle \to_{\mathtt{dB}} S_{k-1}\langle L\langle \lambda y.r\rangle\rangle = t_k$$
   and S_{k-1} is applicative, *i.e.* $S_{k-1} = S_k\langle L'\langle\cdot\rangle p\rangle$ so that S_k is

$$S_k\langle L'\langle L\langle \lambda y.r\rangle\rangle p\rangle \to_{\mathtt{dB}} S_k\langle L'\langle L\langle r[y\leftarrow p]\rangle\rangle\rangle$$

The box subterms of L' and p (including p itself) are box subterms of the ending term of S_{k-1} whose box context B is $S_{k-1} \prec_L B$ and so they are box subterms of t_0 by *i.h.*The other box subterms of $L'\langle L\langle r[y\leftarrow p]\rangle\rangle$ are also box subterms of $L\langle \lambda y.L'\langle \lambda z.r\rangle\rangle p$ and so they are box subterms of t_0 by *i.h.*

(3) S_k *is internal to* S_{k-1}, *i.e.* $S_{k-1} \prec_O S_k$ and $S_k \neq S_{k-1}$. Cases of S_{k-1}:
   (a) `dB`-*step*, *i.e.* S_{k-1} is
   $$t_{k-1} = S_{k-1}\langle L\langle \lambda x.r\rangle p\rangle \to_{\mathtt{dB}} S_{k-1}\langle L\langle r[x\leftarrow p]\rangle\rangle = t_k$$
   Then the hole of S_k is inside $L\langle r[x\leftarrow p]\rangle$. Box subterms identified by a box context B such that $S_k \prec_L B$ in t_{k+1} are also box subterms of t_k, and so the statement follows from the *i.h.*For box subterms identified by a box context B of t_{k+1} such that $S_k \prec_O B$ we have to analyze S_k. Suppose that S_k is a:
      - `dB`-*step*. Note that in a root `dB`-step (*i.e.* at top-level) all the box subterms of the reduct are box subterms of the redex. In this case the redex is contained in $L\langle r[x\leftarrow p]\rangle$ and so by *i.h.* all such box subterms are box subterms of t_0.
      - `ls`-*step*, *i.e.* S_k has the form $t_k = S_k\langle x\rangle \to_{\mathtt{ls}} S_k\langle q\rangle = t_{k+1}$. In t_k, q is identified by a box context B such that $S_k \prec_L B$. From $S_{k-1} \prec_{LO} S_k$ we obtain $S_{k-1} \prec_{LO} B$ and so all box subterms of q are box subterms of t_0 by *i.h.*
   (b) `ls`-*step*: S_{k-1} is $t_{k-1} = S_{k-1}\langle x\rangle \to_{\mathtt{dB}} S_{k-1}\langle q\rangle = t_k$. It is analogous to the `dB`-case: S_k takes place inside q, whose box subterms are box subterms of t_0, by *i.h.*If S_k is a `dB`-redex then it only rearranges constructors in q without changing box subterms, otherwise it substitutes something coming from a substitution that is on the right of S_{k-1} and so whose box subterms are box subterms of t_0 by *i.h.*

(4) S_k *is on the left of* S_{k-1}, *i.e.* $S_{k-1} \prec_L S_k$. For B such that $S_k \prec_L B$, the statement follows from the *i.h.*, because there is a box context B' in t_k such that $S_{k-1} \prec_L B'$ and identifying the same box subterm of B. For B such that $S_k \prec_O B$, we reason as in case 3a. □

From the invariant, one easily obtains the subterm property.

Corollary 11.5 (Subterm)**.** Let $\rho : t \to^k u$ be a standard derivation. Then every $\to_{\mathtt{ls}}$-step in ρ duplicates a subterm of t.

Proof. By induction on k. If $k = 0$ the statement is evidently true. Otherwise, by *i.h.* in $\rho : t \to^{k-1} r$ every $\to_{\mathtt{ls}}$-step duplicated a subterm of t. If the next step is a `dB`-step the statement holds, otherwise it is a `ls`-step that by Lemma 11.4 duplicates a subterm of u which is a box subterm, and so a subterm, of t. □

11.3. **Technical Digression: Shallowness and Standardization.** In [ABKL14] it is shown 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 standardization 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 standardization 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 S is contained in a substitution, and every substitution is involved in an outer redex S');

(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 S cannot turn a non-shallow redex S' such that $S' \prec_{LO} S$ into a shallow redex).

Moreover, the only redex swaps we will consider (Lemma 12.2) will produce shallow residuals.

11.4. **Shallow Terms.** 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.

Lemma 11.6 (Shallow Invariant)**.** Let t be a λ-term and $\rho : t \to^k u$ be a standard derivation. Then u is a shallow term.

Proof. By induction on k. If $k = 0$, the statement is evidently true. Otherwise, by *i.h.* every explicit substitution in r, where $\rho : t \to^{k-1} r$, contains a λ-term. We distinguish the two cases concerning the sort of the next step $r \to u$:

(1) `ls`-*step*. By the subterm property and the fact that t has no ES, the step duplicates a term without substitutions, and—since reduction is shallow—it does not put the duplicated term in a substitution. Therefore, every substitution of u corresponds uniquely to a substitution of r with the same content. Then u is a shallow term by *i.h.*

(2) `dB`-*step*. It is easily seen that the argument of the `dB`-step is on the right of the previous step, so that by Lemma 11.4 it contains a (box) subterm of t. Then, the substitution created by the `dB`-step contains a subterm of t, that is an ordinary λ-term by hypothesis. The step does not affect any other substitution, because reduction is shallow, and so u is a shallow term. □

In this paper we state many properties relative to derivations whose initial term is a λ-term. The shallow invariant essentially means that all these properties may be generalized to (standard) derivations whose initial term is shallow. There is, however, a subtlety that justifies our formulation with respect to λ-terms. Lemma 11.6, indeed, does not hold if one simply assume that t is a shallow term. Consider for instance $(\lambda x.x)(y[y\leftarrow z])$, that is shallow and that reduces in one step (thus via a standard derivation) to $x[x\leftarrow y[y\leftarrow z]]$, which is not shallow. The subtlety is that the position S of the first step of the standard derivation has to be a $\prec_{LO}$-majorant of the position of any ES in the term. For the sake of simplicity, we prefered to assume that the initial term has no ES.

Note also that this Lemma 11.6 is the only point of this section relying on the assumption that reduction is shallow (the hypothesis of the derivation being standard is also necessary, consider $(\lambda x.x)((\lambda y.y)z) \to_{\mathtt{dB}} (\lambda x.x)(y[y\leftarrow z]) \to_{\mathtt{dB}} x[x\leftarrow y[y\leftarrow z]]$).

12. LOU Derivations are Standard

Notation: to avoid ambiguities, in this section we use R, P, Q for redexes, R', P', Q' for their residuals, and S, S', S'' for shallow contexts.

LO derivations are standard (Theorem 11.2), and this is expected. A priori, instead, LOU derivations may not be standard, if the reduction of a useful redex R could turn a

useless redex $P \prec_{LO} R$ 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.

We first need to recall two properties of the standardization order $\prec_{LO}$ relative to residuals, called *linearity* and *enclave*. They are two of the axioms of the axiomatic theory of standardization developed by Melliès in his PhD thesis [Mel96], that in turn is a refinement of a previous axiomatization by Gonthier, Lévy, and Melliès [GLM92] (that did not include the enclave axiom). The axioms of Melliès' axiomatic theory have been proved to hold for the LSC by Accattoli, Bonelli, Kesner, and Lombardi in [ABKL14]. The two properties essentially express that if $R \prec_{LO} P$ then P cannot act on R. Their precise formulation follows.

Lemma 12.1 ([ABKL14])**.** If $R \prec_{LO} P$ then
(1) *Linearity*: R has a unique residual R' after P;
(2) *Enclave*: two cases
   (a) *Creation*: if P creates a redex Q then $R' \prec_{LO} Q$;
   (b) *Nesting*: If $P \prec_{LO} Q$ and Q' is a residual of Q after P then $R' \prec_{LO} Q'$.

Now we can prove the key lemma of the section.

Lemma 12.2 (Useless Persistence)**.** Let $R : t \to_{\mathtt{ls}} u$ be a useless redex, $P : t \to p$ be a useful redex such that $R \prec_{LO} P$, and R' be the unique residual of R after P (uniqueness follows from the just recalled property of *linearity* of $\prec_{LO}$). Then
(1) R' is shallow and useless;
(2) if P is LOU and Q is the LOU redex in p then $R' \prec_{LO} Q$.

Proof.
(1) Let $R : S'\langle S\langle x\rangle[x{\leftarrow}r]\rangle \to_{\mathtt{ls}} S'\langle S\langle r\rangle[x{\leftarrow}r]\rangle$. According to Definition 6.1 (page 19), a `ls`-redex is useless when it is not useful. Then, uselessness of R implies that $r{\downarrow_{S'}}$ is a normal λ-term (otherwise the *relative duplication* clause in the definition of useful redexes would hold) and if $r{\downarrow_{S'}}$ is an abstraction then $S'\langle S[x{\leftarrow}r]\rangle$ is not an applicative context (otherwise *relative creation* would hold).

   Note that `ls`-steps cannot change the useless nature of R. To change it, in our case, they should be able to change the abstraction/normal nature of $r{\downarrow_{S'}}$ or to change the applicative nature of $S'\langle S[x{\leftarrow}r]\rangle$, but both changes are impossible: unfoldings, and thus $r{\downarrow_{S'}}$, cannot be affected by `ls`-steps (formally, an omitted generalization of Lemma 5.6 is required), and `ls`-steps cannot provide/remove arguments to/from context holes. So, in the following we suppose that P is a `dB`-redex.

   By induction on S', the external context of R. Cases:
   (a) *Empty context* $\langle\cdot\rangle$. Consider P, that necessarily takes place in the context S,
   $$P : S\langle x\rangle[x{\leftarrow}r] \to S'\langle x\rangle[x{\leftarrow}r]$$
   The only way in which the residual $R' : S'\langle x\rangle[x{\leftarrow}r] \to_{\mathtt{ls}} S'\langle r\rangle[x{\leftarrow}r]$ of R can be useful is if P turned the non-applicative context S into an applicative context S', assuming that $r{\downarrow}$ is an abstraction. It is easily seen that this is possible only if $P \prec_{LO} R$, against hypothesis. Namely, only if $S = S''\langle L\langle\lambda y.L'\rangle p\rangle$ and S'' is applicative, so that P is
   $$S''\langle L\langle\lambda y.L'\langle x\rangle\rangle p\rangle[x{\leftarrow}r] \to_{\mathtt{dB}} S''\langle L\langle L'\langle x\rangle[y{\leftarrow}p]\rangle\rangle[x{\leftarrow}r]$$
   with $S' = S''\langle L\langle L'[y{\leftarrow}p]\rangle\rangle$ applicative context.
   (b) *Inductive cases*:
      (i) *Abstraction, i.e.* $S' = \lambda y.S''$. Both redexes R and P take place under the outermost abstraction, so the statement follows from the *i.h.*

(ii) *Left of an application, i.e.* $S' = S''q$. Note that P cannot be the eventual root dB-redex (*i.e.* if S'' is of the form $L\langle\lambda y.S'''\rangle$ then P is not the dB-redex involving λy and q), because this would contradict $R \prec_{LO} P$. If the redex P takes place in $S''\langle S\langle x\rangle[x\leftarrow r]\rangle$ then we use the *i.h.*Otherwise P takes place in q, the two redexes are disjoint, and commute. Evidently, the residual R' of R after P is still shallow and useless.

(iii) *Right of an application, i.e.* $S' = qS''$. Since $R \prec_{LO} P$, P necessarily takes place in S'', and the statement follows from the *i.h.*

(iv) *Substitution, i.e.* $S' = S''[y\leftarrow q]$. Both redexes R and P take place under the outermost explicit substitution $[y\leftarrow q]$, so the statement follows from the *i.h.*

(2) Assume that P is LOU. By Point 1, the unique residual R' of any useless redex $R \prec_{LO} P$ is useless, so that the eventual next LOU redex Q either has been created by P or it is the residual of a redex Q^* such that $P \prec_{LO} Q^*$. The enclave property guarantees that $R' \prec_{LO} Q$. □

Now an easy iterated application of the previous lemma shows that LOU derivations are standard.

Proposition 12.3. Every LOU derivation is standard.

Proof. By induction on the length k of a LOU derivation ρ. If $k = 0$ then the statement trivially holds. If $k > 0$ then ρ writes as $\tau; R$ where τ by *i.h.* is standard. Let τ be $R_1; \dots R_k$ and $R_i : t_i \to t_{i+1}$ with $i \in \{1, \dots, k\}$. If $\tau; R$ is not standard there is a term t_i and a redex P of t_i such that
(1) R is a residual of P after $R_i; \dots; R_k$;
(2) $P \prec_{LO} R_i$.
Since R_i is LOU, P is useless. Then, iterating the application of Lemma 12.2 to the sequence $R_i; \dots; R_k$, we obtain that R is useless, which is absurd. Then $\rho = \tau; R$ is standard. □

We conclude by applying Corollary 11.5.

Corollary 12.4 (Subterm)**.** LOU derivations have the subterm property.

13. The Local Boundedness Property, via Outside-In Derivations

In this section we show that LOU derivations have the local boundedness property. We introduce yet another abstract property, the notion of *outside-in derivation*, and show that together with the subterm property it implies local boundedness. We conclude by showing that LOU derivations are outside-in.

Definition 13.1 (Outside-In Derivation)**.** Two ls-steps $t \to_{\mathtt{ls}} u \to_{\mathtt{ls}} r$ are *outside-in* if the second one substitutes on the subterm substituted by the first one, *i.e.* if there exist S and S' such that the two steps have the compact form $S\langle x\rangle \to_{\mathtt{ls}} S\langle S'\langle y\rangle\rangle \to_{\mathtt{ls}} S\langle S'\langle u\rangle\rangle$. A derivation is outside-in if any two consecutive substitution steps are outside-in.

For instance, the first of the following two sequences of steps is outside-in 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 outside-in derivations ensure the local boundedness property because
(1) no substitution can be used twice in a outside-in sequence $u \to^k_{\mathtt{ls}} r$, and
(2) $\to_{\mathtt{ls}}$ steps do not change the number of substitutions, because they duplicate terms without ES by the subterm property.

Therefore, k is necessarily bounded by the number of ES in u—noted $|u|_{[\cdot]}$—which in turn is bounded by the number of preceding $\mathtt{dB}$-steps. The next lemma formalizes this idea.

Lemma 13.2 (Subterm + Outside-In ⇒ Local Boundedness)**.** Let t be a λ-term, $\rho : t \to^n u \to^k_{\mathtt{ls}} r$ be a derivation with the subterm property and whose suffix $u \to^k_{\mathtt{ls}} r$ is outside-in. Then $k \leq |\rho|_{\mathtt{dB}}$.

Proof. Let $u = u_0 \to_{\mathtt{ls}} u_1 \to_{\mathtt{ls}} \dots \to_{\mathtt{ls}} u_k = r$ be the outside-in suffix of ρ and $u_i \to_{\mathtt{ls}} u_{i+1}$ one of its steps, for $i \in \{0, \dots, k-2\}$. Let us use S_i for the external context of the step, *i.e.* the context such that $u_i = S_i\langle S'\langle x\rangle[x\leftarrow p]\rangle \to_{\mathtt{ls}} S_i\langle S'\langle p\rangle[x\leftarrow p]\rangle = u_{i+1}$. The following outside-in step $u_{i+1} \to_{\mathtt{ls}} u_{i+2}$ substitutes on the substituted occurrence of p. By the subterm property, p is a subterm of t and so it is has no ES. Then the ES acting in $u_{i+1} \to_{\mathtt{ls}} u_{i+2}$ is on the right of $[x\leftarrow p]$, *i.e.* the external context S_{i+1} is a prefix of S_i, in symbols $S_{i+1} \prec_O S_i$. Since the derivation $u_0 \to_{\mathtt{ls}} u_1 \to_{\mathtt{ls}} \dots \to_{\mathtt{ls}} u_k$ is outside-in we obtain a sequence $S_k \prec_O S_{k-1} \prec_O \dots \prec_O S_0$ of contexts of u. In particular, every S_i corresponds to a different explicit substitution in u, and so $k \leq |u|_{[\cdot]}$. Now, we show that $|u|_{[\cdot]} = |\rho|_{\mathtt{dB}}$, that will conclude the proof. The subterm property has also another consequence. Given that only ordinary λ-terms are duplicated, no explicit substitution constructor is ever duplicated by $\mathtt{ls}$-steps in ρ: if $r \to_{\mathtt{ls}} p$ is a step of ρ then $|r|_{[\cdot]} = |p|_{[\cdot]}$. Every $\mathtt{dB}$-step, instead, introduces an explicit substitution, *i.e.* $|u|_{[\cdot]} = |\rho|_{\mathtt{dB}}$. □

Since we know that LOU derivations have the subterm property, (Corollary 12.4), what remains to be shown is that they are outside-in.

Proposition 13.3. LOU derivations are outside-in.

Note that while in Lemma 13.2 the hypothesis that the initial term of the derivation is a λ-term (relaxable to a shallow term) is essential, here—as well as for the the subterm property—such an hypothesis is not needed. Note also that the last subcase of the proof uses the generalized form of the normal form property (Lemma 9.1).

Proof. We prove the following implication: if the reduction step $S\langle x\rangle \to_{\mathtt{ls}} S\langle u\rangle$ is LOU, and the LOU redex S' in $S\langle u\rangle$ is a $\mathtt{ls}$-redex then S and S' are outside-in, *i.e.* $S \prec_O S'$ or $S = S'$. Two cases, depending on *why* the reduction step $S\langle x\rangle \to_{\mathtt{ls}} S\langle u\rangle$ is useful:
(1) *Relative Creation*, *i.e.* S is applicative and $u\downarrow_S$ is an abstraction. Two sub-cases:
   (a) *u is an abstraction (in a substitution context).* Then the LOU redex in $S\langle u\rangle$ is the $\mathtt{dB}$-redex having u as abstraction, and there is nothing to prove (because S' is not a $\mathtt{ls}$-redex).

(b) *u is not an abstraction.* Then it must a variable z (because it is a λ-term), and $z{\downarrow_S}$ is an abstraction. But then $S\langle u\rangle$ is simply $S\langle z\rangle$ and the given occurrence of z marks another useful substitution redex, *i.e.* $S' = S$, that is the LOU redex because S already was the position of the LOU redex at the preceding step.

(2) *Relative Duplication, i.e.* $u{\downarrow_S}$ is not an abstraction or S is not applicative, but $u{\downarrow_S}$ contains a β-redex. Two sub-cases:
   (a) *u contains a useful redex S'.* Then the position of the LOU redex S'' in $S\langle u\rangle$ (that is not necessarily $S\langle S'\rangle$) is in u. Two cases:
      (i) *S'' is a* `dB`*-redex.* Then there is nothing to prove, because the LOU redex is not a `ls`-redex.
      (ii) *S'' is a* `ls`*-redex.* Then the two steps are outside-in, because S is a prefix of S''.
   (b) *u is a useful normal form.* Since $u{\downarrow_S}$ does contain a β-redex, we can apply the contextual normal form property (Lemma 9.1) and obtain that there exists a useful `ls`-redex S' in $S\langle u\rangle$ such that $S \prec_O S'$. Then the $\prec_{LO}$-minimum of these redexes is the LOU redex in $S\langle u\rangle$, and S and S' are outside-in as redexes. □

Corollary 13.4 (Local Boundedness Property)**.** LOU derivations (starting on λ-terms) have the local boundedness property.

Proof. By Corollary 12.4 LOU derivations have the subterm property and by Proposition 13.3 they are outside-in. The initial term is a λ-term, and so Lemma 13.2 (subterm + outside-in $\Rightarrow$ local boundedness) provides the local boundedness property. □

At this point, we proved all the required properties for the implementation theorems but for the selection property for LOU derivations, addressed by the next section.

14. 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 efficiently mechanizable, *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 of 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 test can be done in time polynomial in (the size of) t and S, then firing *a single* reduction step can cause an exponential blowup!

Our algorithm consists in the simultaneous computation of four 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-tuples 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 $f(t_1{\downarrow}, \ldots, t_n{\downarrow})$.

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

Figure 1: Computing g.

- *Efficiently computable in compact form relative 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 such a 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 [ADL12], *alpha* is shown to be efficiently computable in compact form, via a dynamic programming algorithm $\mathcal{B}_{alpha}$ taking as 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, permits 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 [ADL12].

Each one of the functions of our interest takes 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 we need 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 that have a free occurrence 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 relative 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 interrelated (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

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

Figure 2: Computing g in compact form.

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

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

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 14.1. The algorithms $\mathcal{A}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ are all correct: for every λ-term t, for every LSC term u and for every context S, we have

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

Proof Sketch.
(1) The equation $\mathcal{A}_g(t) = g(t)$ can be proved by induction on the structure of t. An interesting case:
   - If $t = ur$, then we know that:

$$\mathcal{A}_g(ur) = (\mathsf{app}, b_u \vee b_r \vee (n_u = \mathsf{lam}), V_u \cup V_r \cup \{x \mid n_u = \mathsf{var}(x)\}, W_u \cup W_r)$$
$$\text{where } \mathcal{A}_g(u) = (n_u, b_u, V_u, W_u) \text{ and } \mathcal{A}_g(r) = (n_r, b_r, V_r, W_r);$$

     Now, first of all observe that $redex(t) = \mathsf{true}$ if and only if there is a redex in u or a redex in r or if u is a λ-abstraction. Moreover, the variables occurring in applicative position in t are those occurring in applicative position in either u or in r or x, if u is x itself. Similarly, the variables occurring free in t are simply those occurring free in either u or in r. The thesis can be synthesized easily from the inductive hypothesis.
(2) The equation $\mathcal{B}_g(u) = g(u{\downarrow})$ can be proved by induction on the structure of u, using the correctness of $\mathcal{A}$.
(3) The equation $\mathcal{C}_g(u, S) = g(u{\downarrow_S})$ can be proved by induction on the structure of S, using the correctness of $\mathcal{B}$.

This concludes the proof. □

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 14.2 (Selection Property)**.** The algorithms $\mathcal{A}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ all work in polynomial time. Thus the LOU strategy has the selection property.

Proof. The three algorithms are defined by primitive recursion. More specifically:
- Any call $\mathcal{A}_g(t)$ triggers at most $|t|$ calls to $\mathcal{A}_g$;
- Any call $\mathcal{B}_g(t)$ triggers at most $|t|$ calls to $\mathcal{B}_g$;
- Any call $\mathcal{C}_g(t, S)$ triggers at most $|t| + |S|$ calls to $\mathcal{B}$ and at most $|S|$ calls to $\mathcal{C}$;

Now, the amount of work involved in any single call (not counting the, possibly recursive, calls) is itself polynomial, simply because the tuples produced in output are made of objects whose size is itself bounded by the length of the involved terms and contexts. □

What Proposition 14.2 implicitly tells us is that the usefulness of a given redex in an LSC term t can be checked in polynomial time in the size of t. The Selection Property (Definition 7.3) then holds for LOU derivations: the next redex to be fired is the LO useful one (of course, finding the LO useful redex among useful redexes can trivially be done in polynomial time).

15. Summing Up

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

Theorem 15.1 (Polynomial Implementation of λ)**.** There is an algorithm which takes as input a λ-term t and a natural number n and which, in time polynomial in $m = \min\{n, \#_{\to_{\mathrm{LO}_\beta}}(t)\}$ and $|t|$, outputs an LSC term u such that $t \to^m u{\downarrow}$.

Together with the linear implementation of Turing machines in the λ-calculus given in [ADL12], we obtain our main result.

Theorem 15.2 (Invariance)**.** The λ-calculus is a reasonable model in the sense of the weak invariance thesis.

As we have already mentioned, the algorithm witnessing the invariance of the λ-calculus does *not* produce a λ-term, but a useful normal form, *i.e.* a compact representation (with ES) of a λ-term. Theorem 15.1, 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 [Sta79]:

Corollary 15.3. There is an algorithm which takes as input two λ-terms t and u and checks whether t and u have the same normal form in time polynomial in $\#_{\to_{\mathrm{LO}_\beta}}(t)$, $\#_{\to_{\mathrm{LO}_\beta}}(u)$, $|t|$, and $|u|$.

If one instantiates Corollary 15.3 to the case in which u is a (useful) normal form, one obtains that checking whether the normal form of any term t is equal to (the unfolding of) u can be done in time polynomial in $\#_{\to_{\mathrm{LO}_\beta}}(t)$, $|t|$, 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* normalizable, the algorithms above (correctly) diverge.

16. Discussion

Applications. One might wonder what is the practical relevance of our invariance result, since functional programming languages rely on weak evaluation, for which invariance was already known. The main application of strong evaluation is in the design of proof assistants and higher-order logic programming, typically for type-checking in frameworks with dependent types as the Edinburgh Logical Framework or the Calculus of Constructions, as well as for unification modulo $\beta\eta$ in simply typed frameworks like λ-Prolog. Of course, in these cases the language at work is not as minimalistic as the λ-calculus, it is often typed, and other operations (*e.g.* unification) impact on the complexity of evaluation. Nonetheless, the strong λ-calculus is always the core language, and so having a reasonable cost model for it is a necessary step for complexity analyses of these frameworks. Let us point out, moreover, that in the study of functional programming languages there is an emerging point of view, according to which the theoretical study of the language should be done with respect to strong evaluation, even if only weak evaluation will be implemented, see [SR15]. We also believe that our work may be used to substantiate the practical relevance of some theoretical works. There exists a line of research attempting to measure the number of steps to evaluate a term by looking to its denotational interpretations (*e.g.* relational semantics/intersection types in [dC09, dCPdF11, BL13, LMMP13] and game semantics in [Ghi05, DLL08, Cla13]) with the aim of providing abstract formulations of complexity properties. The problem of this literature is that either the measured strong strategies do not provide reliable complexity measures, or they only address head/weak reduction. In particular, the number of LO

steps to normal form—*i.e.* our cost model—has never been measured with denotational tools. This is particularly surprising, because head reduction is the strategy arising from denotational considerations (this is the leading theme of Barendregt's book [Bar84]) and the LO strategy is nothing but iterated head reduction. We expect that our result will be the starting point for revisiting the quantitative analyses of β-reduction based on denotational semantics.

Mechanizability vs Efficiency. Let us stress that the study of invariance is about *mechanizability* rather than *efficiency*. One is not looking for the smartest or shortest evaluation strategy, but rather for one that can be reasonably implemented. The case of Lévy's optimal evaluation, for instance, hides the complexity of its implementation in the cleverness of its definition. A Lévy-optimal derivation, indeed, can be even shorter than the shortest sequential strategy, but—as shown by Asperti and Mairson [AM98]—its definition hides hyper-exponential computations, so that optimal derivations do not provide an invariant cost model. The leftmost-outermost strategy, is a sort of *maximally unshared* normalizing 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 mechanizability. 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 optimally, they collapse the complexity of too many steps into a single one, making the number of steps an unreliable measure.

Inefficiencies. This work is foundational in spirit and only deals with polynomial bounds, and in particular it does not address an efficient implementation of useful sharing. There are three main sources of inefficiency:

(1) *Call-by-Name Evaluation Strategy*: for a more efficient evaluation one should at least adopt a call-by-need policy, while many would probably prefer to switch to call-by-value altogether. Both evaluations introduce some sharing of *computations* with respect to call-by-name, as they evaluate the argument before it is substituted (call-by-need) or the β-redex is fired (call-by-value). Our choice of call-by-name, however, comes from the desire to show that even the good old λ-calculus with normal order evaluation is invariant, thus providing a simple cost model for the working theoretician.
(2) *High-Level Quadratic Overhead*: in the micro-step evaluation presented here the number of substitution steps is at most quadratic in the number of β-steps, as proved in the high-level implementation theorem. Such a bound is strict, as there exist degenerate terms that produce these quadratic substitution overhead—for instance, the micro-step evaluation of the paradigmatic diverging term Ω, but the degeneracy can also be adapted to terminating terms.
(3) *Low-Level Separate Useful Tests*: for the low-level implementation theorem we provided a separate *global* test for the usefulness of a substitution step. It is natural to wonder if an abstract machine can implement it *locally*. The idea we suggested in [ADL14a] is that some additional labels on subterms may carry information about the unfolding in their context, allowing to decide usefulness in linear time, and removing the need of running a global check.

These inefficiencies have been addressed by Accattoli and Sacerdoti Coen in two studies [ASC14, ASC15], complementary to ours. In [ASC14], they show that (in the much simpler weak case) call-by-value and call-by-need both satisfy an high-level implementation theorem

and that the quadratic overhead is induced by potential chains of renaming substitutions, sometimes called *space leaks*. Moreover, in call-by-value and call-by-need the quadratic overhead can be reduced to *linear* by simply removing variables from values. The same speed-up can be obtained for call-by-name as well, if one slightly modifies the micro-step rewriting rules (see the long version of [ASC14]—that at the time of writing is submitted and can only be found on Accattoli's web page—that builds on a result of [SGM02]). In [ASC15], instead, the authors address the possibility of local useful tests, but motivated by [ASC14], they rather do it for a weak call-by-value calculus generalized to evaluate open terms, that is the evaluation model used by the abstract machine at work in the Coq proof assistant [GL02]. Despite being a weak setting, open terms force to address useful sharing along the lines of what we did here, but with some simplifications due to the weak setting. The novelty of [ASC15] is an abstract machine implementing useful sharing, and studied via a *distillation*, *i.e.* a new methodology for the representation of abstract machines in the LSC [ABM14]. Following the mentioned suggestion, the machine uses simple labels to check usefulness *locally* and—surprisingly—the check takes *constant time*. Globally, the machine is proved to have an overhead that is *linear* both in the number of β-steps and the size of the initial term. Interestingly, that work builds on the schema for usefulness that we provided here, showing that the our approach, and in particular useful sharing, are general enough to encompass more efficient scenarios. But there is more. At first sight call-by-value seemed to be crucial in order to obtain a linear overhead, but the tools of [ASC15]—a posteriori—seem to be adaptable to call-by-name, with a slight slowdown: useful tests are checked in linear rather than constant time (linear in the size of the initial term). For call-by-need with open terms, the same tools seem to apply, even if we do not yet know if useful tests are linear or constant. Generally speaking, our result can be improved along two superposing axes. One is to refine the invariant strategy so as to include as much sharing of computations as possible, therefore replacing call-by-name with call-by-value or call-by-need with open terms, or under abstractions. The other axe is to refine the overhead in implementing micro-step useful evaluation (itself splitting into two high-level and low-level axes), which seems to be doable in (bi)linear time more or less independently of the strategy.

On Non-Deterministic β-Reduction. This paper only deals with the cost of reduction induced by the natural, but inefficient leftmost-outermost strategy. The invariance of full β-reduction, *i.e.* of the usual non-deterministic relation allowed to reduce β-redexes in any order, would be very hard to obtain, since it would be equivalent to the invariance of the cost model induced by the optimal *one-step* deterministic reduction strategy, which is well known to be even non-recursive [Bar84]. Note that, a priori, *non-recursive* does not imply *non-invariant*, as there may be an algorithm for evaluation polynomial in the steps of the optimal strategy and that does not simulate the strategy itself—the existence of such an algorithm, however, is unlikely. The optimal *parallel* reduction strategy is instead recursive but, as mentioned in the introduction, the number of its steps to normal form is well known *not* to be an invariant cost model [AM98].

17. Conclusions

This work can be seen as 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 [DLM08,

DLM12, ADL12]. None of the results in the literature, however, concerns reduction to normal form as we do here.

We provided the first full answer to a long-standing open problem: the λ-calculus is indeed a reasonable machine, if the length of the leftmost-outermost derivation to normal form is used as cost model.

To solve the problem we developed a whole new toolbox: an abstract deconstruction of the problem, a theory of useful derivations, a general view of functions efficiently computable in compact form, and a surprising connection between standard and efficiently mechanizable derivations. Theorem after theorem, an abstract notion of machine emerges, hidden deep inside the λ-calculus itself. While such a machine is subtle, the cost model turns out to be the simplest and most natural one, as it is unitary, machine-independent, and justified by the standardization theorem, a classic result apparently unrelated to the complexity of evaluation.

This work also opens the way to new studies. Providing an invariant cost model, *i.e.* a metric for efficiency, it gives a new tool to compare different implementations, and to guide the development of new, more efficient ones. As discussed in the previous section, Accattoli and Sacerdoti Coen presented a call-by-value abstract machine for useful sharing having only a linear overhead [ASC15], that actually on open λ-terms is asymptotically faster than the abstract machine at work in the Coq proof assistant, studied in [GL02]. Such a result shows that useful sharing is not a mere theoretical tool, and justifies a finer analysis of the invariance of λ-calculus.

Among the consequences of our results, one can of course mention that proving systems to characterize 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, for instance, in the context of *light logics* [GR07, CDLRDR08, BT09]. The kind of bounds we obtain here are however more *general* than those obtained in implicit computational complexity, because we deal with a universal model of computation.

While there is room for finer analyses, we consider the understanding of time invariance essentially achieved. However, the study of cost models for λ-terms is far from being over. Indeed, the study of space complexity for functional programs has only made its very first steps [Sch07, GMR08, DLS10, Maz15], and not much is known about invariant *space* cost models.

References

[ABKL14] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 659–670, 2014.

[ABM14] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376, 2014.

[Acc12] Beniamino Accattoli. An abstract factorization theorem for explicit substitutions. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pages 6–21, 2012.

[ADL12] Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pages 22–37, 2012.

[ADL14a] Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, page 8, 2014.

[ADL14b] Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed (long version). Technical report associated to [ADL14a], Available at `http://arxiv.org/abs/1405.3311`, 2014.

[AG09] Beniamino Accattoli and Stefano Guerrini. Jumping boxes. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, pages 55–70, 2009.

[AM98] Andrea Asperti and Harry G. Mairson. Parallel beta reduction is not elementary recursive. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 303–315, 1998.

[AM10] Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and polytime computability. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, pages 33–48, 2010.

[ASC14] Beniamino Accattoli and Claudio Sacerdoti Coen. On the value of variables. In *Logic, Language, Information, and Computation - 21st International Workshop, WoLLIC 2014, Valparaíso, Chile, September 1-4, 2014. Proceedings*, pages 36–50, 2014.

[ASC15] Beniamino Accattoli and Claudio Sacerdoti Coen. On the relative usefulness of fireballs. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 141–155, 2015.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103. North-Holland, 1984.

[BG95] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995.

[BL13] Alexis Bernadet and Stéphane Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4), 2013.

[BT09] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Inf. Comput.*, 207(1):41–62, 2009.

[dB87] Nicolaas 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.

[dC09] Daniel de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *CoRR*, abs/0905.4251, 2009.

[dCPdF11] Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. A semantic measure of the execution time in linear logic. *Theor. Comput. Sci.*, 412(20):1884–1902, 2011.

[CDLRDR08] Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science*, 4(4), 2008.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*. Studies in logic and the foundations of mathematics. North-Holland Publishing Company, 1958.

[Cla13] Pierre Clairambault. Bounding skeletons, locally scoped terms and exact bounds for linear head reduction. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, pages 109–124, 2013.

[DLL08] Ugo Dal Lago and Olivier Laurent. Quantitative game semantics for linear logic. In *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, pages 230–245, 2008.

[DLM08] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[DLM12] Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science*, 8(3), 2012.

[DLS10] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as*

Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, pages 205–225, 2010.

[DR04] Vincent Danos and Laurent Regnier. Head linear reduction. Technical report, 2004.

[FS91] Gudmund Skovbjerg Frandsen and Carl Sturtivant. What is an efficient implementation of the λ-calculus? In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, pages 289–312, 1991.

[Ghi05] Dan R. Ghica. Slot games: a quantitative model of computation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 85–97, 2005.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 235–246, 2002.

[GLM92] Georges Gonthier, Jean-Jacques Lévy, and Paul-André Melliès. An abstract standardisation theorem. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*, pages 72–81, 1992.

[GMR08] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. A logical account of pspace. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 121–131, 2008.

[GR07] Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for *lambda* -calculus. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, pages 253–267, 2007.

[GR14] Clemens Grabmayer and Jan Rochel. Maximal sharing in the lambda calculus with letrec. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 67–80, 2014.

[Jon96] Neil D. Jones. What not to do when writing an interpreter for specialisation. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, pages 216–237, 1996.

[Lév78] Jean-Jacques Lévy. Réductions correctes et optimales dans le lambda-calcul. Thése d'Etat, Univ. Paris VII, France, 1978.

[LM96] Julia L. Lawall and Harry G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96), Philadelphia, Pennsylvania, May 24-26, 1996.*, pages 92–101, 1996.

[LMMP13] Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. Weighted relational models of typed lambda-calculi. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 301–310, 2013.

[Maz15] Damiano Mazza. Simple parsimonious types and logarithmic space. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, pages 24–40, 2015.

[Mel96] Paul-André Melliès. *Description Abstraite de système de réécriture*. PhD thesis, Paris 7 University, 1996.

[Mil07] Robin Milner. Local bigraphs and confluence: Two conjectures. *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007.

[Ned92] Robert. P. Nederpelt. The fine-structure of lambda calculus. Technical Report CSN 92/07, Eindhoven Univ. of Technology, 1992.

[PJ87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 411–420, 2007.

[SGM02] David Sands, Jörgen Gustavsson, and Andrew Moran. Lambda calculi and linear speedups. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, pages 60–84, 2002.

[SR15] Gabriel Scherer and Didier Rémy. Full reduction in the face of absurdity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 685–709, 2015.

[Sta79] Richard Statman. The typed lambda-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9:73–81, 1979.

[SvEB84] Cees F. Slot and Peter van Emde Boas. On tape versus core; an application of space efficient perfect hash functions to the invariance of space. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 391–400, 1984.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. PhD Thesis, Oxford, 1971.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany