On the Invariance of the Unitary Cost Model for Head Reduction

Beniamino Accattoli, Ugo Dal Lago

On the Invariance of the Unitary Cost Model for Head Reduction*

Beniamino Accattoli[1] and Ugo Dal Lago[2]

1 **INRIA & LIX (École Polytechnique)**
`beniamino.accattoli@inria.fr`
2 **Università di Bologna & INRIA**
`dallago@cs.unibo.it`

Abstract

The λ-calculus is a widely accepted computational model of higher-order functional programs, yet there is not any direct and universally accepted cost model for it. As a consequence, the computational difficulty of reducing λ-terms to their normal form is typically studied by reasoning on concrete implementation algorithms. In this paper, we show that when head reduction is the underlying dynamics, the unitary cost model is indeed invariant. This improves on known results, which only deal with weak (call-by-value or call-by-name) reduction. Invariance is proved by way of a *linear* calculus of explicit substitutions, which allows to nicely decompose any head reduction step in the λ-calculus into more elementary substitution steps, thus making the combinatorics of head-reduction easier to reason about. The technique is also a promising tool to attack what we see as the main open problem, namely understanding for which *normalizing* strategies the unitary cost model is invariant, if any.

1998 ACM Subject Classification F.4.1 Mathematical Logic – Lambda calculus and related systems, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases lambda calculus, cost models, explicit substitutions, implicit computational complexity

Digital Object Identifier 10.4230/LIPIcs.RTA.2012.22

Category Regular Research Paper

1 Introduction

Giving an estimate of the amount of time T needed to execute a program is a natural refinement of the termination problem, which only requires to decide whether T is either finite or infinite. The shift from termination to complexity analysis brings more informative outcomes at the price of an increased difficulty. In particular, complexity analysis depends much on the chosen computational model. Is it possible to express such estimates in a way which is independent from the specific machine the program is run on? An answer to this question can be given following computational complexity, which classifies algorithms (and functions) based on the amount of time (or space) they consume when executed by *any* abstract device endowed with a *reasonable* cost model, depending on the size of input. When can a cost model be considered reasonable? The answer lies in the so-called invariance thesis [23]: any time cost model is reasonable if it is polynomially related to the (standard) one of Turing machines.

* This work was partially supported by the ARC INRIA project "ETERNAL".

23rd International Conference on Rewriting Techniques and Applications (RTA'12).
Editor: A. Tiwari; pp. 22–38

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

If programs are expressed as rewrite systems (e.g. as first-order TRSs), an abstract but effective way to execute programs, rewriting itself, is always available. As a consequence, a natural time cost model turns out to be *derivational complexity*, namely the (maximum) number of rewrite steps which can possibly be performed from the given term. A rewriting step, however, may not be an atomic operation, so derivational complexity is not by definition invariant. For first-order TRSs, however, derivational complexity has been recently shown to be an invariant cost model, by way of term graph rewriting [12, 8, 9].

The case of λ-calculus is definitely more delicate: if β-reduction is weak, i.e., if it cannot take place in the scope of λ-abstractions, one can see λ-calculus as a TRS and get invariance by way of the already cited results [11], or by other means [21]. But if one needs to reduce "under lambdas" because the final term needs to be in normal form (e.g., when performing type checking in dependent type theories), no invariance results are known at the time of writing.

In this paper we give a partial solution to this problem, by showing that the unitary cost model is indeed invariant for the λ-calculus endowed with *head reduction*, in which reduction *can* take place in the scope of λ-abstractions, but *can only* be performed in head position. Our proof technique consists in implementing head reduction in a calculus of explicit substitutions.

Explicit substitutions were introduced to close the gap between the theory of λ-calculus and implementations [1]. Their rewriting theory has also been studied in depth, after Melliès showed the possibility of pathological behaviors [15]. Starting from graphical syntaxes, a new *at a distance* approach to explicit substitutions has recently been proposed [6]. The new formalisms are simpler than those of the earlier generation, and another thread of applications — to which this paper belongs — also started: new results on λ-calculus have been proved by means of explicit substitutions [6, 7].

In this paper we use the *linear-substitution calculus* $\Lambda_{[.]}$, a slight variation over a calculus of explicit substitutions introduced by Robin Milner [17]. The variation is inspired by the structural λ-calculus [6]. We study in detail the relation between λ-calculus head reduction and *linear head reduction* [14], the head reduction of $\Lambda_{[.]}$, and prove that the latter is at most quadratically longer than the former. This is proved without any termination assumption, by a detailed rewriting analysis.

To get the Invariance Theorem, however, other ingredients are required:

1. *The Subterm Property.* Linear head reduction has a property not enjoyed by head β-reduction: linear substitutions along a reduction $t \multimap^* u$ duplicate subterms of t only. It easily follows that $\multimap$-steps can be simulated by Turing machines in time polynomial in the size of t and the length of $\multimap^*$. This is explained in Section 3.
2. *Compact representations.* Explicit substitutions, decomposing β-reduction into more atomic steps, allow to take advantage of sharing and thus provide compact representations of terms, avoiding the exponential blowups of term size happening in plain λ-calculus. Is it reasonable to use these compact representations of λ-terms? We answer affirmatively, by exhibiting a dynamic programming algorithm for checking equality of terms with explicit substitutions modulo unfolding, and proving it to work in polynomial time in the size of the involved compact representations. This is the topic of Section 5.
3. *Head simulation of Turing machines.* We also provide the simulation of Turing machines by λ-terms. We give a new encoding of Turing machines, since the known ones do not work with *head* β-reduction, and prove it induces a polynomial overhead. Some details of the encoding are given in Section 6.

We emphasize the result for head β-reduction, but our technical detour also proves invariance

for linear head reduction. To our knowledge, we are the first ones to use the fine granularity of explicit substitutions for complexity analysis. Many calculi with bounded complexity (e.g. [22]) use `let`-constructs, an avatar of explicit substitutions, but they do not take advantage of the refined dynamics, as they always use big-steps substitution rules.

To conclude, we strongly believe that the main contribution of this paper lies in the technique rather than in the invariance result. Indeed, the main open problem in this area, namely the invariance of the unitary cost model for any *normalizing* strategy remains open but, as we argue in Section 4, seems now within reach.

An extended version of this paper with more detailed proofs is available [4].

2 λ-Calculus and Cost Models: an Informal Account

Consider the pure, untyped, λ-calculus. Terms can be variables, abstractions or applications and computation is modeled by β-reduction. Once a reduction strategy is fixed, one could be tempted to make *time* and *reduction steps* to correspond: firing a β-redex requires one time instant (or, equivalently, a finite number of time instants) and thus the number of reduction steps to normal form could be seen as a measure of its time complexity. This would be very convenient, since reasoning on the complexity of normalization could be done this way directly on λ-terms. However, doing so one could in principle risk to be too optimistic about the complexity of obtaining the normal form of a term t, given t as an input. This section will articulate on this issue by giving some examples and pointers to the relevant literature.

Consider the sequence of λ-terms defined as follows, by induction on a natural number n (where u is the lambda term yxx): $t_0 = u$ and for every $n \in \mathbb{N}$, $t_{n+1} = (\lambda x.t_n)u$. t_n has size linear in n, and t_n rewrites to its normal form r_n in exactly n steps, following a leftmost-outermost strategy:

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

For every n, however, r_{n+1} contains two copies of r_n, hence the size of r_n is *exponential* in n. As a consequence, if we stick to the leftmost-outermost strategy and if we insist on normal forms to be represented explicitly, without taking advantage of sharing, the unitary cost model *is not* invariant: in a linear number of β-steps we reach an object which cannot even be written down in polynomial time.

One may wonder whether this problem is due to the specific, inefficient, adopted strategy. However, it is quite easy to rebuild a sequence of terms exhibiting the same behavior along an innermost strategy: if $s = \lambda y.yxx$, define v_0 to be just $\lambda x.s$, for every $n \in \mathbb{N}$, v_{n+1} to be $\lambda x.v_n s$, and consider $v_n(\lambda x.x)$. Actually, there *are* invariant cost-models for the λ-calculus even if one wants to obtain the normal form in an explicit, linear format, like the difference cost model [10]. But they pay a price for that: they do not attribute a constant weight to each reduction step. Then, another natural question arises: is it that the gap between the unitary cost model and the real complexity of reducing terms is only due to a *representation* problem? In other words, could we take advantage of a shared representation of terms, even if only to encode λ-terms (and normal forms in particular) in a compact way?

The literature offers some positive answers to the question above. In particular, the unitary cost model can be proved to be invariant for both call-by-name and call-by-value

λ-calculi, as defined by Plotkin [19]. In one way or another, the mentioned results are based on sharing subterms, either by translating the λ-calculus to a TRS [11] or by going through abstract machines [21]. Plotkin's calculi, however, are endowed with *weak* notions of reduction, which prevent computation to happen in the scope of a λ-abstraction. And the proposed approaches crucially rely on that.

The question now becomes the following: is it possible to prove the invariance of the unitary cost model for some *strong* notion of reduction? This paper gives a first, positive answer to this question by proving the number of β-reduction steps to be an invariant cost model for *head* reduction, in which one *is* allowed to reduce in the scope of λ-abstractions, but evaluation stops on *head* normal forms.

We are convinced that the exponential blowup in the examples above is, indeed, only due to the λ-calculus being a very inefficient *representation* formalism. Following this thesis we use terms with explicit substitutions as compact representations: our approach, in contrast to other ones, consists in using sharing (here under the form of explicit substitutions) only to obtain compactness, and not to design some sort of optimal strategy reducing shared redexes. Actually, we follow the opposite direction: the leftmost-outermost strategy — being standard — can be considered as the maximally *non-sharing* strategy. How much are we losing limiting ourselves to head reduction? Not so much: in Section 6 we show an encoding of Turing machines for which the normal form is reached by head-reduction only. Moreover, from a denotational semantics point of view head-normal forms — and not full normal forms — are the right notion of result for β-reduction.

The next two sections introduce explicit substitutions and prove that the length of their head strategy is polynomially related to the length of head β-reduction. In other words, the switch to compact representations does not affect the cost model in any essential way.

3 Linear Explicit Substitutions

First of all, we introduce the λ-calculus. Its terms are given by the grammar:

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

and its reduction rule $\to_\beta$ is defined as the context closure of $(\lambda x.t)\ u \mapsto_\beta t\{x/u\}$. $\mathcal{T}_\lambda$ is the set of all terms of the λ-calculus. We will mainly work with head reduction, instead of full β-reduction. We define head reduction as follows. Let an *head context* $\hat{H}$ be defined by:

$$\hat{H} ::= [\cdot] \;\mid\; \hat{H}\ t \;\mid\; \lambda x.\hat{H}.$$

Then define *head reduction* $\to_{\mathtt{h}}$ as the closure by head contexts of $\mapsto_\beta$. Our definition of head reduction is slightly more liberal than the usual one. Indeed, it is non-deterministic, for instance:

$$(\lambda x.I)\ t\ {}_{\mathtt{h}}\!\leftarrow (\lambda x.(I\ I))\ t \to_{\mathtt{h}} I\ I.$$

Usually only one of the two redexes would be considered an head redex. However, this non-determinism is harmless, since one easily proves that $\to_{\mathtt{h}}$ has the diamond property. Reducing $\to_{\mathtt{h}}$ in an outermost way we recover the usual notion of head reduction, so our approach gains in generality without loosing any property of head reduction. Our notion is motivated by the corresponding notion of head reduction for explicit substitutions, which is easier to manage in this more general approach.

The calculus of explicit substitutions we are going to use is a minor variation over a simple calculus introduced by Milner [17]. The grammar is standard:

$$t, u, r ::= x \;\mid\; t\ t \;\mid\; \lambda x.t \;\mid\; t[x/t].$$

$$\begin{array}{lll} (\lambda x.t)\mathtt{L}\ u & \mapsto_{\mathtt{dB}} & t[x/u]\mathtt{L} \\ C[x][x/u] & \mapsto_{\mathtt{ls}} & C[u][x/u] \\ t[x/u] & \mapsto_{\mathtt{gc}} & t \qquad \text{if } x \notin \mathtt{fv}(t) \end{array}$$

Figure 1 $\Lambda_{[\cdot]}$ Rewriting Rules.

The term $t[x/u]$ is an *explicit substitution*. $\mathcal{T}$ is the set of terms with explicit substitutions. Observe that $\mathcal{T}_\lambda \subset \mathcal{T}$. Both constructors $\lambda x.t$ and $t[x/u]$ bind x in t. We note $\mathtt{L}$ a possibly empty list of explicit substitutions $[x_1/u_1]\dots[x_k/u_k]$. Contexts are defined by:

$$C, D, E, F ::= [\cdot] \mid C\ t \mid t\ C \mid \lambda x.C \mid C[x/t] \mid t[x/C].$$

We note $C[t]$ the usual operation of substituting $[\cdot]$ in C possibly capturing free variables of t. We will often use expressions like $C[x][x/u]$ where it is implicitly assumed that C does not capture x. The *linear-substitution calculus* $\Lambda_{[\cdot]}$ is given by the rewriting rules $\to_{\mathtt{dB}}$, $\to_{\mathtt{ls}}$ and $\to_{\mathtt{gc}}$, defined as the context closures of the rules $\mapsto_{\mathtt{dB}}$, $\mapsto_{\mathtt{ls}}$ and $\mapsto_{\mathtt{gc}}$ in Figure 1. We also use the notation $\to_{\Lambda_{[\cdot]}} = \to_{\mathtt{dB}} \cup \to_{\mathtt{ls}} \cup \to_{\mathtt{gc}}$ and $\to_{\mathtt{s}} = \to_{\mathtt{ls}} \cup \to_{\mathtt{gc}}$. Rule $\to_{\mathtt{dB}}$ acts at a *distance*: the function $\lambda x.t$ and the argument u can interact even if there is $\mathtt{L}$ between them. This is motivated by the close relation between $\Lambda_{[\cdot]}$ and graphical formalisms as proof-nets and $\lambda\mathtt{j}$-dags [3, 5], and is also the difference with Milner's presentation of $\Lambda_{[\cdot]}$ [17].

The linear-substitution calculus enjoys all properties required to explicit substitutions calculi, obtained by easy adaptations of the proofs for $\lambda\mathtt{j}$ in [6]. Moreover, it is confluent and preserves β-strong normalization. In particular, $\to_{\mathtt{s}}$ is a strongly normalizing and confluent relation.

Given a term t with explicit substitutions, its normal form with respect to $\to_{\mathtt{s}}$ is a λ-term, noted $t{\downarrow}$, called the *unfolding* of t and verifying the following equalities:

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

Another useful property is the so-called *full-composition*, which states that any explicit substitution can be reduced to its implicit form independently from the other substitutions in the term, formally $t[x/u] \to_{\mathtt{s}}^* t\{x/u\}$. Last, $\Lambda_{[\cdot]}$ simulates λ-calculus ($t \to_\beta u$ implies $t \to_{\Lambda_{[\cdot]}}^* u$) and reductions in $\Lambda_{[\cdot]}$ can be projected on λ-calculus via unfolding ($t \to_{\Lambda_{[\cdot]}} u$ implies $t{\downarrow} \to_\beta^* u{\downarrow}$).

The calculus $\Lambda_{[\cdot]}$ has a strong relation with proof-nets and linear logic: it can be mapped to Danos' and Regnier's pure proof-nets [20] or to $\lambda\mathtt{j}$-dags [5]. The rule $\to_{\mathtt{dB}}$ corresponds to proof-nets multiplicative cut-elimination, $\to_{\mathtt{ls}}$ to the cut-elimination rule between ! (every substitution is in a !-box) and contraction, $\to_{\mathtt{gc}}$ to the cut-elimination rule between ! and weakening. The case of a cut between ! and dereliction is handled by $\to_{\mathtt{ls}}$, as if cut dereliction were always contracted with a weakening.

3.1 Linear Head Reduction, the Subterm Property and Shallow Terms

In this paper, we mainly deal with a specific notion of reduction for $\Lambda_{[\cdot]}$, called *linear head reduction* [14, 13], and related to the representation of λ-calculus into linear logic proof-nets. In order to define it we need the notion of head context for explicit substitutions.

▶ **Definition 3.1** (head context). *Head contexts* are defined by the following grammar:

$$H ::= [\cdot] \mid H\ t \mid \lambda x.H \mid H[x/t].$$

The fundamental property of an head context H is that the hole cannot be duplicated nor erased. In terms of linear logic proof-nets, the head hole is not contained in any box (since boxes are associated with the arguments of applications and with explicit substitutions). We now need to consider a modified version of $\mapsto_{\mathtt{ls}}$:

$$H[x][x/u] \hat{\multimap}_{\mathtt{ls}} H[u][x/u].$$

Now, let $\multimap_{\mathtt{dB}}$ (resp. $\multimap_{\mathtt{ls}}$) be the closure by head contexts of $\mapsto_{\mathtt{dB}}$ (resp. $\hat{\multimap}_{\mathtt{ls}}$). Last, define *head linear reduction* $\multimap$ as $\multimap_{\mathtt{dB}} \cup \multimap_{\mathtt{ls}}$. Please notice that $\multimap$ can reduce under λ, for instance $\lambda y.(H[x][x/u]) \multimap_{\mathtt{ls}} \lambda y.(H[u][x/u])$. Our definition of $\multimap$ gives a non-deterministic strategy, but its non-determinism is again harmless: a simple case analysis shows that $\multimap$ has the diamond property.

In [2] it is proved that linear head reduction has the same factorization property enjoyed by head reduction in λ-calculus: if $t \to^*_{\Lambda_{[\cdot]}} u$ then $t \multimap^* \Rightarrow^*$, where $\Rightarrow$ is the complement of $\to_{\Lambda_{[\cdot]}}$ with respect to $\multimap$ (*i.e.* $\Rightarrow := \to_{\Lambda_{[\cdot]}} \setminus \multimap$).

A term u is a *box-subterm* of a term t (resp. of a context C) if t (resp. C) has a subterm of the form $r\ u$ or of the form $r[x/u]$ for some r.

▶ Remark. By definition of head-contexts, $[\cdot]$ is not a box-subterm of $H[\cdot]$, and there is no box-subterm of $H[\cdot]$ which has $[\cdot]$ as a subterm.

The following fundamental property of head linear reduction is folklore among specialists, but to our knowledge it has never appeared in print before.

▶ **Proposition 3.2** (Subterm Property). *If $t \multimap^* u$ and r is a box-subterm of u, then r is a box-subterm of t.*

The aforementioned proposition is a key point in our study of cost models. Linear head substitution steps duplicate sub-terms, but the Subterm Property guarantees that only subterms of the initial term t are duplicated, and thus each step can be implemented in time polynomial in the size of t, which is the size of the input, the fundamental parameter for complexity analysis. This is in sharp contrast with what happens in the λ-calculus, where the cost of a β-reduction step is not even polynomially related to the size of the initial term.

Proof. Let $t \multimap^k u$. We proceed by induction on k. Suppose that $k > 0$. Then $t \multimap^* v \multimap u$ and by *i.h.* any box-subterm of v is a box subterm of t, so it is enough to show that any box-subterm of u is a box-subterm of v. By induction on $v \multimap u$. The case $v = (\lambda x.r)\mathtt{L}\ s \multimap r[x/s]\mathtt{L}$ is trivial. If $v = H[x][x/s] \multimap H[s][x/s]$ by the previous remark the plug of s in $H[\cdot]$ does not create any new box-subterm, nor modify a box-subterm of $H[\cdot]$. And obviously any box-subterm of s is a box-subterm of v. The inductive cases follow from the *i.h.* and the previous remark. ◀

The subterm property does not only allow to bound the cost of implementing any reduction step, but also to bound the size of intermediate terms:

▶ **Corollary 3.3.** *There is a polynomial $p : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ such that if $t \multimap^k u$ then $|u| \leq p(k, |t|)$.*

Consider a reduction $t \multimap^* u$ where t is a λ-term. Another consequence of the Subterm Property is that for every explicit substitution occurring in u, the substituted term is a λ-term. This is another strong property to be used in the analysis of the next section.

▶ **Definition 3.4** (Shallow Terms). A $\Lambda_{[\cdot]}$-term t is *shallow* if whenever $t = C[u[x/r]]$ then r is a λ-term.

▶ **Corollary 3.5.** *Let t be a λ-term and $t \multimap^* u$. Then u is shallow.*

4 On the Relation Between Λ and $\Lambda_{[\cdot]}$

In this section, linear explicit substitutions will be shown to be an efficient way to implement head reduction. We will proceed by proving three auxiliary results separately:

1. We show that any $\multimap$-reduction ρ projects via unfolding to a $\to_{\mathtt{h}}$-reduction $\rho{\downarrow}$ having as length exactly the number of $\multimap_{\mathtt{dB}}$ steps in ρ; this is the topic of Section 4.1;
2. We show the converse relation, *i.e.* that any $\to_{\mathtt{h}}$-reduction ρ can be simulated by a $\multimap$-reduction having as many $\multimap_{\mathtt{dB}}$-steps as the the steps in ρ, followed by unfolding; this is in Section 4.2;
3. We show that in any $\multimap$-reduction ρ the number of $\multimap_{\mathtt{ls}}$-steps is $\mathcal{O}(|\rho|^2_{\mathtt{dB}})$ where $|\rho|_{\mathtt{dB}}$ is the number of $\multimap_{\mathtt{dB}}$ steps in ρ. By the previous two points, there is a quadratic — and thus polynomial — relation between $\to_{\mathtt{h}}$-reductions and $\multimap$-reduction from a given term; all this is explained in Section 4.3.

4.1 Projection of $\multimap$ on $\to_{\mathtt{h}}$

The first thing one needs to prove about head-linear reduction is whether it is a *sound* way to implement head reduction. This is proved by relatively standard techniques, and requires the following auxiliary lemma, whose proof is by induction on $t \to_{\mathtt{h}} u$:

▶ **Lemma 4.1.** *Let $t \in \mathcal{T}_\lambda$. If $t \to_{\mathtt{h}} u$ then $t\{x/r\} \to_{\mathtt{h}} u\{x/r\}$.*

▶ **Lemma 4.2** (Projection of $\multimap$ on $\to_{\mathtt{h}}$)**.** *Let $t \in \mathcal{T}$. If $\rho : t \multimap^k u$ then $t{\downarrow} \to_{\mathtt{h}}^n u{\downarrow}$ and $n = |\rho|_{\mathtt{dB}} \leq k$.*

Proof. By induction on k. If $k = 0$ it is trivial, so let $k > 0$ and $t \multimap^{n-1} r \multimap u$. Let τ be the reduction $t \multimap^{n-1} r$. By *i.h.* we get $t{\downarrow} \to_{\mathtt{h}}^m r{\downarrow}$ and $m = |\tau|_{\mathtt{dB}} \leq k-1$. Now consider $r \multimap u$. There are two cases. If $r \multimap_{\mathtt{ls}} u$ then $r{\downarrow} = u{\downarrow}$, by definition of $(\cdot){\downarrow}$, as the normal form of $\to_{\mathtt{s}}$ (which contains $\multimap_{\mathtt{ls}}$). If $r \multimap_{\mathtt{dB}} u$ then $r = H[(\lambda x.v)\mathtt{L}\ s] \multimap H[v[x/s]\mathtt{L}] = u$. We prove that $r{\downarrow} \to_{\mathtt{h}} u{\downarrow}$, from which it follows that $t{\downarrow} \to_{\mathtt{h}}^{m+1} u{\downarrow}$, where $m + 1 = |\tau|_{\mathtt{dB}} + 1 = |\rho|_{\mathtt{dB}} \leq k$. We go by induction on H. We use the following notation: if $t = u\mathtt{L}$ and $\mathtt{L} = [y_1/w_1] \dots [y_m/w_m]$ then we write $\sigma_{\mathtt{L}}$ for $\{y_1/w_1{\downarrow}\} \dots \{y_m/w_m{\downarrow}\}$, thus we can write $t{\downarrow} = u{\downarrow}\sigma_{\mathtt{L}}$. Cases:

- $H = [\cdot]$. Then:

$$\begin{aligned} r{\downarrow} &= ((\lambda x.v)\mathtt{L}){\downarrow}\ s{\downarrow} = (\lambda x.v{\downarrow})\sigma_{\mathtt{L}}\ s{\downarrow} = (\lambda x.v{\downarrow}\sigma_{\mathtt{L}})\ s{\downarrow} \\ &\to_{\mathtt{h}} v{\downarrow}\sigma_{\mathtt{L}}\{x/s{\downarrow}\} \overset{*}{=} v{\downarrow}\{x/s{\downarrow}\}\sigma_{\mathtt{L}} = (v[x/s]){\downarrow}\sigma_{\mathtt{L}} = (v[x/s]\mathtt{L}){\downarrow} = u{\downarrow}\,. \end{aligned}$$

  Step (*) holds because $x \notin \mathtt{fv}(w_i)$ for $i \in \{1, \dots, m\}$.
- $H = J[y/w]$. Then by *i.h.* and Lemma 4.1 we get $r{\downarrow} = (J[(\lambda x.v)\mathtt{L}\ s]){\downarrow}\{y/w{\downarrow}\} \to_{\mathtt{h}} (J[v[x/s]\mathtt{L}]){\downarrow}\{y/w{\downarrow}\} = u{\downarrow}$.
- The cases $H = J\ w$ and $H = \lambda y.J$ follow from the *i.h.*

◀

4.2 Projection of $\to_{\mathtt{h}}$ on $\multimap$

Here we map head β-steps to head linear steps followed by unfolding. In other words, we prove that head-linear reduction is not only a sound, but also a *complete* way to implement head reduction. This section is going to be technically more involved than the previous one. First of all, we show that a single head step can be simulated by a step of $\multimap_{\mathtt{dB}}$ followed by unfolding, which is straightforward:

▶ **Lemma 4.3** (Head Simulation)**.** *Let t be a λ-term. If $t \to_{\mathtt{h}} u$, then $t \multimap_{\mathtt{dB}} \to_{\mathtt{s}}^* u$.*

Proof. By induction on $t \to_{\mathtt{h}} u$. The base case: if $(\lambda x.u)\ r \to_{\mathtt{h}} u\{x/r\}$ then $(\lambda x.u)\ r \multimap_{\mathtt{dB}} u[x/r]$ and $u[x/r] \to_{\mathtt{s}}^* u\{x/r\}$ by full composition. The inductive cases follows by the *i.h.*. ◀

We are now going to show that a sequence of $\to_{\mathtt{ls}}$ reduction steps can be factored into some head-linear substitutions, followed by some linear substitutions under non-head contexts. This allows to refine Lemma 4.3. Define $\Rightarrow_{\mathtt{s}}$ as the relation $\to_{\mathtt{s}} \setminus \multimap_{\mathtt{ls}}$, *i.e.* $\Rightarrow_{\mathtt{s}}$ reduces non-head-linear substitution redexes. Moreover, define the *linear unfolding* $t{\downarrow_\circ}$ of t as the normal form of t with respect to $\multimap_{\mathtt{ls}}$ (which exists since $\multimap_{\mathtt{ls}} \subseteq \to_{\mathtt{ls}}$ and $\to_{\mathtt{ls}}$ terminates, and it is unique because $\multimap_{\mathtt{ls}}$ is deterministic).

Now, we can prove that any $\to_{\mathtt{h}}$ step is simulated in $\multimap_{\mathtt{dB}} \multimap_{\mathtt{ls}}^* \Rightarrow_{\mathtt{s}}^*$ (actually, in such a sequence there can be at most one $\multimap_{\mathtt{ls}}$ step):

▶ **Lemma 4.4** (Unfolding Factorization). *The following swaps hold:*

1. $\Rightarrow_{\mathtt{s}} \multimap_{\mathtt{ls}} \subseteq \multimap_{\mathtt{ls}}^+ \Rightarrow_{\mathtt{s}}^+$, *precisely:* $\Rightarrow_{\mathtt{s}} \multimap_{\mathtt{ls}} \subseteq \multimap_{\mathtt{ls}} \Rightarrow_{\mathtt{s}} \cup \multimap_{\mathtt{ls}} \multimap_{\mathtt{ls}} \Rightarrow_{\mathtt{s}} \cup \multimap_{\mathtt{ls}} \Rightarrow_{\mathtt{s}} \Rightarrow_{\mathtt{s}}$.
2. $\to_{\mathtt{ls}}^* \subseteq \multimap_{\mathtt{ls}}^* \Rightarrow_{\mathtt{s}}^*$, *and in particular* $t \multimap_{\mathtt{ls}}^* t{\downarrow_\circ} \Rightarrow_{\mathtt{s}}^* t{\downarrow}$

Proof. 1) Formally, the proof is by induction on $t \Rightarrow_{\mathtt{s}} u$. Informally, $\Rightarrow_{\mathtt{s}}$ cannot create, duplicate or alter the head nature of an $\multimap_{\mathtt{ls}}$-step, so that the second step in $t \Rightarrow_{\mathtt{s}} \multimap_{\mathtt{ls}} r$ can be traced back to a unique $\multimap_{\mathtt{ls}}$ redex. Now, there are two cases: either the two redexes simply permute or the preponement of $\multimap_{\mathtt{ls}}$ duplicate the redex reduced by $\Rightarrow_{\mathtt{s}}$. The second case splits in two subcases, of which we just give two examples:

$$\begin{array}{ccccc}
x[x/y][y/z] & & \Rightarrow_{\mathtt{s}} & & x[x/z][y/z] \\
\downarrow_{\circ\mathtt{ls}} & & \swarrow & & \downarrow_{\circ\mathtt{ls}} \\
y[x/y][y/z] & \multimap_{\mathtt{ls}} & z[x/y][y/z] & \Rightarrow_{\mathtt{s}} & z[x/z][y/z] \\
x[x/y\ y][y/z] & & \Rightarrow_{\mathtt{s}} & & x[x/y\ z][y/z] \\
\downarrow_{\circ\mathtt{ls}} & & \swarrow & & \downarrow_{\circ\mathtt{ls}} \\
(y\ y)[x/y\ y][y/z] & \Rightarrow_{\mathtt{s}} & (y\ z)[x/y\ y][y/z] & \Rightarrow_{\mathtt{s}} & (y\ z)[x/y\ z][y/z]
\end{array}$$

2) There is an abstract lemma (see [4]) which says that if $\to_2 \to_1 \subseteq \to_1^+ \to_2^*$ and $\to_1$ is strongly normalizing then $(\to_1 \cup \to_2)^* \subseteq \to_1^* \to_2^*$. Now, taking $\to_1$ to be $\multimap_{\mathtt{ls}}$, $\to_2$ to be $\Rightarrow_{\mathtt{s}}$ and since $\multimap_{\mathtt{ls}}$ is strongly normalising we get $\to_{\mathtt{ls}}^* = (\multimap_{\mathtt{ls}} \cup \Rightarrow_{\mathtt{s}})^* \subseteq \multimap_{\mathtt{ls}}^* \Rightarrow_{\mathtt{s}}^*$. ◀

We know that a $\to_{\mathtt{h}}$ step is simulated by a sequence of the form $\multimap_{\mathtt{dB}} \multimap_{\mathtt{ls}}^* \Rightarrow_{\mathtt{s}}^* \subseteq \multimap^* \Rightarrow_{\mathtt{s}}^*$. Consider two (or more) $\to_{\mathtt{h}}$-steps. They are simulated by a sequence of the form $\multimap^* \Rightarrow_{\mathtt{s}}^* \multimap^* \Rightarrow_{\mathtt{s}}^*$, while we would like to obtain $\multimap^* \Rightarrow_{\mathtt{s}}^*$. What we need to do is to prove that a sequence of the form $\Rightarrow_{\mathtt{s}}^* \multimap_{\mathtt{dB}}$ can always be reorganized as a sequence $\multimap_{\mathtt{dB}} \Rightarrow_{\mathtt{s}}^*$:

▶ **Lemma 4.5.** *The following inclusions hold:*

1. $\Rightarrow_{\mathtt{s}} \multimap_{\mathtt{dB}} \subseteq \multimap_{\mathtt{dB}} \Rightarrow_{\mathtt{s}}$.
2. $\Rightarrow_{\mathtt{s}}^* \multimap_{\mathtt{dB}} \subseteq \multimap_{\mathtt{dB}} \Rightarrow_{\mathtt{s}}^*$.

Proof. 1) By induction on $t \Rightarrow_{\mathtt{s}} u$. The idea is that $\Rightarrow_{\mathtt{s}}$ cannot create, duplicate nor alter the head nature of $\multimap_{\mathtt{dB}}$ redexes, therefore $\multimap_{\mathtt{dB}}$-step can be preponed. Conversely, $\multimap_{\mathtt{dB}}$-steps cannot erase, duplicate nor alter the non-head nature of $\Rightarrow_{\mathtt{s}}$ redexes, so the two steps commute. 2) Let $t \Rightarrow_{\mathtt{s}}^k \multimap_{\mathtt{dB}} u$. By induction on k using point 1 and a standard diagram chasing. ◀

The next lemma is the last brick for the projection.

▶ **Lemma 4.6.** *If* $t{\downarrow} \to_{\mathtt{h}} u$ *then there exists* r *s.t.* $t{\downarrow_\circ} \multimap_{\mathtt{dB}} r \to_{\mathtt{ls}}^* u$.

Proof. By Lemma 4.3, we get $t{\downarrow} \multimap \to_{\mathtt{ls}}^* u$. By Lemma 4.4, $t \to_{\mathtt{ls}}^* t{\downarrow}$factors as $t \multimap_{\mathtt{ls}}^* t{\downarrow_\circ} \Rightarrow_{\mathtt{s}}^* t{\downarrow}$, and so we get $t{\downarrow_\circ} \Rightarrow_{\mathtt{s}}^* \multimap \to_{\mathtt{ls}}^* u$. By Lemma 4.5.2, we get $t{\downarrow_\circ} \multimap \Rightarrow_{\mathtt{s}}^* \to_{\mathtt{ls}}^* u$, *i.e.* $t{\downarrow_\circ} \multimap \to_{\mathtt{ls}}^* u$. ◀

We can then conclude with the result which gives the name to this section:

▶ **Lemma 4.7** (Projection of $\to_{\mathtt{h}}$ on $\multimap$)**.** *Let t be a λ-term. If $t \to_{\mathtt{h}}^{k} u$ then there exists a reduction ρ s.t. $\rho : t \multimap^{*} r$, with $r \to_{\mathtt{ls}}^{*} u$ and $|\rho|_{\mathtt{dB}} = k$.*

Proof. By induction on k. The case $k = 0$ is trivial, so let $k > 0$ and $t \to_{\mathtt{h}}^{k-1} s \to_{\mathtt{h}} u$. By *i.h.* there exists a reduction τ s.t. $\tau : t \multimap^{*} v$, $v \to_{\mathtt{ls}}^{*} s$ and $|\tau|_{\mathtt{dB}} = k - 1$. Since s is a λ-term we have that $v{\downarrow} = s$ and $v{\downarrow} \to_{\mathtt{h}} u$. By lemma 4.6 there exists r s.t. $v{\downarrow} \multimap_{\mathtt{dB}} r \to_{\mathtt{ls}}^{*} u$. Moreover, $v \multimap_{\mathtt{ls}}^{*} v{\downarrow}$; call γ this reduction. Let ρ be the reduction obtained by concatenating $\tau : t \multimap^{*} v$, $\gamma : v \multimap_{\mathtt{ls}}^{*} v{\downarrow}$ and $v{\downarrow} \multimap_{\mathtt{dB}} r$. We have $|\rho|_{\mathtt{dB}} = |\tau|_{\mathtt{dB}} + 1 = k$ and $r \to_{\mathtt{ls}}^{*} u$, and so we conclude. ◀

This section and the previous one proved for $\Lambda_{[\cdot]}$ results proved by more abstract means by Melliès in the context of the $\lambda\sigma$-calculus in [16][1]. The linear substitution calculus is simpler than the $\lambda\sigma$-calculus, which is why our analysis — developed independently — is simpler (but also less sophisticated and general) than Melliès'[2].

4.3 Quadratic Relation

The last two sections proved that head reduction can be seen as head linear reduction followed by unfolding, and that the number of head steps is exactly the number of $\multimap_{\mathtt{dB}}$-steps in the corresponding head linear reduction. To conclude that head and head linear reduction are polynomially related, we need to show that the number of $\multimap_{\mathtt{ls}}$-steps in a linear head reduction ρ is polynomially related to the number of $\multimap_{\mathtt{dB}}$-steps in ρ.

We do it by giving a precise estimate of the maximal length of a $\multimap_{\mathtt{ls}}$ reduction from a given term. Intuition tells us that any reduction $t \multimap_{\mathtt{ls}}^{*} u$ cannot be longer than the number of explicit substitutions in t (number noted $\mathtt{es}(t)$), since any substitution in t can act at most once on the head variable. However, a formal proof of this fact is not completely immediate, and requires to introduce a measure and prove some (easy) lemmas.

The idea is to statically count the length of the maximum chain of substitutions on the head, and to show that this number decreases at each head linear substitution step. Let us give an example. Consider the reduction:

$$t = (x\ y)[x/y\ r][y/u] \multimap_{\mathtt{ls}} ((y\ r)\ y)[x/y\ r][y/u] \multimap_{\mathtt{ls}} ((u\ r)\ y)[x/y\ r][y/u].$$

It is easy to establish statically on t that $[y/u]$ will give rise to the second $\multimap_{\mathtt{ls}}$-step, since y is the head variable of $y\ r$, which is what is going to be substituted on the head variable of t, *i.e.* $[y/u]$ is an *hereditary* head substitution of t. We use this idea to define the measure. Note that, according to our reasoning, $[y/u]$ is an hereditary head substitution also for $s = (x\ y)[x/(y\ r)[y/u]]$, but we get around these nested cases because we only have to deal with shallow terms.

▶ **Definition 4.8.** *Hereditary head contexts* are generated by the following grammar:

$$HH := H \mid HH[x][x/H].$$

---

[1] Soundness is Theorem 3 (Section 7) in [16] and completeness follows from the fact that $\lambda\sigma$ enjoys *finite normalization cones*, the abstract rewriting notion studied in that paper.

[2] In particular, it can be shown that $\Lambda_{[\cdot]}$ enjoys pushouts modulo permutation equivalence exactly as the λ-calculus, which implies—as Melliès says in [16]—that its normalization cones are at most singletons (*i.e.* trivial in comparison with those of the $\lambda\sigma$-calculus).

The *head measure* $|t|_{HH}$ of a shallow term t is defined by induction on t:

$$\begin{array}{ll} |x|_{HH} = 0; & |t[x/u]|_{HH} = |t|_{HH}, \text{ if } t \neq HH[x]; \\ |\lambda x.t|_{HH} = |t|_{HH}; & |t[x/u]|_{HH} = |t|_{HH} + 1, \text{ if } t = HH[x]; \\ |t\ u|_{HH} = |t|_{HH}. & \end{array}$$

Please notice that $|t|_{HH} = 0$ for any λ-term t.

Next lemma proves that $|t|_{HH}$ correctly captures the number of $\multimap_{\mathtt{ls}}$-reductions from t, what is then compactly expressed by the successive corollary. The detailed proof of the lemma is in [4].

▶ **Lemma 4.9** ($|\cdot|_{HH}$ Decreases with $\multimap_{\mathtt{ls}}$). *Let t be a shallow term.*
1. *t is a $\multimap_{\mathtt{ls}}$-normal form iff $|t|_{HH} = 0$.*
2. *$t \multimap_{\mathtt{ls}} u$ implies $|t|_{HH} = |u|_{HH} + 1$.*
3. *$|t|_{HH} > 0$ implies that t is not $\multimap_{\mathtt{ls}}$-normal.*

Proof. 1) By induction on t. 2) By induction on $t \multimap_{\mathtt{ls}} u$. 3) By induction on t. ◀

Summing up, we get:

▶ **Corollary 4.10** (Exact Bound to $\multimap_{\mathtt{ls}}$-sequences). *$t \multimap_{\mathtt{ls}}^{n} t\!\downarrow$ iff $n = |t|_{HH}$.*

Proof. By induction on n. For $n = 0$ see Lemma 4.9.1, for $n > 0$ it follows from 4.9.2-3. ◀

Now, we are ready to prove the quadratic relation. The following lemma is the key point for the combinatorial analysis. It shows that if the initial term t of a reduction $\rho : t \multimap^{n} u$ is a λ-term, then $|u|_{HH}$ is bounded by the number of $\multimap_{\mathtt{dB}}$ steps in ρ.

▶ **Lemma 4.11.** *Let $t \in \mathcal{T}_\lambda$. If $\rho : t \multimap^{n} u$ then $|u|_{HH} \leq \mathtt{es}(u) = |\rho|_{\mathtt{dB}}$.*

Proof. Note that by definition of $|\cdot|_{HH}$ we get $|u|_{HH} \leq \mathtt{es}(u)$ for any term u. So we only need prove that $\mathtt{es}(u) = |\rho|_{\mathtt{dB}}$. By induction on $k = |\rho|_{\mathtt{dB}}$. If $k = 0$ then ρ is empty, because t is a λ-term and so it is $\multimap_{\mathtt{ls}}$-normal. Then $t = u$ and $\mathtt{es}(u) = 0$. If $k > 0$ then $\rho = \tau; \multimap_{\mathtt{dB}}; \multimap_{\mathtt{ls}}^{m}$ for some m and some reduction τ. Let r be the end term of τ and s the term s.t. $r \multimap_{\mathtt{dB}} s \multimap_{\mathtt{ls}}^{*} u$. By *i.h.* $\mathtt{es}(r) = |\tau|_{\mathtt{dB}} = |\rho|_{\mathtt{dB}} - 1$. Now, $\mathtt{es}(s) = \mathtt{es}(r) + 1 = |\rho|_{\mathtt{dB}}$, because each $\multimap_{\mathtt{dB}}$-step creates an explicit substitution. By lemma 3.2 we get that any box subterm of s is a box-subterm of t, and since t is a λ-term, the duplication performed by a $\multimap_{\mathtt{ls}}$-step does not increase the number of explicit substitutions. Therefore, $\mathtt{es}(u) = \mathtt{es}(s) = |\rho|_{\mathtt{dB}}$. ◀

We finally get:

▶ **Theorem 4.12.** *Let $t \in \mathcal{T}_\lambda$. If $\rho : t \multimap^{n} u$ then $n = O(|\rho|^2_{\mathtt{dB}})$.*

Proof. There exists $k \in \mathbb{N}$ s.t. $\rho = \tau_1; \gamma_1; \ldots; \tau_k; \gamma_k$, where τ_i is a non-empty $\multimap_{\mathtt{dB}}$-reduction and γ_i is a $\multimap_{\mathtt{ls}}$-reduction for $i \in \{1, \ldots, k\}$ and it is non-empty for $i \in \{1, \ldots, k-1\}$. Let $r_1, \ldots, r_k$ be the end terms of $\tau_1, \ldots, \tau_k$, respectively. By Corollary 4.10 $|\gamma_j| \leq |r_j|_{HH}$ and by Lemma 4.11 $|r_j|_{HH} \leq \sum_{i \in \{1,\ldots,j\}} |\tau_i|$. Now $|\rho|_{\mathtt{dB}} = \sum_{i \in \{1,\ldots,k\}} |\tau_i|$ bounds every $|r_j|_{HH}$, hence:

$$\sum_{i \in \{1,\ldots,k\}} |\gamma_i| \leq \sum_{i \in \{1,\ldots,k\}} |r_i|_{HH} \leq k \cdot |\rho|_{\mathtt{dB}}.$$

But k is bounded by $|\rho|_{\mathtt{dB}}$ too, thus $\sum_{i \in \{1,\ldots,k\}} |\gamma_i| \leq |\rho|^2_{\mathtt{dB}}$ and $n \leq |\rho|^2_{\mathtt{dB}} + |\rho|_{\mathtt{dB}} = O(|\rho|^2_{\mathtt{dB}})$. ◀

Putting together the results from the whole of Section 4, we get:

▶ **Corollary 4.13** (Invariance, Part I). *There is a polynomial time algorithm that, given $t \in \mathcal{T}_\lambda$, computes a term u such that $u\downarrow = r$ if t has $\to_{\mathtt{h}}$-normal form r and diverges if u has no $\to_{\mathtt{h}}$-normal form. Moreover, the algorithm works in polynomial time on the unitary cost of the input term.*

One may now wonder why a result like Corollary 4.13 cannot be generalized to, e.g., leftmost-outermost reduction, which is a normalizing strategy. Actually, linear explicit substitutions *can* be endowed with a notion of reduction by levels capable of simulating the leftmost-outermost strategy in the same sense as linear head-reduction simulates head-reduction here. And, noticeably, the subterm property *continues* to hold. What is not true anymore, however, is the quadratic bound we have proved in this section: in the leftmost-outermost linear strategy, one needs to perform *too many* substitutions not related to any β-redex. If one wants to generalize Corollary 4.13, in other words, one needs to further optimize the substitution process. But this is outside the scope of this paper.

5 $\Lambda_{[\cdot]}$ as an Acceptable Encoding of λ-terms

The results of the last two sections can be summarized as follows: linear explicit substitutions provide both a compact representation for λ-terms and an efficient implementation of β-reduction. Here, efficiency means that the overhead due to substitutions remains under control and is at most polynomial in the unitary cost of the λ-term we start from. But one may wonder whether explicit substitutions are nothing more than a way to *hide* the complexity of the problem under the carpet of compactness: what if we want to get the normal form in the *usual*, *explicit* form? Counterexamples from Section 2, read through the lenses of Theorem 4.13 tell us that that this is *indeed* the case: there are families of λ-terms with polynomial unitary cost but whose normal form intrinsically requires exponential time to be produced.

In this section, we show that this phenomenon is due to the λ-calculus being a very inefficient way to represent λ-terms: even if computing the unfolding of a term $t \in \Lambda_{[\cdot]}$ takes exponential time, *comparing* the unfoldings of two terms $t, u \in \Lambda_{[\cdot]}$ for equality can be done in polynomial time in $|t| + |u|$. This way, linear explicit substitutions are proved to be a succint, acceptable encoding of λ-terms in the sense of Papadimitriou [18]. The algorithm we are going to present is based on dynamic programming: it compares all the *relative unfoldings* of subterms of two terms t and u as above, without really computing those unfoldings. Some complications arise due to the underlying notion of equality for λ-terms, namely α-equivalence, which is coarser than syntactical equivalence. But what is the relative unfolding of a term?

▶ **Definition 5.1** (Relative Unfoldings). The unfolding $t\downarrow_C$ of t relative to context C is defined by induction on C:

$$t\downarrow_{[\cdot]} := t\downarrow; \qquad t\downarrow_{u\ C} := t\downarrow_C; \qquad t\downarrow_{u[x/C]} := t\downarrow_C;$$
$$t\downarrow_{C\ u} := t\downarrow_C; \qquad t\downarrow_{\lambda x.C} := t\downarrow_C; \qquad t\downarrow_{C[x/u]} := t\downarrow_C\{x/u\downarrow\}.$$

Constraining sets allow to give sensible judgments about the equivalence of terms even when their free variables differ:

▶ **Definition 5.2** (Constraining Sets and Coherence). A *constraining set* A is a set of pairs (x, y) of variable names. Two constraining sets A and B are *coherent* (noted $A \sim B$) if:

- $(x, y) \in A$ and $(x, z) \in B$ imply $y = z$;
- $(y, x) \in A$ and $(z, x) \in B$ imply $y = z$.

Moreover, A is *auto-coherent* if $A \sim A$. Observe that $\sim$ is not reflexive and that a constraining set is auto-coherent iff it is the graph of a bijection.

As an example $A = \{(x, y), (x, z)\}$ is not auto-coherent, while $B = \{(w, u)\}$ is auto-coherent, like any singleton set of pairs of variables. Moreover, $A \sim B$.

The algorithm we are going to define takes in input a pair (a, b) of terms. We assume them *preprocessed* as follows: the spaces of substituted, abstracted and free names of a and b are all pairwise disjoint and neither a nor b contain any subterm in the form $c[x/d]$, where $x \notin \mathtt{fv}(c)$[3]. We also note $\mathcal{S}$ the set of substituted variables of both terms. The whole algorithm is built around the notion of an unfolding judgment:

▶ **Definition 5.3** (Unfolding Judgments and Unfolding Rules)**.** Let $P = (a, b)$ be a preprocessed pair. An *unfolding judgement* is a triple $(t, C), v, (u, D)$, where the *value* v is either $\bot$ or a constraining set A, also noted $(t, C) \overset{v}{\sim} (u, D)$. The rules for deriving unfolding judgments are in Figure 2. An operation $\diamond$ on values is used, which is defined as follows:

1. $v \diamond w = v \cup w$ if $v \neq \bot \neq w$ and $v \sim w$;
2. $v \diamond w = \bot$ if $v \neq \bot \neq w$ and $v \not\sim w$;
3. $v \diamond w = \bot$ if $v = \bot$ or $w = \bot$.

The rules in Figure 2 induce a binary relation $\sqsubset_{a,b}$ on the space of pairs (of pairs) in the form $((t, C), (u, D))$ such that $C[t] = a$ and $D[u] = b$: $(P, Q) \sqsubset_{a,b} (R, S)$ if knowing v such that $P \overset{v}{\sim} Q$ is necessary to compute w such that $R \overset{w}{\sim} S$.

The meaning of unfolding judgements can be summarized as follows. If $(t, C) \overset{A}{\sim} (u, D)$, then $r = t{\downarrow_C}$ and $s = u{\downarrow_D}$ are α-equivalent, provided free variables of r are modified according to A, seen as a substitution (or, analogously, free variables of s are modified according to A^{-1}). If, instead, $(t, C) \overset{\bot}{\sim} (u, D)$, there cannot be any substitution like A above. The rules in Figure 2 are not only a sound but also complete way of checking equality of terms in the just described sense.

Given a preprocessed pair (a, b), checking the equality of $a{\downarrow}$ and $b{\downarrow}$ through a direct use of the formal system above immediately leads to an exponential blowup, which however can be avoided by way of dynamic programming.

An unfolding matrix for a preprocessed pair (a, b) is the data structure storing intemediate judgements produced while deriving a judgement $(a, [\cdot]) \overset{v}{\sim} (b, [\cdot])$. Relying on unfolding matrices ultimately allows to avoid repeating the same computation many times. Formally, an *unfolding matrix* for a preprocessed pair (a, b) is a bidimensional array whose rows are indexed by pairs $(t, C[\cdot])$ such that $C[t] = a$ and whose columns are indexed by pairs $(t, C[\cdot])$ such that $C[t] = b$. Each element of the unfolding matrix can be either a (possibly empty) constraining sets or $\bot$ or $\#$.

Basically, the unfolding checking algorithm simply proceeds by filling an unfolding matrix with the correct values, following the rules in Figure 2 and starting from an unfolding matrix filled with $\#$:

▶ **Definition 5.4** (Unfolding Checking Algorithm)**.** Let $P = (a, b)$ be a preprocessed pair. We define the following algorithm, which will be referred to as the *unfolding checking algorithm*:

[3] Any term can be turned in this form in polynomial time, by $\to_{\mathtt{gc}}$-normalizing it.

Positive Rules

$$\frac{x,y \notin \mathcal{S}}{(x,C) \overset{\{(x,y)\}}{\sim} (y,D)}\ \mathtt{var} \qquad \frac{(t,C[[\cdot]\ r]) \overset{v}{\sim} (u,D[[\cdot]\ s]) \qquad (r,C[t\ [\cdot]]) \overset{w}{\sim} (s,D[u\ [\cdot]])}{(t\ r,C) \overset{v \diamond w}{\sim} (u\ s,D)}\ @$$

$$\frac{(t,C[E[x][x/[\cdot]]]) \overset{v}{\sim} (u,D)}{(x,C[E[x/t]]) \overset{v}{\sim} (u,D)}\ \mathtt{unf}_l \qquad \frac{(t,C) \overset{v}{\sim} (u,D[E[x][x/[\cdot]]])}{(t,C) \overset{v}{\sim} (x,D[E[x/u]])}\ \mathtt{unf}_r$$

$$\frac{(t,C[\lambda x.[\cdot]]) \overset{v}{\sim} (u,D[\lambda y.[\cdot]]) \qquad (x,y) \in v}{(\lambda x.t,C) \overset{v \setminus \{(x,y)\}}{\sim} (\lambda y.u,D)}\ \lambda_1 \qquad \frac{(t,C[[\cdot][x/r]]) \overset{v}{\sim} (u,D)}{(t[x/r],C) \overset{v}{\sim} (u,D)}\ \mathtt{sub}_l$$

$$\frac{(t,C[\lambda x.[\cdot]]) \overset{v}{\sim} (u,D[\lambda y.[\cdot]]) \qquad \forall z.(x,z),(z,y) \notin v}{(\lambda x.t,C) \overset{v}{\sim} (\lambda y.u,D)}\ \lambda_2 \qquad \frac{(t,C) \overset{v}{\sim} (t,D[[\cdot][x/r]])}{(t,C) \overset{v}{\sim} (u[x/r],D)}\ \mathtt{sub}_r$$

Error Generation Rules

$$\frac{(t,C[\lambda x.[\cdot]]) \overset{v}{\sim} (u,D[\lambda y.[\cdot]]) \qquad (\exists z \neq y.\{(x,z)\} \in v) \vee (\exists z \neq x.\{(z,y)\} \in v) \vee v = \bot}{(\lambda x.t,C) \overset{\bot}{\sim} (\lambda y.u,D)}\ \lambda_3$$

$$\frac{}{(\lambda x.t,C) \overset{\bot}{\sim} (u\ r,D)}\ \mathtt{err}_{\lambda @} \qquad \frac{}{(u\ r,C) \overset{\bot}{\sim} (\lambda x.t,D)}\ \mathtt{err}_{@\lambda} \qquad \frac{x \notin \mathcal{S}}{(x,C) \overset{\bot}{\sim} (u\ r,D)}\ \mathtt{err}_{x@}$$

$$\frac{x \notin \mathcal{S}}{(\lambda y.t,C) \overset{\bot}{\sim} (x,D)}\ \mathtt{err}_{\lambda x} \qquad \frac{x \notin \mathcal{S}}{(x,C) \overset{\bot}{\sim} (\lambda y.t,D)}\ \mathtt{err}_{x\lambda} \qquad \frac{x \notin \mathcal{S}}{(u\ r,C) \overset{\bot}{\sim} (x,D)}\ \mathtt{err}_{@x}$$

Figure 2 Unfolding Rules.

1. Initialize all entries in $\mathtt{M}$ to #;
2. If $\mathtt{M}$ has no entries filled with #, then go to step 7;
3. Choose (t,C) and (u,D) such that $\mathtt{M}[(t,C)][(u,D)] = \#$, and such that $\mathtt{M}[P_1][Q_1], \ldots, \mathtt{M}[P_n][Q_n]$ are all different from #, where $(P_1,Q_1),\ldots,(P_n,Q_n)$ are the immediate predecessors of $((t,C),(u,D))$ in $\sqsubset_{a,b}$;
4. Compute v such that $(t,C) \overset{v}{\sim} (u,D)$;
5. $\mathtt{M}[(t,C)][(u,D)] \leftarrow v$;
6. Go to step 2;
7. Return `yes` if $\mathtt{M}[(a,[\cdot])][(b,[\cdot])]$ is a constraining set which is the identity, otherwise return `no`.

The Unfolding Checking Algorithm has a simple structure: initially $\mathtt{M}$ only contains #, and steps $2-5$ are repeated until $\mathtt{M}$ does not contain any #. At each iteration, a cell containing # is assigned a value v, while the rest of $\mathtt{M}$ is left unchanged.

It is now time to prove that the Unfolding Checking Algorithm is correct, i.e., that it gives correct results, if any:

▶ **Theorem 5.5** (Correctness). *The Unfolding Checking Algorithm, on input (a,b), returns* `yes` *iff* $a{\downarrow} = b{\downarrow}$.

Proof. This can be done by proving the following two lemmas:

- If $(t,C) \overset{v}{\sim} (u,D)$ and $v \neq \bot$ then v induces two renamings σ and θ such that $t{\downarrow_C}\sigma = u{\downarrow_D}$ and $u{\downarrow_D}\theta = t{\downarrow_C}$;
- If $(t,C) \overset{\bot}{\sim} (u,D)$, then there is no such pair of renamings for $t{\downarrow_C}$ and $u{\downarrow_D}$.

Both can be proved by induction on the derivation of $(t,C) \overset{A}{\sim} (u,D)$. For details, see [4]. ◀

This is not the end of the story, however — one also needs to be sure about the time complexity of the algorithm, which turns out to be polynomial:

▶ **Proposition 5.6** (Complexity). *The Unfolding Checking Algorithm works in time polynomial in* $|a| + |b|$.

Proof. The following observations are sufficient to obtain the thesis:

- The number of entries in `M` is $|a||b|$ in total.
- At every iteration, one element of `M` changes its value from # to some non-blank v.
- Step 2 can clearly be performed in time polynomial in $|a| + |b|$.
- Computing the predecessors of a pair P can be done in polynomial time, and so Step 3 can itself be performed in time polynomial in $|a| + |b|$.
- Rules in Figure 2 can all be applied in polynomial time, in particular because the cardinality of any constraining set produced by the algorithm can be $|a| + |b|$, at most. As a consequence, Step 4 can be performed in polynomial time.

This concludes the proof. ◀

6 Encoding Turing Machines

A cost model *for computation time* is said to be invariant if it is polynomially related to the standard cost model on Turing machines. In sections 3 and 4, we proved that head reduction of any λ-term t can be performed on a Turing machine in time polynomial in the number of β-steps leading t to its normal form (provided it exists). This is proved through explicit substitutions, which in Section 5 are shown to be a reasonable representation for λ-terms: two terms t and u in $\Lambda_{[\cdot]}$ can be checked to have α-equivalent unfoldings in polynomial time.

The last side of the story is still missing, however. In this section, we will show how head reduction can simulate Turing machine computation in such a way that the unitary cost of the simulating λ-term is polynomially related to the running time of the encoded Turing machine. Results similar to the one we are going to present are common for the λ-calculus, so we will not give all the details, which can anyway be found in [4].

The simplest objects we need to encode in the λ-calculus are finite sets. Elements of any finite set $A = \{a_1, \dots, a_n\}$ can be encoded as follows: $\lceil a_i \rceil^A \equiv \lambda x_1 . \dots . \lambda x_n . x_i$. Notice that the above encoding induces a total order on A such that $a_i \leq a_j$ iff $i \leq j$. Other useful objects are finite strings over an arbitrary alphabet, which will be encoded using a scheme attributed to Dana Scott. Let $\Sigma = \{a_1, \dots, a_n\}$ be a finite alphabet. A string in $s \in \Sigma^*$ can be represented by a value $\lceil s \rceil^{\Sigma^*}$ as follows, by induction on the structure of s:

$$\lceil \varepsilon \rceil^{\Sigma^*} \equiv \lambda x_1 . \dots . \lambda x_n . \lambda y . y; \qquad \lceil a_i r \rceil^{\Sigma^*} \equiv \lambda x_1 . \dots . \lambda x_n . \lambda y . x_i \lceil r \rceil^{\Sigma^*}.$$

Observe that representations of symbols in Σ and strings in Σ^* depend on the cardinality of Σ. In other words, if $s \in \Sigma^*$ and $\Sigma \subset \Delta$, $\lceil s \rceil^{\Sigma^*} \neq \lceil s \rceil^{\Delta^*}$.

Of course, one should be able to very easily compute the encoding of a string obtained by concatenating another string with a character. Moreover, the way strings are encoded depends on the underlying alphabet, and as a consequence, we also need to be able to convert representations for strings in one alphabet to corresponding representations in another, different, alphabet. This can be done efficiently in the λ-calculus by way of a term $AC(\Sigma)$ which append a character to a string (both expressed in the alphabet Σ) and a term $CS(\Sigma, \Delta)$ which converts a string $s \in \Sigma^*$ into another string in Δ^* obtained by replacing any character in $\Sigma - \Delta$ by the empty string. $AC(\Sigma)$ works in time independent on the size of the input, while $CS(\Sigma, \Delta)$ works in time *proportional* to the size of the argument.

A configuration (u, a, v, q) of a Turing machine $\mathcal{M} = (\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ consists of the portion of the tape on the left of the head (namely $u \in \Sigma^*$), the symbol under the head ($a \in \Sigma$), the tape on the right of the head ($v \in \Sigma^*$), and the current state $q \in Q$. Such a configuration is represented by the term $\lceil (u, a, v, q) \rceil^{\mathcal{M}} \equiv \lambda x.x \lceil u^r \rceil^{\Sigma^*} \lceil a \rceil^{\Sigma} \lceil v \rceil^{\Sigma^*} \lceil q \rceil^{Q}$.

We now encode a Turing machine $\mathcal{M} = (\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ in the λ-calculus. Suppose $\Sigma = \{a_1, \ldots, a_{|\Sigma|}\}$ and $Q = \{q_1, \ldots, q_{|Q|}\}$. The encoding of $\mathcal{M}$ is defined around three λ-terms:

- First of all, we need to be able to build the initial configuration for u from u itself. This can be done in time proportional to $|u|$ by a term $\mathcal{I}(\mathcal{M}, \Delta)$, where Δ is the alphabet of u, which can be different from Σ. $\mathcal{I}(\mathcal{M}, \Delta)$ simply converts u into the appropriate format by way of $CS(\Delta, \Sigma)$, and then packages it into a configuration.
- Then, we need to extract a string from a final configuration C for the string. This can be done in time proportional to the size of C by a term $\mathcal{F}(\mathcal{M}, \Delta)$, which makes essential use of $CS(\Sigma, \Delta)$.
- Most importantly, we need to be able to simulate the transition function of $\mathcal{M}$, i.e. compute a final configuration from an initial configuration (if it exists). This can be done with cost proportional to the number of steps $\mathcal{M}$ takes on the input, by way of a term $\mathcal{T}(\mathcal{M})$. The term $\mathcal{T}(\mathcal{M})$ just performs case analysis depending on the four components of the input configuration, then manipulating them making use of $AC(\Sigma)$.

A peculiarity of the proposed encoding of Turing machines is the use of Scott numerals (instead of Church numerals), which make the simulation work even when head reduction is the underlying notion of computation. Another crucial aspect is *continuation passing*: the λ-terms cited above all take an additional parameter to which the result is applied after being computed. At this point, we can give the desired simulation result:

▶ **Theorem 6.1** (Invariance, Part II). *If $f : \Delta^* \to \Delta^*$ is computed by a Turing machine $\mathcal{M}$ in time g, then there is a term $\mathcal{U}(\mathcal{M}, \Delta)$ such that for every $u \in \Delta^*$, $\mathcal{U}(\mathcal{M}, \Delta) \lceil u \rceil^{\Delta^*} \to_{\mathtt{h}}^{n} \lceil f(u) \rceil^{\Delta^*}$ where $n = \mathcal{O}(g(|u|) + |u|)$.*

Proof. Simply define $\mathcal{U}(\mathcal{M}, \Delta) \equiv \lambda u.\mathcal{I}(\mathcal{M}, \Delta)(\lambda x.\mathcal{T}(\mathcal{M})(\lambda y.\mathcal{F}(\mathcal{M}, \Delta)(\lambda w.w)y)x)u$. It is routine to prove the thesis. ◀

Noticeably, the just described simulation induces a linear overhead: every step of $\mathcal{M}$ corresponds to a constant cost in the simulation, the constant cost not depending on the input but only on $\mathcal{M}$ itself.

7 Conclusions

The main result of this paper is the first invariance result for the λ-calculus when reduction is allowed to take place in the scope of abstractions. The key tool to achieve invariance are linear explicit substitutions, which are *compact* but *manageable* representations of λ-terms.

Of course, the main open problem in the area, namely invariance of the unitary cost model for any normalizing strategy (e.g. for the strategy which always reduces the leftmost-outermost redex) remains open. Although linear explicit substitutions cannot be *directly* applied to this problem, the authors strongly believe that this is anyway a promising direction, on which they are actively working at the time of writing.

References

1 M. Abadi, L. Cardelli, P. L. Curien, and J. J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1:31–46, 1991.

2 B. Accattoli. An abstract factorization theorem for explicit substitutions. Accepted at RTA 2012.

3 B. Accattoli. *Jumping around the box: graphical and operational studies on Lambda Calculus and Linear Logic.* Ph.D. Thesis, Università di Roma La Sapienza, 2011.

4 B. Accattoli and U. Dal Lago. On the invariance of the unitary cost model for head reduction (long version). Available at `http://arxiv.org/abs/1202.1641`.

5 B. Accattoli and S. Guerrini. Jumping boxes. In *Proceedings of CSL 2009*, pages 55–70, 2009.

6 B. Accattoli and D. Kesner. The structural λ-calculus. In *Proceedings of CSL 2010*, volume 6247 of *LNCS*, pages 381–395. Springer, 2010.

7 B. Accattoli and D. Kesner. The permutative -calculus, 2012.

8 M. Avanzini and G. Moser. Complexity analysis by graph rewriting. In *Proceedings of FLOPS 2010*, volume 6009 of *LNCS*, pages 257–271. Springer, 2010.

9 Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and polytime computability. In *Proceedings of RTA 2010*, volume 6 of *LIPIcs*, pages 33–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

10 U. Dal Lago and S. Martini. The weak lambda-calculus as a reasonable machine. *Theoretical Computer Science*, 398:32–50, 2008.

11 U. Dal Lago and S. Martini. On constructor rewrite systems and the lambda-calculus. In *Proceedings of ICALP 2009*, volume 5556 of *LNCS*, pages 163–174. Springer, 2009.

12 U. Dal Lago and S. Martini. Derivational complexity is an invariant cost model. In *Proceedings of FOPARA 2010*, volume 6324 of *LNCS*, pages 88–101. Springer, 2010.

13 V. Danos and L. Regnier. Head linear reduction. Submitted. Available at `http://iml.univ-mrs.fr/~regnier/articles/pam.ps.gz`, 2004.

14 G. Mascari and M. Pedicini. Head linear reduction and pure proof net extraction. *Theoretical Computer Science*, 135(1):111–137, 1994.

15 P.-A. Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In *Proceedings of TLCA 1995*, volume 902 of *LNCS*, pages 328–334. Springer, 1995.

16 P.-A. Melliès. Axiomatic rewriting theory II: the $\lambda\sigma$-calculus enjoys finite normalisation cones. *Journal of Logic and Computation*, 10(3):461–487, 2000.

17 R. Milner. Local bigraphs and confluence: Two conjectures: (extended abstract). *ENTCS*, 175(3):65–73, 2007.

18 C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

19 G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

20 L. Regnier. *Lambda-calcul et réseaux.* Thèse de doctorat, Univ. Paris VII, 1992.

21 D. Sands, J. Gustavsson, and A. Moran. Lambda calculi and linear speedups. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in LNCS, pages 60–82. Springer, 2002.

22 K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3-4):253–280, 2007.

23 P. van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. MIT Press, 1990.

Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of RTA 2012 (LIPIcs, volume 15, http://www.dagstuhl.de/dagpub/978-3-939897-38-5, published in May, 2012), in which rules $\mathtt{unf}_l$, $\mathtt{unf}_r$, λ_1, $\mathtt{sub}_l$, λ_2 and $\mathtt{sub}_r$ were erroneously absent from Figure 2 (on page 34).

Dagstuhl Publishing – October 5, 2012.