On the Invariance of the Unitary Cost Model for Head Reduction (Long Version)

Beniamino Accattoli Ugo Dal Lago

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 derivation complexity is an invariant cost model, if any.

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 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 [21]: any time cost model is reasonable if it is polynomially related to the (standard) one of Turing machines.

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 [11, 8].

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 [10], or by other means [19]. 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 [14]. Starting from graphical syntaxes, a new *at a distance* approach to explicit substitutions has recently been proposed [7]. 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 [7, 6].

In this paper we use the *linear-substitution calculus* $\Lambda_{[\cdot]}$, a slight variation over a calculus of explicit substitutions introduced by Robin Milner [15]. The variation is inspired by the structural λ-calculus [7]. We study in detail the relation between λ-calculus head reduction and *linear head reduction* [13], the head reduction of $\Lambda_{[\cdot]}$, and prove that the latter can be 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 $t \multimap^* u$ duplicates 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 firsts to use the fine granularity of explicit substitutions for complexity analysis. Many calculi with bounded complexity (e.g. [20]) 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.

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 captured 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:

$$
\begin{aligned}
&t_0 \equiv u \equiv r_0\\
&t_1 \to yuu \equiv yr_0yr_0 \equiv r_1\\
&t_2 \to (\lambda x.t_0)(yuu) \equiv (\lambda x.u)(r_1) \to yr_1r_1 \equiv r_2\\
&\quad\vdots
\end{aligned}
$$

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-step 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 $s = \lambda y.yxx$, then define v_0 to be just $\lambda x.s$ and for every $n \in \mathbb{N}$, v_{n+1} to be $\lambda x.v_n s$. 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 [9]. 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 [17]. In one way or another, the mentioned results are based on sharing subterms, either by translating the λ-calculus to a TRS [10] or by going through abstract machines [19]. 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-abstraction, 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 an essential way.

3 Linear Explicit Substitutions

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

$$t, u, r \in \mathcal{T}_\lambda :: x \mid \mathcal{T}_\lambda\ \mathcal{T}_\lambda \mid \lambda x.\mathcal{T}_\lambda$$

and its reduction rule $\to_\beta$ is defined as the context closure of $(\lambda x.t)\ u \mapsto_\beta t\{x/u\}$. 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:

$$\hat{H} ::= [\cdot] \mid \hat{H}\ \mathcal{T}_\lambda \mid \lambda x.\hat{H}$$

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

Lemma 1 *$\to_{\mathtt{h}}$ has the diamond property, namely if $t \to_{\mathtt{h}} u_i$ with $i = 1, 2$ then there exists r such that $u_i \to_{\mathtt{h}} r$.*

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 [15]. The grammar is standard:

$$t, u, r \in \mathcal{T} :: x \mid \mathcal{T}\ \mathcal{T} \mid \lambda x.\mathcal{T} \mid \mathcal{T}[x/\mathcal{T}]$$

The term $t[x/u]$ is an *explicit substitution*. Both costructors $\lambda x.t$ and $t[x/u]$ bind x in t. We note $\mathtt{L}$ a possibly empty list of explicit substitutions $[x_1/u_1]\ldots[x_k/u_k]$. Contexts are defined by:

$$C, D, E, F :: [\cdot] \mid C\ \mathcal{T} \mid \mathcal{T}\ C \mid \lambda x.C \mid C[x/\mathcal{T}] \mid \mathcal{T}[x/C]$$

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 closure 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 $\Lambda_{[\cdot]}$ [15].

The linear-substitution calculus enjoys all properties required to explicit substitutions calculi, obtained by easy adaptations of the proofs for $\lambda\mathtt{j}$ in [7]. 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 [18] 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 derelictions were always contracted with a weakening.

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

Figure 1: $\Lambda_{[\cdot]}$ rewriting rules

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* [13, 12], 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 1 (head context) Head contexts *are defined by the following grammar:*

$$H ::= [\cdot] \mid H\ \mathcal{T} \mid \lambda x.H \mid H[x/\mathcal{T}].$$

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 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 $\mapsto_{\mathtt{ls}}$:

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

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:

Lemma 2 $\multimap$ *has the diamond property.*

Proof. The only possible critical pair is given by

$$\begin{tikzcd}
H_1[\,(\lambda y.\ H_2[x]\,)\mathtt{L}\ v\,]\,[x/u] \arrow[r, "\multimap" description, phantom] \arrow[d, -o] & H_1[\,(\lambda y.H_2[u])\mathtt{L}\ v\,][x/u] \arrow[d, -o] \\
H_1[\,H_2[x]\,[y/v]\mathtt{L}]\,[x/u] \arrow[r, "\multimap" description, phantom] & H_1[H_2[u][y/v]\mathtt{L}][x/u]
\end{tikzcd}$$

There is also the possibility that a $\multimap$-redex R is contained into another $\multimap$-redex R', but then R' cannot duplicate R:

$$\begin{tikzcd}
H_1[\,(\lambda y.H_2[\,(\lambda x.t)\mathtt{L}_2\ u\,])\mathtt{L}_1\ v\,] \arrow[r, "\multimap" description, phantom] \arrow[d, -o] & H_1[H_2[\,(\lambda x.t)\mathtt{L}_2\ u\,][y/v]\mathtt{L}_1] \arrow[d, -o] \\
H_1[\,(\lambda y.H_2[t[x/u]\mathtt{L}_2])\mathtt{L}_1\ v\,] \arrow[r, "\multimap" description, phantom] & H_1[H_2[t[x/u]\mathtt{L}_2][y/v]\mathtt{L}_1]
\end{tikzcd}$$

There are no other cases, in particular no term can have two disjoint $\multimap$-redexes. □

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 1 *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.*

Proposition 3 (Subterm Property) *If* $t \multimap^* u$ *and* r *is a box-suterm 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 sub-terms 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. By induction on the length k of the reduction $t \multimap^* u$. Suppose $k > 0$. 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 $v \multimap u$:

- Base cases:
  - $v = (\lambda x.r)\mathtt{L}\ s \multimap r[x/s]\mathtt{L} = u$: it is evident that the two terms have the same box-subterms.
  - $v = H[x][x/s] \multimap H[s][x/s] = u$: by a 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.
- Inductive cases: just use the *i.h.* and the previous remark.

$\square$

The subterm property does not only concern the cost of implementing reduction steps, but also the size of the end term:

Corollary 4 *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 2 (Shallow Terms) *A $\Lambda_{[\cdot]}$-term t is* shallow *if whenever $t = C[u[x/r]]$ then r is a λ-term.*

Corollary 5 *Let t be a λ-term and $t \multimap^* u$. Then u is shallow.*

Proof. By lemma 3 the content of any substitution of u is a subterm of t, *i.e.* a λ-term. $\square$

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

In this section, linear explicit substitutions will be showed 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 6 *Let $t \in \mathcal{T}_\lambda$. If $t \to_{\mathtt{h}} u$ then $t\{x/r\} \to_{\mathtt{h}} u\{x/r\}$.*

Lemma 7 (Projection of $\multimap$ on $\to_{\mathtt{h}}$) *Let $t \in \mathcal{T}$. If $\rho : t \multimap^k u$ then $t\downarrow \to^n_{\mathtt{h}} u\downarrow$ 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 $m + 1 = |\tau|_{\mathtt{dB}} + 1 = |\rho|_{\mathtt{dB}} \leq k$.
  By induction on H. We use the following notation: if $t = u\mathtt{L}$ and $\mathtt{L} = [y_1/w_1] \ldots [y_m/w_m]$ then we write $\sigma_{\mathtt{L}}$ for $\{y_1/w_1{\downarrow}\} \ldots \{y_m/w_m{\downarrow}\}$, thus we can write $t{\downarrow} = u{\downarrow}\sigma_{\mathtt{L}}$. Cases:
  - $H = [\cdot]$. Then:

$$
\begin{array}{rcll}
r{\downarrow} & = & ((\lambda x.v)\mathtt{L}){\downarrow}\ s{\downarrow} \quad = & \text{(def. of } (\cdot){\downarrow}) \\
 & & (\lambda x.v{\downarrow})\sigma_{\mathtt{L}}\ s{\downarrow} \quad = & \\
 & & (\lambda x.v{\downarrow}\sigma_{\mathtt{L}})\ s{\downarrow} \quad \to_{\mathtt{h}} & \\
 & & v{\downarrow}\sigma_{\mathtt{L}}\{x/s{\downarrow}\} \quad = & x \notin \mathtt{fv}(w_i) \text{ for } i \in \{1, \ldots, m\} \\
 & & v{\downarrow}\{x/s{\downarrow}\}\sigma_{\mathtt{L}} \quad = & \text{(def. of } (\cdot){\downarrow}) \\
 & & (v[x/s]){\downarrow}\sigma_{\mathtt{L}} \quad = & \text{(def. of } (\cdot){\downarrow}) \\
 & & (v[x/s]\mathtt{L}){\downarrow} \quad = u{\downarrow} &
\end{array}
$$

  - $H = H'\ w$. Then

$$
\begin{array}{rclcll}
r{\downarrow} & = & (H'[(\lambda x.v)\mathtt{L}\ s]){\downarrow}\ w{\downarrow} & \to_{\mathtt{h}} & & (\textit{i.h.}) \\
 & & (H'[v[x/s]\mathtt{L}]){\downarrow}\ w{\downarrow} & = & u{\downarrow} &
\end{array}
$$

  - $H = \lambda y.H'$. Then

$$
\begin{array}{rclcll}
r{\downarrow} & = & \lambda y.(H'[(\lambda x.v)\mathtt{L}\ s]){\downarrow} & \to_{\mathtt{h}} & & (\textit{i.h.}) \\
 & & \lambda y.(H'[v[x/s]\mathtt{L}]){\downarrow} & = & u{\downarrow} &
\end{array}
$$

  - $H = H'[y/w]$. Then

$$
\begin{array}{rclcll}
r{\downarrow} & = & (H'[(\lambda x.v)\mathtt{L}\ s]){\downarrow}\{y/w{\downarrow}\} & \to_{\mathtt{h}} & & (\textit{i.h. \& l.}6) \\
 & & (H'[v[x/s]\mathtt{L}]){\downarrow}\{y/w{\downarrow}\} & = & u{\downarrow} &
\end{array}
$$

□

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 last 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 8 (Head Simulation) *Let t be a λ-term. Then $t \to_{\mathtt{h}} u$ 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 $\to_{\mathtt{ls}}^{*}$ can be factored into some head-linear substitutions, followed by non-head-linear ones. This allows to refine Lemma 8. 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* $t{\downharpoonleft}_{\circ}$ 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). We also need the following abstract lemma about postponement of reductions:

Lemma 9 *Define $\to_{1,2}$ as $\to_1 \cup \to_2$. Then:*

1. *If $\to_2^{*}\to_1 \subseteq \to_1^{*}\to_2^{*}$ then $\to_{1,2}^{*} \subseteq \to_1^{*}\to_2^{*}$.*

2. *(Geser-Di Cosmo-Piperno) If* $\rightarrow_2\rightarrow_1\subseteq\rightarrow_1^+\rightarrow_2^*$ *and* $\rightarrow_1$ *is strongly normalising then* $\rightarrow_2^*\rightarrow_1^+\subseteq\rightarrow_1^+\rightarrow_2^*$*, and so* $\rightarrow_{1,2}^*\subseteq\rightarrow_1^*\rightarrow_2^*$.

Proof. 1. By induction on the number k of $\rightarrow_1$ steps in $\tau : t \rightarrow_{1,2}^* u$. The case $k = 0$ is trivial. Let $k > 0$. Then if τ is not of the form of the statement it has the following form:

$$t \rightarrow_1^*\rightarrow_2^+\rightarrow_1\rightarrow_{1,2}^* u$$

Using the hypothesis we get:

$$t \rightarrow_1^*\rightarrow_1^*\rightarrow_2^*\rightarrow_{1,2}^* u$$

The *i.h.* on $\rightarrow_2^*\rightarrow_{1,2}^*$ gives:

$$t \rightarrow_1^*\rightarrow_1^*\rightarrow_1^*\rightarrow_2^* u$$

and we conclude.

2. We prove the first consequence, the second one follows from the first and Point 1. Let τ : $t \rightarrow_2^k\rightarrow_1^h u$. By induction on the pair $(\eta(t), k)$, where $\eta(t)$ is the length of the maximal $\rightarrow_1$ reduction from t, ordered lexicographically. The case $k = 0$ is trivial, then let $k > 0$. Now, if $\tau =\rightarrow_2^{k-1}\rightarrow_2\rightarrow_1\rightarrow_1^{h-1}$ then by applying the first hypothesis to the central subesequence we get $\tau \subseteq\rightarrow_2^{k-1}\rightarrow_1^+\rightarrow_2^*\rightarrow_1^{h-1}$. The measure of the prefix $\rightarrow_2^{k-1}\rightarrow_1^+$ is $(\eta(t), k - 1)$ hence by *i.h.* we get $\tau \subseteq\rightarrow_1^+\rightarrow_2^*\rightarrow_2^*\rightarrow_1^{h-1}$. For the suffix $\rightarrow_2^*\rightarrow_2^*\rightarrow_1^{h-1}$ it is the first component of the measure which decreases, since its starting term is obtained through a $\rightarrow_1^+$-reduction from t, so one can apply the *i.h.* and get $\tau \subseteq\rightarrow_1^+\rightarrow_1^+\rightarrow_2^*$.

□

Now, we can prove that any $\rightarrow_{\mathtt{h}}$ 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 10 (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. $\rightarrow_{\mathtt{ls}}^*\subseteq\multimap_{\mathtt{ls}}^*\Rightarrow_{\mathtt{s}}^*$*, and in particular* $t \multimap_{\mathtt{ls}}^* t{\downarrow_{\circ}} \Rightarrow_{\mathtt{s}}^* t{\downarrow}$

Proof. 1) Formally, the proof is by induction on $t \Rightarrow_{\mathtt{s}} u$, see [2] for the technical details (and more generally for a detailed study of redex permutations in $\Lambda_{[\cdot]}$). 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:
1. The two redexes simply permute.
2. The preponement of $\multimap_{\mathtt{ls}}$ duplicate the redex reduced by $\Rightarrow_{\mathtt{s}}$. Two subcases:
   1. One of the two residuals is a linear head redex. For instance, consider the diagram:

$$\begin{tikzcd}
x[x/y][y/z] \arrow[rr, Rightarrow, "\mathtt{s}"] \arrow[d, "\mathtt{ls}"] & \arrow[d, phantom, "\swarrow"] & x[x/z][y/z] \arrow[d, "\mathtt{ls}"] \\
y[x/y][y/z] \arrow[r, "\mathtt{ls}"] & z[x/y][y/z] \arrow[r, Rightarrow, "\mathtt{s}"] & z[x/z][y/z]
\end{tikzcd}$$

   2. Both residuals are non-linear-head redexes:

$$\begin{tikzcd}
x[x/y\ y][y/z] \arrow[rr, Rightarrow, "\mathtt{s}"] \arrow[d, "\mathtt{ls}"] & \arrow[d, phantom, "\swarrow"] & x[x/y\ z][y/z] \arrow[d, "\mathtt{ls}"] \\
(y\ y)[x/y\ y][y/z] \arrow[r, Rightarrow, "\mathtt{s}"] & (y\ z)[x/y\ y][y/z] \arrow[r, Rightarrow, "\mathtt{s}"] & (y\ z)[x/y\ z][y/z]
\end{tikzcd}$$

2) Apply Lemma 9.2 taking $\rightarrow_1:=\multimap_{\mathtt{ls}}$, $\rightarrow_2:=\Rightarrow_{\mathtt{s}}$ and since $\multimap_{\mathtt{ls}}$ is strongly normalizing (from $\multimap_{\mathtt{ls}}\subseteq\rightarrow_{\mathtt{ls}}$ and termination of $\rightarrow_{\mathtt{ls}}$) we get $(\multimap_{\mathtt{ls}} \cup \Rightarrow_{\mathtt{s}})^* \subseteq\multimap_{\mathtt{ls}}^*\Rightarrow_{\mathtt{s}}^*$, and conclude since $\rightarrow_{\mathtt{ls}}=\multimap_{\mathtt{ls}} \cup \Rightarrow_{\mathtt{s}}$. □

We know that a $\rightarrow_{\mathtt{h}}$ 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) $\rightarrow_{\mathtt{h}}$-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 11 *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 $t \Rightarrow_{\mathtt{s}} u$, see [2] for the technical details. The idea is that $\Rightarrow_{\mathtt{s}}$ cannot create, duplicate nor alter the head nature of $\multimap_{\mathtt{dB}}$ redexes, and so the $\multimap_{\mathtt{dB}}$-step can be preponed. Conversely, $\multimap_{\mathtt{dB}}$-steps cannot erase, duplicate nor alter the non-head nature of $\Rightarrow_{\mathtt{s}}$ redexes, therefore the two steps commute.

2. Let $t \Rightarrow_{\mathtt{s}}^{k}\multimap_{\mathtt{dB}} u$. The proof is by induction on k using point 1, and it is a standard diagram chasing.

□

The next lemma is the last brick for the projection.

Lemma 12 *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 8 we get $t{\downarrow}\multimap\to_{\mathtt{ls}}^{*} u$. By lemma 10 $t \to_{\mathtt{ls}}^{*} t{\downarrow}$ 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 11.2 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 13 (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 12 there exists r s.t. $v{\downarrow_{\circ}} \multimap_{\mathtt{dB}} r \to_{\mathtt{ls}}^{*} u$. Moreover, $v \multimap_{\mathtt{ls}}^{*} v{\downarrow_{\circ}}$; call γ this reduction. Let ρ be the reduction obtained by concatenating $\tau : t \multimap^{*} v$, $\gamma : v \multimap_{\mathtt{ls}}^{*} v{\downarrow_{\circ}}$ and $v{\downarrow_{\circ}} \multimap_{\mathtt{dB}} r$. We have $|\rho|_{\mathtt{dB}} = |\tau|_{\mathtt{dB}} + 1 = k$ and $r \to_{\mathtt{ls}}^{*} u$, and so we conclude. □

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 multiplicative 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 exponential steps in a linear head reduction ρ is a polynomial of the number of multiplicative 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 3 Hereditary head contexts *are generated by the following grammar:*

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

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

$$\begin{array}{lcl@{\qquad}lcll}
|x|_{HH} & = & 0 & |t[x/u]|_{HH} & = & |t|_{HH} & \textit{if } t \neq HH[x] \\
|\lambda x.t|_{HH} & = & |t|_{HH} & |t[x/u]|_{HH} & = & |t|_{HH}+1 & \textit{otherwise} \\
|t\ u|_{HH} & = & |t|_{HH} & & & &
\end{array}$$

Please notice that $|t|_{HH} = 0$ for any λ-term t. We need a lemma about HH-contexts.

Lemma 14 *Let t be a shallow term.*
1. *If $|t|_{HH} = 0$ and $t = HH[x]$ then the context $HH[\cdot]$ is an head context.*
2. *If t is shallow, $t \multimap_{\mathtt{ls}} u$, $t = HH[x]$ and $HH[\cdot]$ does not capture x, then there exists HH_0 s.t. $u = HH_0[x]$ and HH_0 does not capture x.*

Proof. 1. By induction on t. If $t = r[y/u]$ the hypothesis $|t|_{HH} = 0$ implies that $r \neq HH[y]$ and $|r|_{HH} = 0$. If $t = HH[x]$ for some x then $HH[\cdot] = HH_0[\cdot][y/u]$ for some hereditary head context HH_0, and $r = HH_0[x]$. By *i.h.* HH_0 is an head context, and so is HH. All other cases are straightforward.

2. By induction on $t \multimap_{\mathtt{ls}} u$. There are two interesting cases:
   - *Base case*: $t = H[y][y/r] \multimap_{\mathtt{ls}} H[r][y/r] = u$. Then $r = H_0[x]$ and $u = H[H_0[x]][y/r]$. We conclude by defining $HH_0 := H[H_0][y/r]$.
   - *Substitution inductive case*: $t = r[y/v] \multimap_{\mathtt{ls}} s[y/v] = u$ because $r \multimap_{\mathtt{ls}} s$. There are two subcases:
     - $t = HH[x][y/v]$ (and $x \neq y$). Then by *i.h.* exists HH'_0 s. t. $s = HH'_0[x]$ and so we conclude taking $HH_0 := HH'_0[y/v]$.
     - $t = HH[y][y/v]$ and $v = H[x]$. Then by *i.h.* exists HH'_0 s. t. $s = HH'_0[y]$ and so we conclude taking $HH_0 := HH'_0[y][y/H]$.

This concludes the proof. □

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.

Lemma 15 ($|\cdot|_{HH}$ 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. $\Rightarrow$) By induction on t. The only interesting case is if $t = u[x/r]$. Then u is a $\multimap_{\mathtt{ls}}$-normal form and so by *i.h.* we get $|u|_{HH} = 0$. By hypothesis $u \neq H[x]$, otherwise t would not be $\multimap_{\mathtt{ls}}$-normal. By Lemma 14.1 if $u = HH[y]$ then HH is an head context. Since for any term v there is exactly one head context H_0 s.t. $v = H_0[s]$ and s is a variable, we get that $HH = H$ and $u \neq HH[x]$. By definition of $|\cdot|_{HH}$ we get $|t|_{HH} = |u|_{HH} = 0$.
$\Leftarrow$) By induction on t. The only interesting case is if $t = u[x/r]$. The hypothesis implies that $|u|_{HH} = 0$ and $u \neq HH[x]$. In particular, $u \neq H[x]$ and $\cdot[x/r]$ does not give an $\multimap_{\mathtt{ls}}$-redex. By *i.h.* u is a $\multimap_{\mathtt{ls}}$-normal form. Then t is a $\multimap_{\mathtt{ls}}$-normal form.

2. By induction on $t \multimap_{\mathtt{ls}} u$. Cases:
   - $t = H[x][x/r] \multimap_{\mathtt{ls}} H[r][x/r] = u$. Since x is free in $H[x]$ we get that $H[x]$ is a $\multimap_{\mathtt{ls}}$-normal form, and by Point 1 $|H[x]|_{HH} = 0$. It follows that $|t|_{HH} = 1$. The hypothesis that t is shallow implies that r is a λ-term and the hypothesis $H[x][x/r] \multimap_{\mathtt{ls}} H[r][x/r]$ implies that H does not capture any free variable of r. Then $H[r]$ is a $\multimap_{\mathtt{ls}}$-normal form and by Point 1 $|H[r]|_{HH} = 0$. By lemma 14.1 if $H[r]$ has the form $HH[y]$ for some y then $HH[\cdot]$ is an head context, and so y is a variable of r. Since $x \notin \mathtt{fv}(r)$ we get $x \neq y$ and $H[r]$ has not the form $HH[x]$. Hence $|H[r][x/r]|_{HH} = |u|_{HH} = 0$, and $|t|_{HH} = |u|_{HH} + 1$.

- $t = \lambda x.r \multimap_{\mathtt{ls}} \lambda x.s = u$. Using the *i.h.*.
- $t = r\ v \multimap_{\mathtt{ls}} s\ v = u$. Using the *i.h.*.
- $t = r[x/v] \multimap_{\mathtt{ls}} s[x/v] = u$. By *i.h.* $|r|_{HH} = |s|_{HH} + 1$. The measures $|t|_{HH}$ and $|u|_{HH}$ are given by $|r|_{HH}$ and $|s|_{HH}$, respectively, plus the eventual contribution of $[x/v]$. By lemma 14.2 either $[x/v]$ contributes to both $|t|_{HH}$ and $|u|_{HH}$ or it contributes to none. In both cases we get $|t|_{HH} = |u|_{HH} + 1$.

3. By induction on t. The base case $t = x$ is trivial, and the cases $t = \lambda x.u$ and $t = u\ r$ follows from the *i.h.*. The only interesting case is when $t = u[x/r]$. If $|u|_{HH} > 0$ then we conclude using the *i.h.*. Otherwise, it must be that $u = HH[x]$. By Lemma 14.1 HH is an head context and so $t = H[x][x/r] \multimap_{\mathtt{ls}} H[r][x/r]$ and we conclude.

□

Summing up, we get:

Corollary 16 (exact bound to $\multimap_{\mathtt{ls}}$-sequences) $t \multimap_{\mathtt{ls}}^{n} t\downarrow_{\circ}$ *iff* $n = |t|_{HH}$.

Proof. By induction on n. For $n = 0$ the statement is given by Lemma 15.1. Then let $n > 0$.
Direction $\Rightarrow$: if $t \multimap_{\mathtt{ls}} u \multimap_{\mathtt{ls}}^{n-1} t\downarrow_{\circ}$ then by *i.h.* $|u|_{HH} = n - 1$. By Lemma 15.2 we get $|t|_{HH} = n$.
Direction $\Leftarrow$: by Lemma 15.3 t is not $\multimap_{\mathtt{ls}}$-normal and so $t \multimap_{\mathtt{ls}} u$. By Lemma 15.2 we get $|u|_{HH} = n - 1$ and by *i.h.* $u \multimap_{\mathtt{ls}}^{n-1} u\downarrow_{\circ} = t\downarrow_{\circ}$, and so $t \multimap_{\mathtt{ls}}^{n} t\downarrow_{\circ}$. □

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 (noted $|\rho|_{\mathtt{dB}}$).

Lemma 17 *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. It is easy to see that $\multimap_{\mathtt{ls}}$-steps do not change the number of substitutions in a term (*i.e.* $\mathtt{es}(s)$): by lemma 3 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 18 *Let $t \in \mathcal{T}_{\lambda}$. If $\rho : t \multimap^{n} u$ then $n = O(|\rho|_{\mathtt{dB}}^{2})$.*

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 16 $|\gamma_j| \leq |r_j|_{HH}$ and by Lemma 17 $|r_j|_{HH} \leq \sum_{i \in \{1,\ldots,j\}} |\tau_i|$. 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 $\sum_{i \in \{1,\ldots,k\}} |\gamma_i| \leq |\rho|_{\mathtt{dB}}^{2}$ and $n \leq |\rho|_{\mathtt{dB}}^{2} + |\rho|_{\mathtt{dB}} = O(|\rho|_{\mathtt{dB}}^{2})$. □

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

Corollary 19 (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 $\rightarrow_{\mathtt{h}}$-normal form r and diverges if u has no $\rightarrow_{\mathtt{h}}$-normal form. Moreover, the algorithm works in polynomial time on the derivation complexity of the input term.*

One may now wonder why a result like Corollary 19 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 strategy, one needs to perform *too many* substitutions not related to any β-redex. If one wants to generalize Corollary 19, 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 implementation of β-reduction in such a way that the overhead due to substitutions remains under control and is 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 19 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. This way, linear explicit substitutions are proved to be a succint, acceptable encoding of λ-terms in the sense of Papadimitriou [16]. 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 4 (Relative Unfoldings) *The unfolding $t{\downarrow}_C$ of t relative to context C is defined by induction on C:*

$$\begin{array}{lll} t{\downarrow}_{[\cdot]} := t{\downarrow}; & t{\downarrow}_{u\ C} := t{\downarrow}_C; & t{\downarrow}_{u[x/C]} := t{\downarrow}_C; \\ t{\downarrow}_{C\ u} := t{\downarrow}_C; & t{\downarrow}_{\lambda x.C} := t{\downarrow}_C; & t{\downarrow}_{C[x/u]} := t{\downarrow}_C\{x/u{\downarrow}\}. \end{array}$$

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

Definition 5 (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.*

The algorithm tests pairs (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 $x \notin \mathtt{fv}(c)$[1]. 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:

[1]Any term can be turned in this form in polynomial time, by $\rightarrow_{\mathtt{gc}}$-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

Definition 6 (Unfolding Judgments) *Let $P = (a, b)$ be a preprocessed pair. An unfolding judgement is a triple $(t, C), v, (u, D)$, where 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 induces 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 $R \overset{w}{\sim} S$.

Lemma 20 *The relation $\sqsubset^*_{a,b}$ is a partial order, while $\sqsubset^+_{a,b}$ is a strict order.*

Proof. Let $\lessdot$ the strict order on $(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$ defined as the product order of the standard lexicographic order on $\mathbb{N} \times \mathbb{N}$. Let $||C||$ be the number of substitutions into which $[\cdot]$ is embedded inside C. Observe that reflexivity and transitivity of $\sqsubset^*_{a,b}$ hold by definition. About antisimmetry, just observe that if $((t, C), (u, D)) \sqsubset_{a,b} ((r, E), (s, F))$, then $((||E||, |E|), (||F||, |F|)) \lessdot ((||C||, |C|), (||D||, |D|))$. $\square$

Lemma 21 *For every P for a and for every Q for b there is exactly one v such that $P \overset{v}{\sim} Q$.*

Proof. Let $P = (t, C)$ and $Q = (u, D)$. We proceed by induction on the relation $\sqsupset_{a,b}$, which is a strict order on a finite set, thus a well order. Let us distinguish some cases depending on the form of t and u:

- If both t and u are variables not in $\mathcal{S}$, then we can apply rule `var`.
- If both t and u are abstractions, then we can apply the inductive hypothesis to the pairs obtained by taking the bodies of these two abstractions and apply the inductive hypothesis. Depending on the (unique!) outcome, we can apply exactly one between λ_1, λ_2 and λ_3.
- If both t and u are applications, then we can apply the inductive hypothesis to the pairs obtained by taking the immediate subterm, and conclude observing the shape of rule @.

- If any of t and u is a substitution, then we can proceed as usual and then apply either rule $\mathtt{sub}_l$ or $\mathtt{sub}_r$.
- If any of t and u is a substituted variable, then we can again apply the inductive hypothesis, and the rule $\mathtt{unf}_l$ or $\mathtt{unf}_r$.
- In all the other cases, we can easily conclude by observing the shape of the six rules $\mathtt{err}_{\lambda @}$, $\mathtt{err}_{@\lambda}$. $\mathtt{err}_{\lambda x}$, $\mathtt{err}_{x\lambda}$, $\mathtt{err}_{x@}$, $\mathtt{err}_{@x}$.

This concludes the proof. □

Lemma 22 *If P and Q are pairs for a and b (respectively) and $P \overset{v}{\sim} Q$, then $|v| \leq |a| \cdot |b|$.*

Proof. By induction on the structure of the proof that $P \overset{A}{\sim} Q$, it is easy to prove that if $(x, y) \in A$, then x is free in a and y is free in b. □

Definition 7 (Unfolding Matrix) *Let $P = (a, b)$ be a preprocessed pair. An unfolding matrix $\mathtt{M}$ for P is bidimensional array with the following form:*
1. *its rows are indexed by the pairs $(t, C[\cdot])$ such that $C[t] = a$;*
2. *its columns are indexed by the pairs $(t, C[\cdot])$ such that $C[t] = b$;*
3. *the values of the matrix are either (possibly empty) constraining sets or $\perp$ 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 8 (Unfolding Checking Algorithm) *Let $P = (a, b)$ be a preprocessed pair. We define the following algorithm, which which be referred to as the* unfolding checking algorithm*:*

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*.*

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

Lemma 23 *Relative unfoldings verify:*
1. *If C does not contain an explicit substitution for x then $x{\downarrow_C} = x$.*
2. *$t[x/r]{\downarrow_C} = t{\downarrow_{C[[\cdot][x/r]]}}$.*
3. *$(\lambda x.t){\downarrow_C} = \lambda x.(t{\downarrow_{C[\lambda x.[\cdot]]}})$.*
4. *$(t\ r){\downarrow_C} = t{\downarrow_{C[[\cdot]\ r]}}\ r{\downarrow_{C[t\ [\cdot]]}}$.*
5. *$t{\downarrow_{C[E[x][x/[\cdot]]]}} = x{\downarrow_{C[E[x/t]]}}$.*

Proof. All points are by induction on C. Relevant cases:
- If $C = [\cdot]$, then:

$$
\begin{aligned}
&1. & x{\downarrow_{[\cdot]}} &= x;\\
&2. & t{\downarrow_{[\cdot][x/r]}} &= t{\downarrow_{[\cdot]}}\{x/r{\downarrow}\} = t{\downarrow}\{x/r{\downarrow}\} = (t[x/r]){\downarrow};\\
&3. & (\lambda x.t){\downarrow_{[\cdot]}} &= (\lambda x.t){\downarrow} = \lambda x.t{\downarrow} = \lambda x.(t{\downarrow_{[\cdot]}}) = \lambda x.(t{\downarrow_{\lambda x.[\cdot]}});\\
&4. & (t\ r){\downarrow_{[\cdot]}} &= (t\ r){\downarrow} = t{\downarrow_{[\cdot]}}\ r{\downarrow_{[\cdot]}} = t{\downarrow_{([\cdot]\ r)}}\ r{\downarrow_{(t\ [\cdot])}}\\
&5. & t{\downarrow_{E[x][x/[\cdot]]}} &= t;\\
& & x{\downarrow_{E[x/t]}} &= x{\downarrow_E}\{x/t\} =_{p.1} x\{x/t\} = t.
\end{aligned}
$$

- If $C = Du$, then:

  1. $x{\downarrow}_{(D\ u)} = x{\downarrow}_{D} =_{i.h.} x;$
  2. $t{\downarrow}_{D[[\cdot][x/r]]\ u} = t{\downarrow}_{D[[\cdot][x/r]]} =_{i.h.} t[x/r]{\downarrow}_{D} = t[x/r]{\downarrow}_{(D\ u)};$
  3. $(\lambda x.t){\downarrow}_{(D\ u)} = (\lambda x.t){\downarrow}_{D} =_{i.h.} \lambda x.(t{\downarrow}_{D[\lambda x.[\cdot]]}) = \lambda x.(t{\downarrow}_{D[\lambda x.[\cdot]]\ u});$
  4. $$\begin{aligned}(t\ r){\downarrow}_{(D\ u)} &= (t\ r){\downarrow}_{D} =_{i.h.} t{\downarrow}_{D[[\cdot]\ r]}\ r{\downarrow}_{D[t\ [\cdot]]}\\ &= t{\downarrow}_{(D[[\cdot]\ r]\ u)}\ r{\downarrow}_{(D[t\ [\cdot]]\ u)};\end{aligned}$$
  5. $$\begin{aligned}t{\downarrow}_{(D[E[x][x/[\cdot]]]\ u)} &= t{\downarrow}_{D[E[x][x/[\cdot]]]} =_{i.h.} x{\downarrow}_{D[E[x/t]]}\\ &= x{\downarrow}_{(D[E[x/t]]\ u)}.\end{aligned}$$

- If $C = D[y/u]$, then:

  1. $x{\downarrow}_{D[y/u]} = x{\downarrow}_{D}\{y/u{\downarrow}\} =_{i.h.} x\{y/u{\downarrow}\} = x;$
  2. $$\begin{aligned}t[x/r]{\downarrow}_{D[y/u]} &= t[x/r]{\downarrow}_{D}\{y/u{\downarrow}\}\\ &=_{i.h.} t{\downarrow}_{D[[\cdot][x/r]]}\{y/u{\downarrow}\} = t{\downarrow}_{D[[\cdot][x/r]][y/u]}.\end{aligned}$$
  3. $$\begin{aligned}(\lambda x.t){\downarrow}_{D[y/u]} &= (\lambda x.t){\downarrow}_{D}\{y/u{\downarrow}\} =_{i.h.} \lambda x.(t{\downarrow}_{D[\lambda x.[\cdot]]})\{y/u{\downarrow}\}\\ &= \lambda x.(t{\downarrow}_{D[\lambda x.[\cdot]]}\{y/u{\downarrow}\}) = \lambda x.(t{\downarrow}_{D[\lambda x.[\cdot]][y/u]}).\end{aligned}$$
  4. $$\begin{aligned}(t\ r){\downarrow}_{D[y/u]} &= (t\ r){\downarrow}_{D}\{y/u{\downarrow}\} =_{i.h.} (t{\downarrow}_{D[[\cdot]\ r]}\ r{\downarrow}_{D[t\ [\cdot]]})\{y/u{\downarrow}\}\\ &= (t{\downarrow}_{D[[\cdot]\ r]}\{y/u{\downarrow}\})\ (r{\downarrow}_{D[t\ [\cdot]]}\{y/u{\downarrow}\})\\ &= t{\downarrow}_{D[[\cdot]\ r][y/u]}\ r{\downarrow}_{D[t\ [\cdot]][y/u]};\end{aligned}$$
  5. $$\begin{aligned}t[x/r]{\downarrow}_{D[y/u]} &= t[x/r]{\downarrow}_{D}\{y/u{\downarrow}\}\\ &=_{i.h.} t{\downarrow}_{D[[\cdot][x/r]]}\{y/u{\downarrow}\} = t{\downarrow}_{D[[\cdot][x/r]][y/u]}.\end{aligned}$$

This concludes the proof. □

Lemma 24 *If $(t, C) \overset{v}{\sim} (u, D)$ and $v \neq \bot$ then $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C})$ and $\pi_2(v) = \mathtt{fv}(u{\downarrow}_{D})$, where π_1 and π_2 are the usual projections functions.*

Proof. By induction on the derivation $(t, C) \overset{v}{\sim} (u, D)$. The rule `var` is the only atomic case, and the statement trivially holds. The inductive cases:

- $\mathtt{unf}_l$) By *i.h.* $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C[E[x][x/[\cdot]]]})$. We have to prove that $\pi_1(v) = \mathtt{fv}(x{\downarrow}_{C[E[x/t]]})$. Lemma 23.5 gives $t{\downarrow}_{C[E[x][x/[\cdot]]]} = x{\downarrow}_{C[E[x/t]]}$, and so we conclude.
- $\mathtt{unf}_r$) As the previous case.
- $\mathtt{sub}_l$) By *i.h.* $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C[[\cdot][x/r]]})$. We have to prove that $\pi_1(v) = \mathtt{fv}(t[x/r]{\downarrow}_{C})$. By lemma 23.2 $t{\downarrow}_{C[[\cdot][x/r]]} = t[x/r]{\downarrow}_{C}$, and so we conclude.
- $\mathtt{sub}_r$) As the previous case.
- λ_1) By *i.h.* $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C[\lambda x.[\cdot]]})$. We have to prove that $\pi_1(v)\backslash\{x\} = \mathtt{fv}((\lambda x.t){\downarrow}_{C})$. By Lemma 23.3 $\lambda x.(t{\downarrow}_{C[\lambda x.[\cdot]]}) = (\lambda x.t){\downarrow}_{C}$, and $\mathtt{fv}(\lambda x.(t{\downarrow}_{C[\lambda x.[\cdot]]})) = \mathtt{fv}(t{\downarrow}_{C[\lambda x.[\cdot]]}) \setminus \{x\} = \pi_1(v) \setminus \{x\}$. Similarly for $(u, D[\lambda y.[\cdot]])$.
- λ_2) By *i.h.* $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C[\lambda x.[\cdot]]})$. The reasoning is as in the previous point, except that the hypothesis $(x, z) \notin v$ for all z implies that $x \notin \pi_1(v)$ and so $\pi_1(v) \setminus \{x\} = \pi_1(v)$. Similarly for $(u, D[\lambda y.[\cdot]])$.
- @) By hypothesis $v \diamond w \neq \bot$, which happens only if $v \neq \bot \neq w$ and $v \sim w$. In that case $v \diamond w = v \cup w$ and so $\pi_1(v \diamond w) = \pi_1(v) \cup \pi_1(w)$. Since $v \neq \bot \neq w$ we can apply the *i.h.* to the hypothesis of the rule, and get $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C[[\cdot]\ r]})$ and $\pi_1(w) = r{\downarrow}_{C[t\ [\cdot]]}$. We have to prove

that $\pi_1(v) \cup \pi_1(w) = \mathtt{fv}(t\ r{\downarrow_C})$. By Lemma 23.4 we get $t\ r{\downarrow_C} = t{\downarrow_{C[[\cdot]\ r]}}\ r{\downarrow_{C[t\ [\cdot]]}}$, and so we conclude. Similarly for $(u\ s, D)$.

- For other rules $v = \bot$, so there is nothing to prove.

$\square$

Definition 9 (Unifying Renaming) *A constraining set $v = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ is a* unifying renaming *for two λ-terms t and u if v is a bijection of $\mathtt{fv}(t)$ and $\mathtt{fv}(u)$ such that:*
1. $u\{x_1/y_1\}\ldots\{x_k/y_k\} = t$.
2. $t\{y_1/x_1\}\ldots\{y_k/x_k\} = u$.

Lemma 25 *If there is a unifying renaming between t and u then it is unique.*

Proof. By induction on t. Straightforward. $\square$

Lemma 26 *If $(t, C) \overset{v}{\sim} (u, D)$ and $v \neq \bot$ then v is auto-coherent,* i.e. *it is the graph of a bijection.*

Proof. By induction on the derivation $(t, C) \overset{v}{\sim} (u, D)$. For rule $\mathtt{var}$ it is obvious. For rules $\mathtt{unf}_l$, $\mathtt{unf}_r$, $\mathtt{sub}_l$, $\mathtt{sub}_r$, λ_1 and λ_2 it simply follows from the *i.h.*. For rule @ the *i.h.* gives that v and w are two bijections. Now, $v \diamond w \neq \bot$ implies that $v \neq \bot \neq w$ and $v \sim w$. Suppose that $v \diamond w = v \cup w$ is not a bijection. This happens iff $v \diamond w$ is not a function or it is not injective, but since v and w are bijections both cases are absurd because of $v \sim w$. $\square$

Lemma 27 $(t, C) \overset{v}{\sim} (u, D)$ *and v auto-coherent implies v is a unifying renaming for $t{\downarrow_C}$ and $u{\downarrow_D}$.*

Proof. By induction on the derivation of $(t, C) \overset{A}{\sim} (u, D)$, using lemma 23. Cases:

- $\mathtt{var}$) Immediate.
- $\mathtt{sub}_l$) By *i.h.* A is a unifying renaming for $t{\downarrow_{C[[\cdot][x/r]]}}$ and $u{\downarrow_D}$. By lemma 2 we get:

$$t[x/r]{\downarrow_C} = t{\downarrow_{C[[\cdot][x/r]]}}$$

  Thus A is an unifying renaming for $t[x/r]{\downarrow_C}$ and $u{\downarrow_D}$.
- $\mathtt{sub}_r$) As the previous one.
- λ_1) By *i.h.* $A \cup (x, y)$ is a unifying renaming for $t{\downarrow_{C[\lambda x.[\cdot]]}}$ and $u{\downarrow_{D[\lambda y.[\cdot]]}}$ and thus by definition $x \in \mathtt{fv}(t{\downarrow_{C[\lambda x.[\cdot]]}})$ and $y \in \mathtt{fv}(u{\downarrow_{D[\lambda y.[\cdot]]}})$. By lemma 23.2 we get:

$$(\lambda x.t){\downarrow_C} = \lambda x.(t{\downarrow_{C[\lambda x.[\cdot]]}})$$

  And similarly $(\lambda y.u){\downarrow_D} = \lambda y.(u{\downarrow_{D[\lambda y.[\cdot]]}})$. Since $\mathtt{fv}(\lambda x.(t{\downarrow_{C[\lambda x.[\cdot]]}})) = \mathtt{fv}(t{\downarrow_{C[\lambda x.[\cdot]]}}) \setminus \{x\}$ and $\mathtt{fv}(\lambda y.(u{\downarrow_{D[\lambda y.[\cdot]]}})) = \mathtt{fv}(u{\downarrow_{D[\lambda y.[\cdot]]}}) \setminus \{y\}$ we get that A is a unifying renaming for $(\lambda x.t){\downarrow_C}$ and $(\lambda y.u){\downarrow_D}$.
- λ_2) By *i.h.* A is a unifying renaming for $t{\downarrow_{C[\lambda x.[\cdot]]}}$ and $u{\downarrow_{D[\lambda y.[\cdot]]}}$ s.t. $(x, z), (z, y) \notin A, \forall z$. As in the previous case $(\lambda x.t){\downarrow_C} = \lambda x.(t{\downarrow_{C[\lambda x.[\cdot]]}})$ and $(\lambda y.u){\downarrow_D} = \lambda y.(u{\downarrow_{D[\lambda y.[\cdot]]}})$. By definition of unifying renaming we get that $x \notin \mathtt{fv}(t{\downarrow_{C[\lambda x.[\cdot]]}})$ and $y \notin \mathtt{fv}(u{\downarrow_{D[\lambda y.[\cdot]]}})$, and since $\mathtt{fv}(t{\downarrow_{C[\lambda x.[\cdot]]}}) = \mathtt{fv}(\lambda x.(t{\downarrow_{C[\lambda x.[\cdot]]}}))$ and $\mathtt{fv}(u{\downarrow_{D[\lambda y.[\cdot]]}}) = \mathtt{fv}(\lambda y.(u{\downarrow_{D[\lambda y.[\cdot]]}}))$ we get that A is a unifying renaming of $(\lambda x.t){\downarrow_C}$ and $(\lambda y.u){\downarrow_D}$.
- @) By *i.h.* A is a unifying renaming for $t{\downarrow_{C[[\cdot]\ r]}}$ and $u{\downarrow_{D[[\cdot]\ s]}}$, and B is a unifying renaming for $r{\downarrow_{C[t\ [\cdot]]}}$ and $s{\downarrow_{D[u\ [\cdot]]}}$. By lemma 23.4 we get

$$(t\ r){\downarrow_C} = t{\downarrow_{C[[\cdot]\ r]}}\ r{\downarrow_{C[t\ [\cdot]]}}$$

  and

$$(u\ s){\downarrow_D} = u{\downarrow_{D[[\cdot]\ s]}}\ s{\downarrow_{D[u\ [\cdot]]}}$$

  Since $\mathtt{fv}((t\ r){\downarrow_C}) = \mathtt{fv}(t{\downarrow_{C[[\cdot]\ r]}}) \cup \mathtt{fv}(r{\downarrow_{C[t\ [\cdot]]}})$ (and analogously for $(u\ s){\downarrow_D}$) and $A \sim B$, we get that $A \cup B$ is a unifying renaming for $(t\ r){\downarrow_C}$ and $(u\ s){\downarrow_D}$.

- $\mathtt{unf}_l$) By *i.h.* A is a unifying renaming for $t{\downarrow}_{C[E[x][x/[\cdot]]]}$ and $u{\downarrow}_D$. By lemma 23.5 we get
$$t{\downarrow}_{C[E[x][x/[\cdot]]]} = x{\downarrow}_{C[E[x/t]]}$$
And so A is a unifying renaming for $x{\downarrow}_{C[E[x/t]]}$ and $u{\downarrow}_D$.
- $\mathtt{unf}_r$) As in the previous case.

This concludes the proof. □

Lemma 28 *$(t, C) \overset{\perp}{\sim} (u, D)$ implies there is no unifying renaming for $t{\downarrow}_C$ and $u{\downarrow}_D$.*

Proof. The relative unfolding of a lambda is always a lambda, and the relative unfolding of an application is always an application. Therefore for rules $\mathtt{err}_{\lambda @}$ and $\mathtt{err}_{@\lambda}$ the relative unfoldings have different topmost constructors and thus are different. For rules $\mathtt{err}_{\lambda x}$, $\mathtt{err}_{x\lambda}$, $\mathtt{err}_{x@}$ and $\mathtt{err}_{@x}$ note that Lemma 23.1 states that the relative unfolding of a non-substituted variable is a variable, thus again the relative unfoldings of the two terms have different topmost constructors and are different. For rule λ_3 if $v = \perp$ then it follows from the *i.h.* and Lemma 23.3.

Otherwise suppose that there exists $z \neq y$ s.t. $\{(x, z)\} \in v$. By Lemmas 26 and 27 we get that v is an unifying renaming for $t{\downarrow}_{C[\lambda x.[\cdot]]}$ and $u{\downarrow}_{D[\lambda y.[\cdot]]}$. Moreover, by Lemma 24 we get $\pi_1(v) = \mathtt{fv}(t{\downarrow}_{C[\lambda x.[\cdot]]})$ and $\pi_2(v) = \mathtt{fv}(u{\downarrow}_{D[\lambda y.[\cdot]]})$.

Now, by Lemma 23.3 we get $(\lambda x.t){\downarrow}_C = \lambda x.(t{\downarrow}_{C[\lambda x.[\cdot]]})$ and $(\lambda y.u){\downarrow}_D = \lambda y.(u{\downarrow}_{D[\lambda y.[\cdot]]})$. From $\{(x, z)\} \in v$ we get that $x \in \mathtt{fv}(t{\downarrow}_{C[\lambda x.[\cdot]]})$. It is easily seen that in this case if there is an unifying renaming for $(\lambda x.t){\downarrow}_C$ and $(\lambda y.u){\downarrow}_D$ it is necessarily obtained from an unifying renaming for $u{\downarrow}_{D[\lambda y.[\cdot]]}$ and $u{\downarrow}_{D[\lambda y.[\cdot]]}$ by removing the pair (x, y). We know that v is an unifying renaming for $u{\downarrow}_{D[\lambda y.[\cdot]]}$ and $u{\downarrow}_{D[\lambda y.[\cdot]]}$. By Lemma 26 it is a bijection. By hypothesis it contains the pair (x, z), and so it cannot contain the pair (x, y). So there cannot be an unifying renaming for $(\lambda x.t){\downarrow}_C$ and $(\lambda y.u){\downarrow}_D$. □

Theorem 29 (Correctness) *The Unfolding Checking Algorithm, on input (a, b), returns* `yes` *iff $a{\downarrow} = b{\downarrow}$*

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

Proposition 30 (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 due to Lemma 22. As a consequence, Step 4 can be performed in polynomial time.

□

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 the same unfolding 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 derivational complexity 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 first thing we need to encode is a form of recursion. We denote by H the term tt, where $t \equiv \lambda x.\lambda y.y(xxy)$. H is a call-by-name fixed-point operator: for every term u,

$$Hu \rightarrow_{\mathtt{h}} (\lambda y.y(tty))u \rightarrow_{\mathtt{h}} u(Hu).$$

The λ-term H provides the necessary computational expressive power to encode the whole class of computable functions.

The simplest objects we need to encode in the λ-calculus are finite sets. Elements of any finite set $A = \{a_1, \ldots, a_n\}$ can be encoded as follows: $\lceil a_i \rceil^A \equiv \lambda x_1. \ldots .\lambda x_n.x_i$. 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 Scott. Let $\Sigma = \{a_1, \ldots, a_n\}$ 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:

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

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.

Lemma 31 *Given a finite alphabet Σ, there are a term $AC(\Sigma)$ and a constant $n_{AC}^{\Sigma} \in \mathbb{N}$ such that for every $a \in \Sigma$, every term t and every $s \in \Sigma^*$, there is $n \leq n_{AC}^{\Sigma}$ such that $AC(\Sigma)t\lceil a \rceil^{\Sigma}\lceil u \rceil^{\Sigma^*} \rightarrow_{\mathtt{h}}^{n} t\lceil au \rceil^{\Sigma^*}$.*

Proof. The term we are looking for is defined as follows:

$$AC(\Sigma) \equiv \lambda y.\lambda a.\lambda u.aM_1 \ldots M_{|\Sigma|}uy$$

where for any i, $M_i \equiv \lambda u.\lambda y.y(\lambda x_1. \ldots .\lambda x_{|\Sigma|}.\lambda w.x_i u)$. Observe that:

$$\begin{aligned} AC(\Sigma)t\lceil a_i \rceil^{\Sigma}\lceil u \rceil^{\Sigma^*} &\rightarrow_{\mathtt{h}}^{3} \lceil a_i \rceil^{\Sigma} M_1 \ldots M_{|\Sigma|}\lceil u \rceil^{\Sigma^*} t \\ &\rightarrow_{\mathtt{h}}^{|\Sigma|} M_i \lceil u \rceil^{\Sigma^*} t \rightarrow_{\mathtt{h}}^{2} t\lceil a_i u \rceil^{\Sigma^*}. \end{aligned}$$

In other words, n_{AC}^{Σ} can be set to be $|\Sigma| + 5$. This concludes the proof. □

Given alphabets Σ and Δ, the function $\mathcal{G}_{\Delta}^{\Sigma} : \Sigma^* \rightarrow \Delta^*$ is defined by stipulating that for every $n \in \mathbb{N}$ and every $a_1, \ldots, a_n \in \Sigma$, $\mathcal{G}_{\Delta}^{\Sigma}(a_1 \ldots a_n) = u_1 \ldots u_n$. where u_i is a_i if $a_i \in \Delta$ and u_i is ε otherwise

Lemma 32 *Given finite alphabets Σ and Δ, there are a term $CS(\Sigma, \Delta)$ and a linear function $f_{CS}^{\Sigma,\Delta} : \mathbb{N} \rightarrow \mathbb{N}$ such that for every $u \in \Sigma$ there is $n \leq f_{CS}^{\Sigma,\Delta}(|u|)$ such that $CS(\Sigma, \Delta)t\lceil u \rceil^{\Sigma^*} \rightarrow_{\mathtt{h}}^{n} t\lceil \mathcal{G}_{\Delta}^{\Sigma}(u) \rceil^{\Delta^*}$.*

Proof. The term we are looking for is defined as follows:

$$CS(\Sigma,\Delta) \equiv H(\lambda x.\lambda z.\lambda u.uN_1 \ldots N_{|\Sigma|}Nz),$$

where for any i,

$$N_i \equiv \left\{ \begin{array}{ll} \lambda u.\lambda z.x(\lambda u.AC(\Sigma)z\lceil a_i\rceil^{\Delta}u)u & \text{if } a_i \in \Delta \\ \lambda u.\lambda z.x(\lambda u.zu)u & \text{otherwise.} \end{array} \right.$$

and $N \equiv \lambda z.z\lceil\varepsilon\rceil^{\Delta^*}$. Let P_i be $N_i\{CS(\Sigma,\Delta)/x\}$. Then the thesis can be proved by induction on u as soon as $f_{CS}^{\Sigma,\Delta}$ is defined as $f_{CS}^{\Sigma,\Delta}(x) = x(|\Sigma| + n_{AC}^{\Sigma} + 7) + |\Sigma| + 5$:

$$\begin{array}{rll} CS(\Sigma,\Delta)t\lceil\varepsilon\rceil^{\Sigma^*} & \rightarrow_{\mathtt{h}}^{2} & (\lambda z.\lambda u.uP_1 \ldots P_{|\Sigma|}Nz)t\lceil\varepsilon\rceil^{\Sigma^*} \\ & \rightarrow_{\mathtt{h}}^{2} & \lceil\varepsilon\rceil^{\Sigma^*}P_1 \ldots P_{|\Sigma|}Nt \\ & \rightarrow_{\mathtt{h}}^{|\Sigma|+1} & Nt \rightarrow_{\mathtt{h}} t\lceil\varepsilon\rceil^{\Delta^*} \\ CS(\Sigma,\Delta)\lceil a_iu\rceil^{\Sigma^*} & \rightarrow_{\mathtt{h}}^{2} & (\lambda z.\lambda u.uP_1 \ldots P_{|\Sigma|}Nz)t\lceil a_iu\rceil^{\Sigma^*} \\ & \rightarrow_{\mathtt{h}}^{2} & \lceil a_iu\rceil^{\Sigma^*}P_1 \ldots P_{|\Sigma|}Nt \\ & \rightarrow_{\mathtt{h}}^{|\Sigma|+1} & P_i\lceil u\rceil^{\Sigma^*}t \end{array}$$

Now, if $a_i \in \Delta$, then there are $n \leq f_{CS}^{\Sigma,\Delta}(|u|)$ and $m \leq n_{AC}^{\Sigma}$ such that

$$\begin{array}{rll} P_i\lceil u\rceil^{\Sigma^*}t & \rightarrow_{\mathtt{h}}^{2} & CS(\Sigma,\Delta)(\lambda u.AC(\Delta)t\lceil a_i\rceil^{\Delta}u)\lceil u\rceil^{\Sigma^*} \\ & \rightarrow_{\mathtt{h}}^{n} & (\lambda u.AC(\Delta)t\lceil a_i\rceil^{\Delta}u)\lceil \mathcal{G}_{\Delta}^{\Sigma}(u)\rceil^{\Delta^*} \\ & \rightarrow_{\mathtt{h}} & AC(\Delta)t\lceil a_i\rceil^{\Delta}\lceil \mathcal{G}_{\Delta}^{\Sigma}(u)\rceil^{\Delta^*} \\ & \rightarrow_{\mathtt{h}}^{m} & t\lceil a_i\mathcal{G}_{\Delta}^{\Sigma}(u)\rceil^{\Delta^*} \end{array}$$

while otherwise there is $n \leq f_{CS}^{\Sigma,\Delta}(|u|)$ such that

$$\begin{array}{rll} P_i\lceil u\rceil^{\Sigma^*}t & \rightarrow_{\mathtt{h}}^{2} & CS(\Sigma,\Delta)(\lambda u.tu)\lceil u\rceil^{\Sigma^*} \\ & \rightarrow_{\mathtt{h}}^{n} & (\lambda u.tu)\lceil \mathcal{G}_{\Delta}^{\Sigma}(u)\rceil^{\Delta^*} \\ & \rightarrow_{\mathtt{h}} & t\lceil \mathcal{G}_{\Delta}^{\Sigma}(u)\rceil^{\Delta^*} \end{array}$$

This concludes the proof. □

A deterministic Turing machine $\mathcal{M}$ is a tuple $(\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ consisting of:

- A finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$;
- A distinguished symbol $a_{blank} \in \Sigma$, called the *blank symbol*;
- A finite set $Q = \{q_1, \ldots, q_m\}$ of *states*;
- A distinguished state $q_{initial} \in Q$, called the *initial state*;
- A distinguished state $q_{final} \in Q$, called the *final state*;
- A partial *transition function* $\delta : Q \times \Sigma \rightharpoonup Q \times \Sigma \times \{\leftarrow, \rightarrow, \downarrow\}$ such that $\delta(q_i, a_j)$ is defined iff $q_i \neq q_{final}$.

A configuration for $\mathcal{M}$ is a quadruple in $\Sigma^* \times \Sigma \times \Sigma^* \times Q$. For example, if $\delta(q_i, a_j) = (q_l, a_k, \leftarrow)$, then $\mathcal{M}$ evolves from (ua_p, a_j, v, q_i) to (u, a_p, a_kv, q_l) (and from $(\varepsilon, a_j, v, q_i)$ to $(\varepsilon, a_{blank}, a_kv, q_l)$). A configuration like (u, a_i, v, q_{final}) is *final* and cannot evolve. Given a string $u \in \Sigma^*$ which does not contain any occurrence of a_{blank}, the *initial configuration* for u is $(\varepsilon, a_{blank}, u, q_{initial})$, while the *final* for u is $(\varepsilon, a_{blank}, u, q_{final})$.

A Turing machine $(\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ computes the function $f : \Delta^* \rightarrow \Delta^*$ (where $\Delta \subseteq \Sigma$ and a_{blank} is not in Δ) in time $g : \mathbb{N} \rightarrow \mathbb{N}$ iff for every $u \in \Delta^*$, the initial configuration for u evolves to a final configuration for $f(u)$ in $g(|u|)$ steps.

A configuration (s, a, v, q) of a Turing machine $\mathcal{M} = (\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ 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 $AC(\Sigma)$.

The following three lemmas formalize the above intuitive argument:

Lemma 33 *Given a Turing machine $\mathcal{M} = (\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ and a blank-free alphabet Δ, there are a term $\mathcal{I}(\mathcal{M},\Delta)$ and a linear function $f_{\mathcal{I}}^{\mathcal{M},\Delta}$ such that for every $u \in \Delta^*$, there in $n \in \mathbb{N}$ such that $\mathcal{I}(\mathcal{M},\Delta)t\lceil u\rceil^{\Delta^*} \rightarrow_{\mathtt{h}}^{n} t\lceil C\rceil^{\mathcal{M}}$ where C is the initial configuration for u and $n \leq f_{\mathcal{I}}^{\mathcal{M},\Delta}(|u|)$.*

Proof. Simply,

$$\begin{aligned}
\mathcal{I}(\mathcal{M},\Delta) &= \lambda x.\lambda u.CS(\Delta,\Sigma)(\lambda z.x(\lambda y.y\lceil\varepsilon\rceil^{\Sigma^*}\ \lceil a_{blank}\rceil^{\Sigma}\ z\ \lceil q\rceil^{Q}))u;\\
f_{\mathcal{I}}^{\mathcal{M},\Delta}(x) &= f_{CS}^{\Sigma,\Delta}(x) + 3.
\end{aligned}$$

Indeed,

$$\begin{aligned}
\mathcal{I}(\mathcal{M},\Delta)t\lceil u\rceil^{\Delta^*} &\rightarrow_{\mathtt{h}}^{2} CS(\Delta,\Sigma)(\lambda z.t(\lambda y.y\lceil\varepsilon\rceil^{\Sigma^*}\ \lceil a_{blank}\rceil^{\Sigma}\ z\ \lceil q\rceil^{Q}))\lceil u\rceil^{\Delta^*}\\
&\rightarrow_{\mathtt{h}}^{n} (\lambda z.t(\lambda y.y\lceil\varepsilon\rceil^{\Sigma^*}\ \lceil a_{blank}\rceil^{\Sigma}\ z\ \lceil q\rceil^{Q}))\lceil u\rceil^{\Sigma^*}\\
&\rightarrow_{\mathtt{h}} t(\lambda y.y\lceil\varepsilon\rceil^{\Sigma^*}\ \lceil a_{blank}\rceil^{\Sigma}\ \lceil u\rceil^{\Sigma^*}\ \lceil q\rceil^{Q}))\lceil u\rceil^{\Sigma^*}
\end{aligned}$$

where $n \leq f_{CS}^{\Delta,\Sigma}|u|$. This concludes the proof. □

Lemma 34 *Given a Turing machine $\mathcal{M} = (\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$ and for every alphabet Δ, there are a term $\mathcal{F}(\mathcal{M},\Delta)$ and a linear function $f_{\mathcal{F}}^{\mathcal{M},\Delta}$ such that for every final configuration C for $u \in \Delta^*$ there is $n \in \mathbb{N}$ $\mathcal{F}(\mathcal{M},\Delta)t\lceil C\rceil^{\mathcal{M}} \rightarrow_{\mathtt{h}}^{n} t\lceil u\rceil^{\Delta^*}$, where $n \leq f_{\mathcal{F}}^{\mathcal{M},\Delta}(|u|)$.*

Proof. Simply,

$$\begin{aligned}
\mathcal{F}(\mathcal{M},\Delta) &= \lambda x.\lambda y.y(\lambda v.\lambda a.\lambda u.\lambda q.CS(\Sigma,\Delta)xu)\\
f_{\mathcal{F}}^{\mathcal{M},\Delta}(x) &= f_{CS}^{\Sigma,\Delta}(x) + 7
\end{aligned}$$

Indeed,

$$\begin{aligned}
\mathcal{F}(\mathcal{M},\Delta)t\lceil C\rceil^{\mathcal{M}} &\rightarrow_{\mathtt{h}}^{2} \lceil C\rceil^{\mathcal{M}}(\lambda v.\lambda a.\lambda u.\lambda q.CS(\Sigma,\Delta)tu) \qquad \rightarrow_{\mathtt{h}}^{5} CS(\Sigma,\Delta)t\lceil u\rceil^{\Sigma^*}\\
&\rightarrow_{\mathtt{h}}^{n} t\lceil u\rceil^{\Delta^*}
\end{aligned}$$

where $n \leq f_{CS}^{\Sigma,\Delta}|u|$. This concludes the proof. □

Lemma 35 *Given a Turing machine $\mathcal{M} = (\Sigma, a_{blank}, Q, q_{initial}, q_{final}, \delta)$, there are a term $\mathcal{T}(\mathcal{M})$ and a linear function $f_{\mathcal{T}}^{\mathcal{M}}$ such that for every configuration C,*

- *if D is a final configuration reachable from C in n steps, then $\mathcal{T}(\mathcal{M})t\lceil C\rceil^{\mathcal{M}} \rightarrow_{\mathtt{h}}^{m} t\lceil D\rceil^{\mathcal{M}}$ where $m \leq f_{\mathcal{T}}^{\mathcal{M}}(n)$;*
- *the term $\mathcal{T}(\mathcal{M})t\lceil C\rceil^{\mathcal{M}}$ diverges if there is no final configuration reachable from C.*

Proof. $\mathcal{T}(\mathcal{M})$ is defined as

$$H(\lambda x.\lambda z.\lambda y.y(\lambda u.\lambda a.\lambda v.\lambda q.q(M_1 \ldots M_{|Q|})uavz)),$$

where, for any i and j:

$$\begin{array}{rcl}
M_i & \equiv & \lambda u.\lambda a.\lambda v.\lambda z.a(N_i^1 \ldots N_i^{|\Sigma|})uvz; \\
N_i^j & \equiv & \left\{\begin{array}{lr}
\lambda u.\lambda v.\lambda z.z(\lambda x.xu\lceil a_j\rceil^{\Sigma}v\lceil q_i\rceil^{Q}) & \text{if } q_i = q_{\mathit{final}} \\
\lambda u.\lambda v.\lambda z.xz(\lambda z.zu\lceil a_k\rceil^{\Sigma}v\lceil q_l\rceil^{Q}) & \text{if } \delta(q_i,a_j) = (q_l,a_k,\downarrow) \\
\lambda u.\lambda v.\lambda z.uP_1^{l,k} \ldots P_{|\Sigma|}^{l,k}P^{l,k}vz & \text{if } \delta(q_i,a_j) = (q_l,a_k,\leftarrow) \\
\lambda u.\lambda v.\lambda z.vR_1^{l,k} \ldots R_{|\Sigma|}^{l,k}R^{l,k}uz & \text{if } \delta(q_i,a_j) = (q_l,a_k,\rightarrow);
\end{array}\right. \\
P_i^{l,k} & \equiv & \lambda u.\lambda v.\lambda z.AC(\Sigma)(\lambda w.xz(\lambda x.xu\lceil a_i\rceil^{\Sigma}w\lceil q_l\rceil^{Q}))\lceil a_k\rceil^{\Sigma}v \\
P^{l,k} & \equiv & \lambda v.\lambda z.AC(\Sigma)(\lambda w.xz(\lambda x.x\lceil \varepsilon\rceil^{\Sigma}\lceil a_{blank}\rceil^{\Sigma}w\lceil q_l\rceil^{Q}))\lceil a_k\rceil^{\Sigma}v \\
R_i^{l,k} & \equiv & \lambda v.\lambda u.\lambda z.AC(\Sigma)(\lambda w.xz(\lambda x.xw\lceil a_i\rceil^{\Sigma}v\lceil q_l\rceil^{Q}))\lceil a_k\rceil^{\Sigma}u \\
R^{l,k} & \equiv & \lambda u.\lambda z.AC(\Sigma)(\lambda w.xz(\lambda x.xw\lceil a_{blank}\rceil^{\Sigma}\lceil \varepsilon\rceil^{\Sigma^*}\lceil q_l\rceil^{Q}))\lceil a_k\rceil^{\Sigma}u
\end{array}$$

It is routine to prove the thesis. □

At this point, we can give the desired simulation result:

Theorem 36 (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 $\mathcal{U}(\mathcal{M},\Delta) \equiv \lambda u.\mathcal{I}(\mathcal{M},\Delta)(\lambda x.\mathcal{T}(\mathcal{M})(\lambda y.\mathcal{F}(\mathcal{M},\Delta)(\lambda w.w)y))y)u$. 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 (extended version). available at `https://sites.google.com/site/beniaminoaccattoli/tech-report.pdf`.

[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 derivational complexity for head reduction (extendedversion). Available at `http://www.cs.unibo.it/~dallago/idchr.pdf`.

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

[6] B. Accattoli and D. Kesner. The permutative λ-calculus. To appear in the proceedings of LPAR 2012.

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

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

[10] 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.

[11] 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.

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

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

[14] 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.

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

[16] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

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

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

[19] 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.

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

[21] 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.