The Abstract Machinery of Interaction*

Beniamino Accattoli¹, Ugo Dal Lago²,³, and Gabriele Vanoni²,³

¹ Inria & École Polytechnique, Palaiseau, France
² University of Bologna, Italy
³ Inria Sophia Antipolis, France

Abstract. We provide an original reformulation of an abstract machine introduced by Danos and Regnier based on Girard’s Geometry of Interaction. We present our machine as acting directly on \(\lambda\)-terms with simple sharing annotations, rather than on linear logic proof-nets. The inductive nature of the \(\lambda\)-calculus allows direct and self-contained proofs of soundness and adequacy, which we consider with respect to linear head evaluation. Moreover, we improve the literature showing that the denotational semantics induced by our machine is invariant for open terms and erasing steps, solving a notorious issue of the Geometry of Interaction.

Keywords: \(\lambda\)-calculus · Geometry of Interaction · Abstract Machines.

1 Introduction

The advantage, and at the same time the drawback, of the \(\lambda\)-calculus is its distance from low-level, implementative details. It comes with just one rule, \(\beta\)-reduction, and with no indications about how to implement it. It is an advantage when reasoning about programs expressed as \(\lambda\)-terms. It is a drawback, instead, when one wants to implement the \(\lambda\)-calculus, or do complexity analyses, because \(\beta\)-steps do not look as atomic operations. In particular, terms can grow exponentially with the number of \(\beta\)-steps, a degeneracy known as size explosion, which is why \(\beta\)-reduction cannot be reasonably implemented as it is specified.

Environment Machines. Implementations solve this issue by evaluating the \(\lambda\)-calculus up to sharing of sub-terms, where sharing is realized through a data structure called environment, collecting the sharing annotations generated by the machine during the execution, one for each encountered \(\beta\)-redex. For common weak evaluation strategies (i.e. that do not inspect under \(\lambda\)-abstractions) such as call-by-name/value/need, the number of \(\beta\)-steps is a reasonable time cost model [15,47,21]. Environment machines—whose most famous examples are Landin’s SECD [37], Felleisen and Friedman’s CEK [28] or Krivine’s KAM [36]—can be extended to open terms and tuned to run within a linear overhead with respect to the number of \(\beta\)-steps [13,11];—said differently, they respect the time cost model. An overview by Accattoli is in [4].

* The Second and Third Authors are supported by the ANR project 16CE250011 REPAS, and the ERC Consolidator Grant DIAPASoN DLV-818616.
For space, the situation is different. First of all, no reasonable cost models are known\footnote{Recent news is that Forster, Kunze, Roth have found a reasonable cost model for space in the $\lambda$-calculus \cite{ForsterKunzeRoth}, currently submitted.} Then, environment machines store information for every $\beta$-step, therefore using space linear in time, which is the worst possible use of space\footnote{On sequential models space cannot exceed time, as one needs a unit of time to use a unit of space.}.

**Beyond Environments.** In practice, frameworks based on the $\lambda$-calculus are invariably implemented using environments. Nonetheless, the lack of a fixed execution schema for the $\lambda$-calculus leaves open, in theory, the possibility of alternative implementation schemes. The theory of linear logic indeed provides a completely different style of abstract machines, rooted in Girard’s Geometry of Interaction \cite{Girard:1990} (shortened to GoI). These GoI machines were pioneered by Danos and Regnier and Mackie in the nineties \cite{Danos:1997,Danos:1999}. The basic idea is that the machine does not use environments, while it keeps track of information that allows retrieving previous $\beta$-redexes, by using a data structure called token, saving information about the history of the computation. The key point is that the token does not store information about every single $\beta$-redex, thus disentangling space-consumption from time-consumption. In other words, GoI machines are good candidates for space-efficient implementation schemes, as first shown by Dal Lago and Schöpp \cite{DalLago:2018}. The price to pay is that the machine wastes a lot of time to retrieve $\beta$-redexes, so that time is sacrificed for space. The same, however, happens with space-efficient Turing machines.

**The Interaction Abstract Machine.** The original GoI machine is the Interaction Abstract Machine (IAM) by Danos and Regnier \cite{Danos:1997,Danos:1999}, formulated on linear logic proof-nets as a reversible automaton. In \cite{Danos:2015}, Danos, Herbelin, and Regnier prove that the IAM is sound with respect to linear head evaluation, a refinement of head evaluation, arising from the linear logic decomposition of the $\lambda$-calculus.

Soundness of GoI machines amounts to show that the they provide a denotational semantics invariant by linear head evaluation—see Sect. 6 for an overview of the difference with environment machines. Danos and Regnier’s proof of soundness for the IAM is indirect, as it follows from a sequence of results relating the IAM to AJM games, AJM games to HO games, HO games to another abstract machine, the PAM, and finally the PAM to linear head evaluation.

The understanding of the IAM requires advanced expertise in linear logic, and it is thus out of scope for most people with an interest in the implementation of functional languages. Moreover, the soundness proof requires further expertise in game semantics, and it is not as neat as for environment machines. Finally, environment machines are a completely different paradigm, that can even be misleading when dealing with the IAM.

The aim of this paper is to recast the IAM in a setting not requiring any background in linear logic or game semantics, and to guide the reader through the subtleties of the IAM. At the same time, providing a clean and self-contained technical development, and improving on some of the properties of the IAM.
The Interaction Abstract Lambda Machine. The first contribution of the paper is a formulation of the IAM as a machine acting on $\lambda$-terms with simple sharing annotations (aka explicit substitutions) rather than proof nets. The starting point of our Interaction Abstract Lambda Machine (IALM) is seeing a position in the code $t$ (what is usually the position of the token on the proof net representation of $t$) as a pair $(u, C)$ of a sub-term $u$ and a context $C$ such that $C(u) = t$. These positions are nothing else but a readable presentation of pointers.\footnote{For the acquainted reader, they play a role akin to labels in Lévy’s labeled $\lambda$-calculus, itself having deep connections with the IAM \cite{14}}

Direct Soundness and Adequacy Proofs. We do more than simply changing the syntax. As Danos and Regnier, we show that the IALM provides a denotational semantics of linear head evaluation. In contrast to Danos and Regnier, however, our proof of soundness is self-contained, and based on a variation over Sands’ improvements \cite{46}, a natural notion of bisimulation. We also give a clean and direct proof of adequacy, that is, the fact that the machine semantics is not empty if and only if linear head evaluation terminates. A contribution of this work, we believe, precisely lies in the way in which we state and prove these results.

Exhaustible States. A first novelty of our study is that we identify a new invariant of the IALM—probably of independent interest—based on what we call exhaustible states. Informally, a state of the IALM is exhaustible if its token can be emptied in a certain way, somehow mimicking the computation which leads to the state itself. The invariant is an essential ingredient of the proof of soundness.

Open Terms and Erasing Steps. A second relevant point is that our semantics is also naturally invariant when considering open terms and erasing steps. It is a novelty, since GoI is usually not invariant for open terms and erasing steps. This crucial improvement is better explained referring to the GoI presentation in terms of paths in a proof net. Our approach considers only paths starting on the conclusion of the net corresponding to the output of the $\lambda$-term, while the IAM considers paths between any two conclusions of the net. The intuitionistic nature of $\lambda$-terms indeed gives a privileged point of observation, that turns out to be the essential ingredient for the invariance of the interpretation for open terms with respect to erasing steps.

The Aim of the Paper. This paper is only the last chapter of a long-time endeavor by the authors directed to understanding complexity measures and implementation schemas for the $\lambda$-calculus. It is our first step towards the exploration of the time-space tradeoff in GoI machines. We provide an original reformulation of the original machine by Danos and Regnier, together with a neat self-contained development, new concepts (exhaustible states) and improved properties (invariance for open terms and erasing steps). The aim is to set the ground for a formal, robust, and systematic study of GoI machines, their complexity, and—
particular—of the time-space tradeoff, that will be rather carried out in future work.

Three guiding ideas of our work are:

- **Exporting technology**: making the machinery of interaction accessible to a wider audience, not necessarily expert in linear logic or game semantics,
- **Connecting with environment machines**: stressing the many differences with environment machines, for readers with a background in implementations,
- **Winking at proof assistants**: developing definitions and proofs inductively, having in mind a potential future formalization in proof-assistants, which is instead out of scope if the underlying syntax is graphical.

**Related Work on GoI.** This is certainly not the first paper on the GoI. Indeed, the literature on the topic and its applications is huge, and goes from Girard’s original papers [32], to Abramsky et al’s reformulation using the Int-construction [1], Danos and Regnier’s using path algebras [24], Ghica’s applications to circuit synthesis [31], together with extensions by Hoshino, Muroya, and Hasuo to languages with various kinds of effects [34]. In all these cases, the GoI interpretation, even when given on $\lambda$-terms, goes through linear logic (or symmetric monoidal categories) in an essential way.

In [26], Danos and Regnier introduce another GoI machine, the JAM, as an optimization of the IAM. Our setting can smoothly accommodate the JAM, but we omitted it for lack of space. A similar optimization is also considered by Fernandez and Mackie in [29]. Laurent gives a GoI machine for the additive connectives of linear logic in [38].

The GoI has also been studied in relationship with implementation of functional languages, by Gonthier, Abadi and Levy to study optimal implementations [33], and by Mackie with his Geometry of Interaction Machine for PCF [39] and System T [40].

Recently, the space-efficiency studied by Dal Lago and Schöpp [22] is exploited by Mazza in [42] and, together with Terui, in [43]. Dal Lago and co-authors have also introduced variants of the IAM acting on proof nets for a number of extensions of the $\lambda$-calculus [18,19,20,23].

Curien and Herbelin study abstract machines related to game semantics and the IAM in [16,17]. Muroya and Ghica have recently studied the GoI in combination with rewriting and abstract machines in [45].

**Related Work on Environment Machines.** Environment machines for the $\lambda$-calculus have been recently closely scrutinized as for their time efficiency. Before 2014, the topic had been mostly neglected—the two only examples are by Blelloch and Greiner in 1995 [15] and by Sands, Gustavsson, and Moran in 2002 [47]. Since 2014—motivated by advances by Accattoli and Dal Lago on time cost models for the $\lambda$-calculus [4]—Accattoli and co-authors have explored time analyses of environment machines from different angles [61,178].

**Proofs.** Omitted proofs are in the Appendix. In case of acceptance, this long version with Appendix will be made available on arXiv.
2 A Gentle Introduction to the Geometry of Interaction

This section is meant to be an informal introduction to Girard’s Geometry of Interaction as implemented by the IALM, the abstract machine we are introducing in this paper. As such, it can be safely skipped by the reader familiar with GoI. Readers acquainted with environment machines, instead, are invited not to look for connections with what they know—the IALM computes in a radically different way.

Suppose one wants to evaluate the term $t = \lambda z. (\lambda x.x)(\lambda y.y)$, whose head normal form is $\lambda z.\lambda y.y$. Moreover, let us suppose one wants to do so without any form or rewriting nor any substitution, by just looking for the head variable of its (head) normal form, namely $y$. The IALM does precisely so by traveling around $t$ in a purely local way.

First of all, positions in $t$ are captured, in the state of the IALM, as pairs $(C, u)$ where $C$ is a context and $C(u) = t$. There are three such positions which are relevant in our example, namely those corresponding to the three occurrences of variables in $t$, which we give explicit names to.

$$C_L^x = \lambda z. (\lambda x.x)(\lambda y.y) \quad C_R^y = \lambda z. (\lambda x.x)(\lambda y.y) \quad C_y = \lambda z. (\lambda x.x)(\lambda y.y)$$

Then a state of the machine is a position together with the token, which is formalized as a pair of stacks, and a direction $\downarrow$ or $\uparrow$ called polarity. If we start from the initial state $(t, \langle \cdot \rangle, \epsilon, \epsilon, \downarrow)$—where $(t, \langle \cdot \rangle)$ is the initial position and the two stacks are empty—the IALM looks for the head variable of $t$. It does not perform any transition, however, because $t$ is already in weak head normal form.

In contrast to environment machines, that are either weak (that is, never enter abstractions) or strong (they enter into all abstractions), the IALM has a finer mechanism that allows entering into some abstractions. In particular, each $p$ in the initial stack allows the machine to go under one of the head abstractions. Thus, starting from $(t, \langle \cdot \rangle, p, \epsilon, \downarrow)$ yields the computation below, of which we consider the first four transitions. These steps look for the head variable of $t$: indeed, we are performing nothing more than a visit of the leftmost branch of $t$, called the spine, until we find a variable.

<table>
<thead>
<tr>
<th>Sub-term</th>
<th>Context</th>
<th>IO stack</th>
<th>Signature stack</th>
<th>Polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda z. (\lambda x.x)(\lambda y.y)$</td>
<td>$\langle \cdot \rangle$</td>
<td>$p$</td>
<td>$\epsilon$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$(\lambda x.x)(\lambda y.y)$</td>
<td>$\lambda z.\langle \cdot \rangle$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$\lambda x.x$</td>
<td>$\lambda z.\langle \cdot \rangle(\lambda y.y)$</td>
<td>$p$</td>
<td>$\epsilon$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$xx$</td>
<td>$\lambda z. (\lambda x.(\cdot)) (\lambda y.y)$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\lambda z. (\lambda x.(\cdot)x)(\lambda y.y)$</td>
<td>$p$</td>
<td>$\epsilon$</td>
<td>$\downarrow$</td>
</tr>
</tbody>
</table>

Two things are worth being observed here.

– *Pushing and popping $p$: the first stack, called the IO-stack, accounts for the abstractions and applications encountered along the track: the symbol $p$ is pushed on applications, and pulled on abstractions (when the polarity is $\downarrow$).*
No saving of arguments: contrary to environment machines, the arguments of encountered applications—potentially contributing to β redexes—are not saved, this way saving space, and disentangling space from time.

When arrived in the state \((x, \lambda z. (\lambda x. \langle \cdot \rangle x) (\lambda y. y), p, \epsilon)\), we proceed by looking at the term that would be substituted for \(x\) during (linear) head evaluation (linear head evaluation is introduced in the next section, here the reader can think of head evaluation). In the KAM, this term is available in the applicative stack, while in the IALM, this needs to be looked for. This is the purpose of the next two steps.

<table>
<thead>
<tr>
<th>Sub-term</th>
<th>Context</th>
<th>IO stack</th>
<th>Signature stack</th>
<th>Polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(\lambda z. (\lambda x. \langle \cdot \rangle x) (\lambda y. y))</td>
<td>(p)</td>
<td>(\epsilon)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>(\lambda x.x)</td>
<td>(\lambda z. (\lambda x. \langle \cdot \rangle (\lambda y. y)))</td>
<td>((x, C^L_x, \epsilon) \cdot p)</td>
<td>(\epsilon)</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\lambda y.y)</td>
<td>(\lambda z. (\lambda x.x) \langle \cdot \rangle)</td>
<td>(p)</td>
<td>((x, C^L_x, \epsilon))</td>
<td>(\downarrow)</td>
</tr>
</tbody>
</table>

Two crucial aspects of the IALM show up here.

- **Phases**: the IALM starts looking for the argument of the given occurrence of \(x\), i.e. the term that should be substituted for \(x\), from a natural place, namely the \(\lambda\)-abstraction which binds that occurrence. Rather than traveling down in the term (\(\downarrow\)), it is now time to traveling up (\(\uparrow\)). One needs to keep track of which of the (possibly many) occurrences of the bound variable one is coming from. This is done by simply pushing the position of the found occurrence of \(x\) in the IO-stack, and switching the machine in upward modality \(\uparrow\).

- **Signature stack and copies of sub-terms**: the upward journey is guided by the context (note the blue color). It ends when entering inside an application from its left argument: this indeed witnesses the presence of a redex. The journey then naturally goes on, repeating the same \(\downarrow\)-search of the head variable in the application’s right argument. Observe that the second stack, called the signature stack, gets touched for the first time. Its role is to keep track of which (virtual) copy of an argument one is currently traveling inside. This is a key point: the machine never copies any sub-term, but a sub-term may be used differently in different moments of the evaluation process, which is why the machine retains information to distinguish between different uses.

The IALM is now facing the sub-term \(\lambda y.y\), which has been virtually substituted for the first occurrence of \(x\), this way forming the virtual redex \((\lambda y.y)x\). This is witnessed by the presence of a \(p\) symbol in the IO-stack. After a couple of steps, we are back at the very same position, but now traveling upwards, to find the argument of the above mentioned redex to be substituted on \(y\).

<table>
<thead>
<tr>
<th>Sub-term</th>
<th>Context</th>
<th>IO stack</th>
<th>Signature stack</th>
<th>Polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda y.y)</td>
<td>(\lambda z. (\lambda x.x) \langle \cdot \rangle)</td>
<td>(p)</td>
<td>((x, C^L_x, \epsilon))</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>(y)</td>
<td>(\lambda z. (\lambda x.x) (\lambda y. \langle \cdot \rangle))</td>
<td>(\epsilon)</td>
<td>((x, C^L_x, \epsilon))</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>(\lambda y.y)</td>
<td>(\lambda z. (\lambda x.x) \langle \cdot \rangle)</td>
<td>((y, C_y, \epsilon))</td>
<td>((x, C^L_x, \epsilon))</td>
<td>(\uparrow)</td>
</tr>
</tbody>
</table>
Next, we look for the argument of the (virtual) copy of \( \lambda y. y \) we are stumbling upon, realizing that, correctly, it is simply the second occurrence of \( x \).

Further steps deal with the search for the term meant to replace the second occurrence of \( x \). This term is \( \lambda y. y \) again. Since we are in front of an abstraction and no \( p \) are present on top of the IO-stack, the IALM terminates the execution.

It is worth noticing that, since the normal form of \( t \) is \( \lambda z. \lambda y. y \), one \( p \) in the initial IO-stack is not enough to inspect the head variable of \( t \). In fact one needs one \( p \) for each head abstraction in the head (linear) normal form of the term. In the case of our example, if one had started from \((t, \langle \cdot \rangle, p^2, \epsilon)\), the IALM would have finished the computation with the token in the root of \( t \), unveiling part of the structure of its head (linear) normal form. In particular, one can read the head variable in the IO-stack.

### 3 Linear Head Evaluation

Here we define linear head evaluation, that is the operational semantics with respect to which the IALM shall be proved sound.

**The Linear Substitution Calculus.** Linear head evaluation, introduced by Mascari and Danos & Regnier in [41,27] as a strategy on proof nets, admits various presentations. The simple one adopted here, noted \( \bowtie \), is flexible. It was introduced by Accattoli in [2], formulated as a strategy of a \( \lambda \)-calculus with explicit sharing, the **linear substitution calculus**.\(^7\) The LSC presentation of \( \bowtie \) has isomorphic alternative formulations on proof nets [5], environment machines [6], processes [3], and multi types [10], which is why we use it.

\(^7\) The LSC is a subtle reformulation of Milner’s calculus with explicit substitutions [44,35], inspired by Accattoli and Kesner structural \( \lambda \)-calculus [12]
Linear head evaluation is to the LSC—and thus proof nets for the $\lambda$-calculus—what head evaluation is to the $\lambda$-calculus.

**LSC Terms.** Let $\mathcal{V}$ be a countable set of variables. Terms of the *linear substitution calculus* (LSC) are defined by the following grammar.

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

The construct $t[x\leftarrow u]$ is called an *explicit substitution* or ES. *Free* and *bound* variables are defined as usual: both $\lambda x.t$ and $t[x\leftarrow u]$ bind $x$ in $t$ (but not in $u$ for the ES). Terms are considered modulo $\alpha$-equivalence, which allows to appropriately define the capture-avoiding (meta-level) substitution of all the free occurrences of $x$ for $u$ in $t$, noted $t\{x\leftarrow u\}$ and not to be confused with $t[x\leftarrow u]$.

**Contexts and Plugging.** The LSC makes a crucial use of contexts to define its operational semantics. First of all, we need a notion of substitution context, that simply packs together various ES:

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

Then, we also need a more general notion of context. The study of the IALM actually requires a notion of *leveled context*, introduced right next, that is more informative than what needed to simply define the LSC.

*Leveled contexts* are defined by the following grammar.

$$\begin{align*}
C_0 & ::= \langle \cdot \rangle \mid \lambda x.C_0 \mid C_0 t \mid C_0[x\leftarrow t]; \\
C_{n+1} & ::= C_{n+1} t \mid \lambda x.C_{n+1} \mid C_{n+1}[x\leftarrow t] \mid tC_n \mid t[x\leftarrow C_n].
\end{align*}$$

Contexts of level 0 are also called *head contexts* and are denoted by $H, K, G$. A context in the form $C_n$ is said to be $n$-ary and $C_n$ is the set of all $n$-ary contexts, for every $n \in \mathbb{N}$. $\mathcal{C} = \bigcup_{n \in \mathbb{N}} C_n$ is simply the set of all contexts.

The index $n$ in $C_n$ counts the number of arguments and ES in which the hole $\langle \cdot \rangle$ is contained in $C_n$. Such an index has a natural interpretation in linear logic terms. According to the standard (call-by-name) translation of the $\lambda$-calculus into linear logic proof-nets, in a context $C_n$ the hole lies inside exactly $n$ $!$-boxes. Please note that both Mackie’s and Danos and Regnier’s instead used the call-by-value encoding of the $\lambda$-calculus into proof-nets.

Given a context $C_n$, the *plugging* $C_n(t)$ of a term $t$ in $C_n$ is defined by replacing the hole $\langle \cdot \rangle$ with $t$, potentially capturing the free variables of $t$—the plugging $C_n(C_m)$ of a context for a context is defined similarly. Plugging is also used for substitution contexts, but we write in a post-fixed manner, that is $\langle t \rangle L$, to stress that the ES actually appears on the right of $t$.

The level of a context shall be omitted when not relevant to the discussion, however please note that any ordinary context can be written *in a unique way* as a leveled context, so that the omission is harmless.
Positions. A position (of level \( n \)) in a term \( u \) is a pair \((t,C_n)\) such that \( C_n(t) = u \). \( \mathcal{P} \text{OS} \) is the set of all positions.

Linear Head Evaluation. The LSC comes with a notion of reduction that resembles the decomposed, micro-step process of cut-elimination in linear logic proof-nets. Essentially, the meta-level of substitution \( t\{x\mapsto u\} \) is decomposed into a sequence of many replacements from \( t[x\mapsto u] \) of one occurrence of \( x \) in \( t \) with \( u \) at the time. Linear head evaluation, moreover, is the reduction that only replaces the head variable occurrence \( y \), if it is bound by an ES \([y\mapsto r]\) and leaves the other occurrences of \( y \), if any, bound by \([y\mapsto r]\).

The rewriting rules are first defined at top level and then closed by head contexts. A feature of the LSC is that contexts are also used to define the substitution rule at top level:

\[
\text{LIN} \text{E} \text{AR} \text{N} \text{AL} \text{ HEAD} \text{ EVALUATION} \quad \Rightarrow \\
\begin{array}{ll}
\langle \lambda x.t \rangle L & \rightarrow_{dB} \langle t[x\mapsto u] \rangle L \\
H(x)[x\mapsto t] & \rightarrow_{1s} H(t)[x\mapsto t] \\
t[x\mapsto u] & \rightarrow_{gc} t \quad \text{if } x \notin \text{fv}(t)
\end{array}
\]

\[\text{Contextual closure} \quad \frac{t \rightarrow_a u}{C(t) \rightarrow_a C(u)} \quad a \in \{dB, ls, gc\}.
\]

\[\text{Notation: } \vdash := \vdash_{dB} \cup \vdash_{ls} \cup \vdash_{gc} \]

Often, the literature does not include rule \( \rightarrow_{gc} \), responsible for erasing steps, in the definition of \( \vdash \). The reason is that \( \rightarrow_{gc} \) is strongly normalizing and it can be postponed. We include it to stress that the IALM is invariant also with respect to \( \rightarrow_{gc} \) on open terms.

Note that our definition of \( \vdash \) allows more than one \( \vdash \) redex at a time in a term. It is not a problem, as \( \vdash \) has the diamond property—this is standard.

Relationship with Head Evaluation, and Normal Forms. Linear head evaluation is studied at length in the literature, in particular its relationship with head evaluation is well known. On a given a term \( t \), linear head evaluation \( \vdash \) terminates on the linear head normal form \( \text{lnnf}(t) \) if and only if head evaluation \( \Rightarrow \) terminates on the head normal form \( \text{hnf}(t) \). Moreover, \( \text{hnf}(t) \) is obtained from \( \text{lnnf}(t) \) by simply unfolding ES, that is turning them into meta-level substitutions. A linear head normal form has the same shape \( \lambda x_1.\ldots \lambda x_k. (y_{t_1} \ldots t_h) \) of a head normal form but for the fact that each spine sub-term may be surrounded by a substitution context \( L_i \), that is, they have the cumbersome shape (where \( L_i \) surrounds \( \lambda x_1.\ldots \lambda x_k. (y_{t_1} \ldots t_h) \) and \( L_j' \) surrounds \( y_{t_1} \ldots t_j \)):

\[
(\lambda x_1.\langle \lambda x_2.\ldots \langle \lambda x_k(\langle \langle y \rangle L'_0 t_1 \rangle L'_1 \ldots t_h \rangle L'_h) \rangle L_k \ldots L_2) L_1
\]

(1)

where none of the ES in \( L_i \) and \( L_j' \) binds \( y \) (otherwise there would be a \( \vdash_{1s} \) redex). Unfolding the ES of a linear head normal form produces a head normal form having the same spine structure, that is, with the same abstractions, the same head variable and the same number of arguments—concretely, unfolding the term in (1) one obtains the head normal form \( \lambda x_1.\ldots \lambda x_k. (y_{u_1} \ldots u_h) \) for some \( u_1,\ldots, u_h \). Therefore, in the paper we shall refer to a \( \vdash \)-normal term up
to substitution \( \lambda x_1 \ldots \lambda x_k . (yu_1 \ldots u_h) \) meaning that we harmlessly ignore the substitution contexts around the spine sub-terms. Please note that we do not have any restriction on closed terms, and thus the number of \( \lambda \)-abstractions in the spine of \( \text{hnf}(t) \) and \( \text{hnf}(t) \) could also be 0.

4 The Interaction Abstract Lambda Machine

Here we introduce the data structures used by the IALM, present the transitions, and prove some simple properties of the machine. With respect to the overview in Section 2, terms are now taken in the LSC, that is, with explicit substitutions. Token Data and Signatures. The token is composed by two stacks, called IO stack \( \Psi \) and signature stack \( \Sigma \), that can contain two kinds of data. One are occurrences of the special symbol \( p \), needed to cross abstractions and applications. The other one is given by signatures, a recursively-defined data structure enriching a variable position with extra information. Namely, a signature \( e \) is either a position \( (x, C '< \lambda x.D_n') \) or a position \( (x, C 'D_n[x-t']) \) coming in both cases together with a list of length \( n \) of signatures, one for every level in \( D_n \).

Definition 1 (Signature). The set of signatures \( \mathcal{H} \) is the smallest set such that:

1. \( (x, C '< \lambda x.D_n'), \Sigma_n) \in \mathcal{H} \) if \( \Sigma_n \) is a sequence of \( n \geq 0 \) signatures;
2. \( (x, C 'D_n'[x-t']), \Sigma_n) \in \mathcal{H} \) if \( \Sigma_n \) is a sequence of \( n \geq 0 \) signatures.

Observe that \( \mathcal{H} \) is not empty precisely because \( n \) can be 0.

States. A state of the machine is given by a position, plus two stacks, modelling the token, and a mode of operation called polarity.

Definition 2 (IALM State). A state \( s \) of the IALM is a quintuple \( (t, C, \Psi, \Sigma, p) \) where:

1. \( t \) is a LSC-term: the Code Term;
2. \( C \) is a context: the Code Context;
3. \( \Psi \) is an element of \( (\{p\} \cup \mathcal{H})^* \): the Input-Output Stack;
4. \( \Sigma \) is an element of \( \mathcal{H}^* \): the Signature Stack;
5. \( p \) is an element in \( \mathcal{P} = \{\uparrow, \downarrow\} \): the Polarity.

We call \( S_{\text{IALM}} \) the set of all IALM states.

In the following, polarities are represented mostly via colors: the code term in red, to represent \( \downarrow \), and the code context in blue, to represent \( \uparrow \). This way, the fifth component is often omitted.

Initial States. The IALM always starts from states in the form \( s_{t,k} := (t, \langle \cdot \rangle, p^k, \varepsilon) \), where \( t \) is a term, \( k \geq 0 \), and \( \varepsilon \) stands for the empty stack. Intuitively, this means that we are evaluating the term \( t \) and that the machine is allowed to inspect \( k \) \( \lambda \)-abstractions of the linear head normal form of \( t \).

Note that there are many initial states for a given term \( t \), one for each IO stack \( p^k \)—this is in contrast to environment machines, that have only one initial state for every given term \( t \), as in that case \( t \) is the input.
### Sub-term Context IO stack Signature stack

<table>
<thead>
<tr>
<th>Sub-term</th>
<th>Context</th>
<th>IO stack</th>
<th>Signature stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ut$</td>
<td>$C$</td>
<td>$\psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$u$</td>
<td>$C(\cdot t)$</td>
<td>$p \cdot \psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\lambda x.t$</td>
<td>$C$</td>
<td>$p \cdot \psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$t$</td>
<td>$C(\lambda x.\cdot)$</td>
<td>$\psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$x$</td>
<td>$C(\lambda x.D_n)$</td>
<td>$\psi$</td>
<td>$\Sigma_n \cdot \Sigma$</td>
</tr>
<tr>
<td>$\lambda x.t$</td>
<td>$C$</td>
<td>$(x, C(\lambda x.D_n), \Sigma_n) \cdot \psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$x$</td>
<td>$C(\lambda x.D_n)$</td>
<td>$\psi$</td>
<td>$\Theta \cdot \Sigma$</td>
</tr>
<tr>
<td>$t[x\leftarrow u]$</td>
<td>$C$</td>
<td>$\psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$t$</td>
<td>$C(\cdot[x\leftarrow u])$</td>
<td>$\psi$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$x$</td>
<td>$C(D_n[x\leftarrow u])$</td>
<td>$\psi$</td>
<td>$\Sigma_n \cdot \Sigma$</td>
</tr>
<tr>
<td>$u$</td>
<td>$C(D_n(x)[x\leftarrow \cdot])$</td>
<td>$\psi$</td>
<td>$(x, C(D_n[x\leftarrow u]), \Sigma_n) \cdot \Sigma$</td>
</tr>
</tbody>
</table>

**Fig. 1.** Transitions for $\downarrow$-states, that depend on the code sub-term (first column).

**Transitions.** The IALM transition system is defined in two sets of transitions, Fig. 1 containing the transitions for $\downarrow$-states and Fig. 2 for the $\uparrow$-states. Of course, some transition change the polarity. Their union is noted $\rightarrow IALM$.

The transitions mimic on terms how usual token machines travel over proof-nets, with components simply defined by induction, with no reference to an underlying graphical formalism nor to linear logic.

**Game Intuition.** An intuition, connecting with game semantics, is that

- **Questions**: states $(t, C, \Psi, \Sigma)$ of polarity $\downarrow$ represent a query about the head variable (of the $\multimap$-normal form) of $t$, parameterized by the IO stack $\Psi$ read as input. When a head variable is found the polarity changes putting the answer—that is the position of the head variable—in a signature $e$ in the IO stack, now to be read as output.

- **Answers**: states $(t, C, \Psi, \Sigma)$ of polarity $\uparrow$ represent the process of answering, whose output is $\Psi$. This process may find that a sub-term $u$ should replace the head variable, say $x$, stored in the first signature $e$ of the output. Then the polarity changes to $\downarrow$, querying about the head variable of (the $\multimap$-normal form of) $u$. The machine also saves the signature of $x$ on the signature stack, to backtrack to it later on, once the query about $u$ is completed.

A state $s$ is **reachable** if $s_{t,k} \rightarrow IALM^* s$ for an initial state $s_{t,k}$.

**The Code Invariant.** Given a state $(t, C, \Psi, \Sigma)$ the position $(t, C)$ represents the position of the token $(\Psi, \Sigma)$ inside the term $C(t)$ travelling in the direction given by the polarity $p$. An inspection of the rules shows that $C(t)$ is an invariant of the execution, that is, the machine travels on a code that never changes.
Proposition 1 (Code invariant). If \( (t, C, \Psi, \Sigma, p) \rightarrow_{IALM} (u, D, \Phi, \Theta, p') \), then \( C(t) = D(u) \).

Final States. If the IALM starts on the initial state \( s_{t,k} = (t, \langle \cdot \rangle, p^k, \epsilon) \) the execution may either never stop or end in one of three possible final states. To explain them let \( \lambda x_0 \ldots \lambda x_i. (y u_1 \ldots u_j) \) be the linear head normal form \( \text{lnnf}(t) \) of \( t \) up to substitutions. The three kinds of final states of the machine are:

- **Failure**: \( \langle \lambda x.t, C, \epsilon, \Sigma \rangle \), this is the way of the machine to say that \( i > k \), that is, \( \text{lnnf}(t) \) has more head abstraction of those that the input/question \( k \) of the initial state \( s_{t,k} \) allowed to explore.

- **Open success**: \( \langle y, C, p^n, \Sigma \rangle \), meaning that the machine found the head variable, and it is the *free* variable \( y \). Note that if instead the variable \( y \) is bound by a \( \lambda \)-abstraction or an ES, then the machine moves.

- **Bound success**: \( \langle t, \langle \cdot \rangle, p^n \cdot e \cdot p^p, \Sigma \rangle \), meaning that the head variable has been found and it is \( y = x_m \). When the machine \( \downarrow \)-travels on the head variable \( y \), and it is abstracted, the signature \( e \) containing \( x \) is put on the IO stack and the polarity switches—the answer has been found. The sequence \( p^n \) on top of the \( e \) in the final state comes from the \( \uparrow \) backtracking along the spine of \( \text{lnnf}(t) \) for the equivalent of \( m \) abstractions, each one adding one \( p \). At this point the IALM stops. Thus the abstraction binding \( y \) is \( \lambda x_m \).

In the case \( \text{lnnf}(t) \) is in the form \( y u_1 \ldots u_j \), the third kind of final state is never reached, since that would imply that the head variable is bound. In general, the exhaustible state invariant of the next section shall guarantee that the IALM never stops in states that are not in one of these three shapes.
The Semantics. The characterization of final states allows to define a semantic interpretation of a term, that in the following sections is shown to be sound and adequate with respect to linear head evaluation $\triangleright$.

**Definition 3 (IALM Semantics).** We define the IALM semantics of LSC-terms by way of a family of functions $[[\cdot]]_k : \Lambda \to \mathbb{N} \times \mathbb{N} \cup \{\downarrow, \bot\} \cup \mathcal{V}$, where $k \in \mathbb{N}$, defined as follows.

$$[[t]]_k = \begin{cases} 
(h,j) & \text{if } (t,\langle\cdot\rangle,p^k,\epsilon) \text{ is terminating in the state } (t,\langle\cdot\rangle,p^h \cdot \cdot p^j,\epsilon), \\
x & \text{if } (t,\langle\cdot\rangle,p^k,\epsilon) \text{ is terminating in the state } (x,C,p^h,\Sigma), \\
\downarrow & \text{if } (t,\langle\cdot\rangle,p^k,\epsilon) \text{ is terminating in the state } (\lambda x.u,C,\epsilon,\Sigma), \\
\bot & \text{otherwise.}
\end{cases}$$

**4.1 Further Basic Properties of the IALM**

**Reversibility.** The IALM is not only deterministic, but also bideterministic, or reversible. In other words, for each state $s$ there cannot be two distinct states $s'$ and $s''$ such that $s' \rightarrow_{IALM} s$ and $s'' \rightarrow_{IALM} s$. Inspecting the rules allows one to see this property. Moreover, the IALM itself gives a way to revert a computation, by simply switching the polarity.

**Proposition 2 (Reversibility).** If $(t,C,\Psi,\Sigma,p) \rightarrow_{IALM} (u,D,\Gamma,\Theta,q)$, then $(u,D,\Gamma,\Theta,q^\downarrow) \rightarrow_{IALM} (t,C,\Psi,\Sigma,p^\downarrow)$.

**The Proper State Invariant.** Now we explain two easy invariants of the stacks composing the token. The signature stack $\Sigma$ contains information needed when the token enters an argument of an application or an ES, (i.e. when the token enters a box) and to restore it when the token exits from them. This is witnessed by the fact that on reachable states the length of $\Sigma$ is equal to the level of the code context $C$—this is the balanced invariant below.

About the IO stack, note that every time that there is a change from a $\downarrow$-state to an $\uparrow$-state (or vice versa) a signature is pushed, or popped, from the IO stack $\Psi$. Thus, for reachable states, the number of signatures in $\Psi$ gives the polarity of the state. These intuitions are formalized by the proper invariant below, for which we need some definitions.

Given an input-output stack $\Psi$ we denote with $|\Psi|_\epsilon$ the number of signatures in $\Psi$. Given a polarity $p$ we use $p^n$ for the polarity obtained by switching $p$ exactly $n$ times (i.e., $\downarrow^0 = \downarrow$, $\uparrow^0 = \uparrow$, $\downarrow^{n+1} = \uparrow^n$ and $\uparrow^{n+1} = \downarrow^n$).

**Definition 4 (Balanced and proper states).** A state $s = (t,C_n,\Psi,\Sigma,p)$ is

1. balanced, if the level $n$ of the position $(t,C_n)$ is equal to the length of the signature stack $\Sigma$;
2. proper, if it is balanced and the polarity $p$ is $|\Psi|_\epsilon$;
3. anti-proper, if it is balanced and the polarity $p$ is $|\Psi|_\epsilon$.

As expected, all reachable states are balanced and proper.
Lemma 1 (Proper state invariant). Let $s$ be a reachable state. Then $s$ is proper.

Note that the IO stack $\Psi$ of a proper $\uparrow$-state always contains at least one signature, which is why it can be seen as the answer to a query about the head variable.

All reachable states are proper, but anti-proper states shall have a role, too, although of a genuinely technical nature: the exhaustible state invariant of the next section shall be based on some anti-proper states, and on their dynamics.

Pumping. An important property that is used in the proofs is a sort of pumping lemma for the IALM IO stack $\Psi$. Intuitively, the IALM consumes the next entry of the initial input only when the question asked by the previous one(s) has been fully answered.

Lemma 2 (Pumping). If $(t,C,\epsilon,\Sigma,p) \rightarrow^n_{IALM} (u,D,\Psi,\Theta,q)$, then $(t,C,\Gamma,\Sigma,p) \rightarrow^n_{IALM} (u,D,\Psi\cdot\Gamma,\Theta,q)$.

Monotonicity of Runs. Last, increasing the input from $p^k$ to $p^{k+1}$ increases the length of the IALM run, except if the run of $p^k$ was successful.

We write $|t|_k$ for the length of the IALM run of initial state $s_{t,k} := (t,\langle \cdot \rangle, p^k, \epsilon)$, that is for the length of the maximum sequence of transitions $s_{t,k}$, if the IALM terminates, and $|t|_k = \infty$ if the machine diverges. The next lemma compares run lengths, for which we consider that $i < \infty$ for every $i \in \mathbb{N}$ and $\infty \neq \infty$. We also write $s_{t,k+1}^n$ for the state such that $s_{t,k+1}^n \rightarrow^n_{IALM} s_{t,k+1}^n$, if it exists.

Lemma 3 (Monotonicity of runs). The length of runs cannot decrease if the input increases, that is, $|t|_k \leq |t|_{k+1}$. Moreover, if $|t|_k = n \in \mathbb{N}$ and the final state $s_{t,k}^n$ is bound (resp. open) successful then $|t|_k = |t|_h$ for every $h > k$ and the final state $s_{t,h}^n$ is bound (resp. open) successful.

5 The Exhaustible State Invariant

This section defines and proves a key invariant of reachable IALM states. The intuition is that whenever a signature $e$ occurs in such a state, it is there for a reason, because no signatures occur in initial states, and transitions only add signatures to which the machine is supposed to come back to. In particular, one can somehow revert the process which is responsible for having put $e$ in the state, and exhaust $e$.

Why It Is Needed. The exhaustible state invariant is meant to show that some undesirable configurations never arise. On states such as $(\lambda x.t, C, e \cdot \Psi, \Sigma)$ the IALM requires the signature $e$ to have the shape $(x, C(\lambda x.D), \Sigma)$, that is, to be associated to a position isolating an occurrence of $x$ in $\lambda x.t$, otherwise the machine is stuck. Similarly, on states such as $(t, C(D(x)[x\leftarrow (\cdot)]), \Psi, e \cdot \Sigma)$ the position of $e$ is expected to isolate an occurrence of $x$ in $D(x)$, or the machine is
stuck. Luckily, the machine never gets stuck for these reasons, and exhaustible states are the technical tool to prove it. Therefore, the invariant is needed to characterize the final states of the IALM.

One could redefine the transitions of the IALM asking—for these states—to jump to whatever variable position is in the signature stack of states in the IO stack of states, one accounting for the signatures in the IO stack of states. Then the IALM would not get stuck, and the invariant would not be needed for characterizing final states, but we would then need it for soundness—there is no easy way out.

First Reading? Then we suggest to skip this section, as the invariant is involved. It is nonetheless a key technical ingredient and one of the contributions of the paper. The key result used in the rest of the paper is Corollary 1. Preliminaries. Exhaustible states rest on some test states for their signatures. More specifically, each signature $e$ in a state $s$ has an associated test state $s_e$, supposed to test the exhaustibility of $e$ in $s$. Actually, there shall be two classes of test states, one accounting for the signatures in the IO stack of $s$, called IO states, and one for the those in the signature stack of $s$, called outer states.

Outer States. Let $(u, C_{n+1})$ be a position. Then, for every decomposition of $n$ into two natural numbers $m, k$ with $m + k = n$, we can find contexts $C_m$ and $C_k$, and a term $r$ satisfying

- either $t = C_m\langle r C_k \langle u \rangle \rangle$. In this case, the $m + 1$-outer context of the position $(u, C_{n+1})$ is the context $O_{m+1} := C_m\langle r \langle \rangle \rangle$ of level $m + 1$ and the $m + 1$-outer position is $(C_k\langle u \rangle, O_{m+1})$;
- or $t = C_m\langle r [x \leftarrow C_k \langle u \rangle] \rangle$. In this case, the $m + 1$-outer context of the position $(u, C_{n+1})$ is the context $O_{m+1} := C_m\langle r [x \leftarrow \langle \rangle] \rangle$ of level $m + 1$ and the $m + 1$-outer position is $(C_k \langle u \rangle, O_{m+1})$.

It is easy to realize that any position having level $n$ has unique $m$-outer context and $m$-outer position, for every $1 \leq m \leq n + 1$, and that, moreover, outer positions are hereditary, in the following sense: the $i$-outer position of the $m$-outer position of $(u, C_{n+1})$ is exactly the $i$-outer position of $(u, C_{n+1})$.

Definition 5 (Outer State). Let $s = (t, C_n, \Psi, e_n \cdot e_2 \cdot e_1, p)$ be a balanced state with $n \geq 1$, $1 \leq m \leq n$, and $(u, O_m)$ be the $m$-outer position of $(t, C_n)$. The $m$-outer state of $s$ is the state $\text{out}_m(s) := (u, O_m, e_m, e_2 \cdot e_1, ↑)$.

Note $m$-outer states do not depend on the polarity of the state, nor on the underlying IO-stack, and that they are stable by head translations of the position $(t, C_n)$, in the sense that if $t = H(r)$ then $(t, C_n, \Psi, \Sigma, p)$ and its head translation $(r, C_n(H), \Phi, \Sigma, p)$ induce the same outer states (because the two positions have the same outer positions and their last components are themselves the same).

IO States. The second class of states we are interested in are those in which we put in evidence the signatures in the IO-stack.

Definition 6 (IO States). Let $s = (t, C, \Psi, \Sigma, p)$ be a balanced state. Then the IO-states of $s$ are those states which can be written as $(t, C, \Phi \cdot e, \Sigma, ↑_\Phi \cdot e_1)$ where $\Psi = \Phi \cdot e \cdot \Phi'$ is any decomposition of the IO-stack $\Psi$ of $s$. 

\[ \Psi = \Phi \cdot e \cdot \Phi' \]
The notions of IO and Outer signatures implicitly put in evidence one signature, which is the one of which we shall test the nature in our invariant. This is captured in the following definition.

**Definition 7 (IO and Outer Signature).** Let \( s = (t, C, \Psi, \Sigma, p) \) be a balanced state. Then the IO-signature of \( s \) is the signature \( e \) such that \( \Psi = \Phi \cdot e \), if any. The outer signature of \( s \), instead, is the leftmost signature in \( \Sigma \).

Finally, to any signature one can associate a set of states, dubbed its signature states, which are those coherent with the variable in the state.

**Definition 8 (Signature State).** Given a signature \( e = (x, D, \Sigma) \), states in the form \( (x, D, \epsilon, \Sigma \cdot \Theta) \) are said to be generated by \( e \).

**The Exhaustibility Invariant.** After having introduced all the necessary preliminary ingredients, we can now state the invariant.

**Definition 9 (Exhaustible States).** \( E \) is the smallest set of those states \( s \) such that the following conditions hold.

1. **IO Decomposition:** for any IO-state \( s' \) of \( s \), it holds that \( s' \xrightarrow{\ast}_{IALM} s'' \in E \), where \( s'' \) is generated by the IO-signature of \( s' \).
2. **Outer Decomposition:** for any outer state \( s' \) of \( s \), it holds that \( s' \xrightarrow{\ast}_{IALM} s'' \in E \), where \( s'' \) is generated by the outer-signature of \( s' \).

States in \( E \) are called exhaustible.

Informally, exhaustible states are those for which every outer and IO state can be exhausted, i.e., rewritten into their signature states (themselves exhaustible). The set \( E \) being the smallest set of such states implies that checking that a state is exhaustible can be finitely certified, i.e. there must be a finitary proof of it.

**Lemma 4 (Exhaustible invariant).** Let \( s \) be a IALM reachable state. Then \( s \) is exhaustible.

The proof of Lemma 4 is long, but logically quite simple, being structured around a simple induction on the length of the run from the initial state to \( s \).

**Exhaustible and Final States.** We are now ready to prove that the IALM never gets stuck for a mismatch of signatures.

**Corollary 1 (Signatures never block the IALM).** Let \( s \) be a reachable state.

1. If \( s = (\lambda x.t, C, e \cdot \Psi, \Sigma) \) then \( e = (x, C(\lambda x.E), \Theta) \) for some context \( E \).
2. If \( s = (u, C(D|x| \langle \cdot \rangle), \Psi, e \cdot \Sigma) \) then \( e = (x, C(E|x| \leftarrow u), \Theta) \) for some context \( E \) and some stack \( \Theta \).

**Proof.**

1. By Lemma 4 \( s \) is exhaustible. By IO-decomposition, \( (\lambda x.t, C, e, \Sigma) \xrightarrow{\ast}_{IALM} (y, D, e, \Theta) \) for some \( y, D \), and \( \Theta \), thus \( s \) is not stuck, i.e. it can make a transition. This is possible only if \( e = (x, C(\lambda x.E), \Theta) \) for some context \( E \).
2. By Lemma 4 \( s \) is exhaustible. Then, by outer decomposition, its outer state \( s' = (u, C(D|x| \langle \cdot \rangle), e \cdot \Sigma) \) evolves into \( (z, F, e, \Gamma) \) for some \( z, F \), and \( \Gamma \), thus \( s' \) is not stuck, i.e. it can make a transition. This is possible only if \( e = (x, C(E|x| \leftarrow u), \Theta) \) for some context \( E \) and some stack \( \Theta \).
6 Soundness of the IALM, Explained

Soundness of the IALM has a nature that is fundamentally different from soundness of environment machines.

Soundness for Environment Machines. An environment abstract machine $M$ executes a term $t$ according to a strategy $\rightarrow$ if from the initial state $s_i$ of code $t$ it computes a representation of the normal form $\text{nf}_{\rightarrow}(t)$. In particular, the machine maintains a code representation of how the strategy $\rightarrow$ modifies the term $t$ they both evaluate. Soundness is a weak bisimulation between the transitions $s \rightarrow_M s'$ of the machine and the steps $t \rightarrow u$ of the strategy. In particular, a run $\rho_t$ of the machine on $t$ passes through some states representing $u$, and the final states $s_f$ of the machine decode to $\rightarrow$-normal forms.

Soundness for the IALM. The IALM, and more generally GoI machines, do implement strategies, but in a different way. The IALM has many initial states, therefore many runs, for a given code $t$, one for each possible initial IO stack $p^n$. Moreover, the machine does not trace how the strategy modifies the term. If $t \leadsto u$, a run of code $t$ never passes through a representation of $u$—soundness denotes something else. The idea is that, on a fixed input, the run of code $t$ is bisimilar to the run of code $u$. Notably, the latter is shorter—rewriting the code is a way of improving the associated IALM.

Note the difference with environment machines: there the bisimulation is between steps on terms and transitions on states. For the IALM, it is between transitions on states (of code $t$) and transitions on states (of code $u$).

On Not Computing Results. Another difference is that the IALM does not compute a code representation of the result $\text{hnf}(t)$. It recovers the micro information $\|t\|_k$ about it, by exploring only the immutable code $t$. This is in accordance with other models: space-efficient Turing machines do not compute the whole output but only single bits of it. To compute the spine of $\text{hnf}(t)$, one needs to compute $\|t\|_k$ for various values of $k$, one for each abstraction of the spine, starting each time with a different input $\Psi = p^k$, and then once more for the head variable (adding a $p$), if the machine ever terminates. On a linear head normal form $t$, the runs of the IALM become an immediate interactive reading of the spine of $t$. Inputs represent questions about the head of the normal form, the answer is encoded in the IO stack at the end of the run.

The Statement. Soundness is the fact that the computation of $\|t\|_k$ is an invariant of $\leadsto$ on $t$. The next section proves the following statement.

Theorem 1 (Soundness). If $t \leadsto u$, then $\|t\|_k = \|u\|_k$ for each $k \geq 0$.

7 Im-Proving Soundness

Preliminaries for Bisimulations. A deterministic transition system (DTS) is a pair $\mathcal{S} = (\mathcal{S}, \mathcal{T})$, where $\mathcal{S}$ is a set and $\mathcal{T} : \mathcal{S} \rightarrow \mathcal{S}$ a partial function. If $\mathcal{T}(s) = t$, ...
then we write \( s \rightarrow s' \), and if \( s \) rewrites in \( s' \) in \( n \) steps then we write \( s \rightarrow^n s' \).

We call \( F_S \) the subset of \( S \) containing all \( s \in S \) such that \( T(s) \) is undefined. \( s \) is terminating if there exists \( n \geq 0 \) and \( s' \in F_S \) such that \( s \rightarrow^n s' \). We call \( S_\uparrow \) the set of terminating states of \( S \) and by \( S \downarrow \) we denote \( S \setminus S_\uparrow \).

**Bisimulations and Improvements.** Inspired by [40], we adopt a notion of improvement that is a weak-bisimulation between two DTS preserving termination and guaranteeing that whenever \( s \) and \( q \) are related and terminating then \( q \) terminates in no more steps than \( s \)—the no-more-steps part implies that the definition is asymmetric in the way it treats the two DTS.

**Definition 10 (Improvement).** Given two DTS \( S \) and \( Q \), a relation \( R \subseteq S \times Q \) is an improving bisimulation, or simply an improvement, if \((s,q) \in R\) implies the followings, schematized in Fig. 3.

- Final state left: if \( s \in F_S \), then \( q \in F_Q \).
- Final state right: if \( q \in F_Q \), then \( s \rightarrow^n s' \), for some \( s' \in F_S \) and \( n \geq 0 \).
- Transition left: if \( s \rightarrow s' \), then there exists \( s'', q', n, m \) such that \( s' \rightarrow^m s'' \), \( q' \rightarrow^n q', s''Rq' \) and \( n \leq m + 1 \).
- Transition right: if \( q \rightarrow q' \), then there exists \( s', q'', n, m \) such that \( s \rightarrow^m s' \), \( q' \rightarrow^n q'', s'Rq'' \) and \( m \geq n + 1 \).

What improves along an improvement is the number of transitions required to reach a final state, noted as follows.

**Definition 11 (Evaluation length).** Given a DTS \((S, \cdot )\), the evaluation length map \(|\cdot | : S \rightarrow \mathbb{N} \cup \{\infty\} \) is defined as follows.

\[
|s| = \begin{cases} 
  n & \text{if } s \rightarrow^n s' \text{ and } s' \in F_S \\
  \infty & \text{if } s \in S_\uparrow
\end{cases}
\]

The next proposition collects the expected properties of improvements.

**Proposition 3.** Let \( R \) be an improvement on two DTS \( S \) and \( Q \), and \( sRq \).

1. Termination equivalence: \( s \in S_\uparrow \) if and only if \( q \in Q_\downarrow \).
2. Improvement: \( |s| \geq |q| \).

**Splitting the Rules.** Linear head reduction is made out of three rewriting rules, each one requiring a different improvement. Therefore, we study them separately in the next three sub-sections, and sum everything up afterwards.
7.1 Creation of Explicit Substitutions

Here we define a relation $\triangleright_d B$ on IALM states induced by rule $\triangleright_d B$, and prove it to be an improvement.

Steps, Positions, and Context Rewriting. Lifting a step $t \triangleright_d B u$ to a relation between a IALM state $s$ of code $t$ and a state $q$ of code $u$ requires changing all positions relative to $t$ in $s$ to positions relative to $u$ in $q$. A first point to note is that we also have to change all the positions in signatures and stacks, so that $\triangleright_d B$ has to relate positions, signatures, stacks, and states.

A second more technical aspect is that one needs to extend linear head evaluation to contexts. Let us explain why. Consider a step $t \triangleright_d B u$ where—for simplicity—the redex is at top level and the associated state $\langle \lambda x.r \rangle L w, \langle \cdot \rangle, \epsilon, \epsilon \rangle$ has empty stacks. This should be $\triangleright_d B$-related to a state $\langle r[x \leftarrow w] \rangle L, \langle \cdot \rangle, \epsilon, \epsilon \rangle$.

Let’s have a look at how the two states evolve:

\[
\begin{align*}
\langle \lambda x.r \rangle L w, \langle \cdot \rangle, \epsilon, \epsilon \rangle &\quad \triangleright_d B \quad \langle r[x \leftarrow w] \rangle L, \langle \cdot \rangle, \epsilon, \epsilon \rangle \\
\langle \lambda x.r \rangle L, \langle \cdot \rangle w, p, \epsilon \rangle &\quad \triangleright_d B \quad \langle r[x \leftarrow w], L, \epsilon, \epsilon \rangle \\
\lambda x.r, L w, p, \epsilon \rangle &\quad \triangleright_d B \quad r, \langle \lambda x.\langle \cdot \rangle \rangle L w, \epsilon, \epsilon \rangle \\
r, \langle \lambda x.(\cdot) \rangle L w, \epsilon, \epsilon \rangle &\quad \triangleright_d B \quad \langle r[x \leftarrow w], L, \epsilon, \epsilon \rangle
\end{align*}
\]

To close the diagram, we need $\triangleright_d B$ to relate the two bottom states. Note that their relation can be seen as a $\triangleright_d B$ step involving the contexts of the two positions. Therefore we extend the definition of $\triangleright_d B$ to contexts adding the following top level clause (then included in $\triangleright_d B$ via a closure by head contexts):

\[
\langle \lambda x.c \rangle L t \quad \triangleright_d B \quad \langle C[x \leftarrow t] \rangle L
\]

The new clause, in turn, requires an extension of $\triangleright_d B$. Consider:

\[
\begin{align*}
(x, (\lambda x.D_n) u, \epsilon, \Sigma_n) &\quad \triangleright_d B \quad (x, D_n[x \leftarrow u], \epsilon, \Sigma_n) \\
(\lambda x.D_n(x), \langle \cdot \rangle u, (x, (\lambda x.D_n) u, \Sigma_n), \epsilon) &\quad \triangleright_d B \quad (u, (\lambda x.D_n(x)) \langle \cdot \rangle, (x, (\lambda x.D_n) u, \Sigma_n)) \\
(u, (\lambda x.D_n(x)) \langle \cdot \rangle, (x, (\lambda x.D_n) u, \Sigma_n)) &\quad \triangleright_d B \quad (u, D_n(x) [x \leftarrow \langle \cdot \rangle], (x, D_n[x \leftarrow u], \Sigma_n))
\end{align*}
\]

We are then led to add the following clause to $\triangleright_d B$ (closed by head contexts):

\[
\langle \lambda x.t \rangle L C \quad \triangleright_d B \quad \langle t[x \leftarrow C] \rangle L
\]

Note that in the two local bisimulation diagrams the right side is shorter. This is typical of when the machine travels through the redex. Outside of the redex, however, the two sides have the same length, as the next example shows—example that also motivates a further the extension of $\triangleright_d B$ to contexts. Consider the case where $t \triangleright_d B u$ and the diagram is:
We then need to extend \(\rightarrow_{\text{dB}}\) so that \(t \rightarrow_{\text{dB}} u\). A similar situation happens also when entering an ES with transition \(\rightarrow_{\text{var}}\). To close these diagrams, we add two further cases of reduction on contexts. Note that this time they have to be expressed via steps on terms (then included in \(\rightarrow_{\text{dB}}\) via a closure by head contexts), as their direct definition would require contexts with two holes.

\[
\begin{array}{c}
(t, \langle \cdot \rangle r, e, \epsilon) \\
\downarrow \\
(r, t \langle \cdot \rangle, \epsilon, e)
\end{array}
\quad \begin{array}{c}
(u, \langle \cdot \rangle r, e, \epsilon) \\
\downarrow \\
(r, u \langle \cdot \rangle, \epsilon, e)
\end{array}
\]

\[\begin{array}{c}
t \rightarrow_{\text{dB}} u \\
\uparrow \\
tC \rightarrow_{\text{dB}} uC
\end{array}\]

\[\begin{array}{c}
t \rightarrow_{\text{dB}} u \\
\uparrow \\
t[x-C] \rightarrow_{\text{dB}} u[x-C]
\end{array}\]

**Definition 12.** The (overloaded) binary relation \(\triangleright_{\text{dB}}\) between positions, stacks, and states is defined by the following rules.

\[
\begin{align*}
&\frac{t \rightarrow_{\text{dB}} u}{(t, H) \triangleright_{\text{dB}} (u, H)} \quad \text{rd}_{\text{dB}} \\
&\frac{C \rightarrow_{\text{dB}} D}{(t, C) \triangleright_{\text{dB}} (t, D)} \quad \text{ct}_{\text{dB}} \\
&\frac{e \triangleright_{\text{dB}} \epsilon}{(x, C) \triangleright_{\text{dB}} (x, D)} \quad \text{tok}_{1_{\text{dB}}} \\
&\frac{\Xi \triangleright_{\text{dB}} \Pi}{(x, C, \Xi) \triangleright_{\text{dB}} (x, D, \Pi)} \quad \text{sig}_{\text{dB}} \\
&\frac{p \cdot \Gamma \triangleright_{\text{dB}} \epsilon \cdot \Delta}{(x, C) \triangleright_{\text{dB}} (x, D)} \quad \text{tok}_{2_{\text{dB}}} \\
&\frac{e \triangleright_{\text{dB}} e'}{e \cdot \Gamma \triangleright_{\text{dB}} e' \cdot \Delta} \quad \text{tok}_{3_{\text{dB}}} \\
&\frac{(t, C, \Psi, \Sigma) \triangleright_{\text{dB}} (u, D, \Phi, \Theta, p = q)}{(t, C, \Psi, \Sigma, p) \triangleright_{\text{dB}} (u, D, \Phi, \Theta, q)} \quad \text{state}_{\text{dB}}
\end{align*}
\]

Note that \(\triangleright_{\text{dB}}\) contains all pairs \((t, \langle \cdot \rangle, p^k, \epsilon), (u, \langle \cdot \rangle, p^k, \epsilon)\), where \(t \rightarrow_{\text{dB}} u\), i.e. all the initial states containing a dB-redex and its corresponding reduct.

The proof of the next theorem is a tedious easy check of diagrams.

**Theorem 2.** \(\triangleright_{\text{dB}}\) is an improvement between IALM states.

### 7.2 Linear Substitution

Along the lines of the previous sub-section, here we define the candidate improvement \(\triangleright_{1s}\) on states induced by rule \(\rightarrow_{1s}\), and prove it an improvement.

**Steps, Positions, and Context Rewriting.** As for \(\rightarrow_{\text{dB}}\), we are led to extend the rewriting relation to contexts. There are however some new subtleties. Given \(t \rightarrow_{\text{dB}} u\) and a position \(r, C\) for \(t\), for \(\triangleright_{\text{dB}}\) the redex in \(t\) falls always entirely either in \(t\) or \(C\). If \(t \rightarrow_{1s} u\), instead, the redex can be split between the two.

Consider the following diagram (where to simplify we assume the step to be at top level and the stacks to be empty).

\[
\begin{array}{c}
(H(x)[x\rightarrow t], \langle \cdot \rangle, \epsilon, \epsilon) \\
\downarrow \\
(H(x), \langle \cdot \rangle[x\rightarrow t], \epsilon, \epsilon)
\end{array}
\quad \begin{array}{c}
(H(r)[x\rightarrow r], \langle \cdot \rangle, \epsilon, \epsilon) \\
\downarrow \\
(H(r), \langle \cdot \rangle[x\rightarrow r], \epsilon, \epsilon)
\end{array}
\]

To close it, we have to ◁₁₃-relate the two bottom states, where the pattern of the redex/reduct is split between the two parts of the position. This shall motivate clause rdx₂ in the definition of ◁₁₃.

The new rule comes with consequences. Consider the following diagram, where the two starting states are related by the new clause for ◁₁₃:

\[(x, H[x=t], \epsilon, \epsilon) \quad \text{◁₁₃} \quad (t, H[x=t], \epsilon, \epsilon)\]

(2)

To close the diagram, as usual, we have to ◁₁₃-relate them. There are, however, two delicate points. First, we cannot see the context \(H[x]\) as making a \(\rightarrow_\text{ls}\) step towards \(H[x=t]\), because \(t\) does not occur in \(H[x]\). For that, we have to introduce a variant of \(\rightarrow_\text{ls}\) on contexts that is parametric in \(t\) (and more general than the one to deal with the showed simplified diagram):

\[H[x]\] \(\rightarrow_\text{ls}, t\) \(H[C]\]

The second delicate point of diagram (2) is that the extension of ◁₁₃ has to also ◁₁₃-relate signature stacks of different length, namely \(\epsilon\) and \((x, H[x=t], \epsilon)\). This happens because positions of the two states do isolate the same term, but at different depths, as one is in the ES. Then the definition of ◁₁₃ has two new clauses, one for signatures and one for states, to handle such a case. Luckily, the mismatch in length of signature stacks is at most 1.

Last, as for ◁₆₈, we need the following two clauses of reduction on contexts (then included in ◁₁₃ via a closure by head contexts).

\[\frac{t \rightarrow_\text{ls} u}{tC \rightarrow_\text{ls} uC}\]
\[\frac{t \rightarrow_\text{ls} u}{t[x=C] \rightarrow_\text{ls} u[x=C]}\]

Definition 13. Binary relation ◁₁₃ is defined by the following rules.

\[\frac{t \rightarrow_\text{ls} u}{(t, H) \rightarrow_\text{ls} (u, H)}\]
\[\frac{C \rightarrow_\text{ls} D}{(t, C) \rightarrow_\text{ls} (t, D)}\]
\[\frac{K = K'(G[x=t])}{(H(x), K) \rightarrow_\text{ls} (H(t), K)}\]
\[\frac{t \rightarrow_\text{ls} u}{tC \rightarrow_\text{ls} uC}\]
\[\frac{t \rightarrow_\text{ls} u}{t[x=C] \rightarrow_\text{ls} u[x=C]}\]

Theorem 3. ◁₁₃ is an improvement between IALM states.
7.3 Garbage Collection

Steps, Positions, and Context Rewriting. The candidate improvement \( \triangleright_{gc} \) induced by \( \triangleright_{oc} \) requires an extension of \( \triangleright_{oc} \) with a rule on contexts which is similar to the parametric one for \( \triangleright_{ls} \). Let \( t[x \leftarrow u] \triangleright_{gc} t \) and consider:

\[
(t[x \leftarrow u], \langle \cdot \rangle, \epsilon, \epsilon) \xrightarrow{\triangleright_{gc}} (t, \langle \cdot \rangle, \epsilon, \epsilon)
\]

To close the diagram, we extend the definition of \( \rightarrow_{gc} \) to context with the following parametric rule (closed by head contexts):

\[
C[x \leftarrow u] \rightarrow_{gc,t} C \quad \text{if} \quad x \notin \text{fv}(t).
\]

We also need, as for \( \triangleright_{ab} \) and \( \triangleright_{ls} \), the rules (closed by head contexts):

\[
t \rightarrow_{gc} u \quad t \rightarrow_{gc} u
\]

\[
t \triangleright_{oc} uC \quad t[x \leftarrow C] \triangleright_{oc} u[x \leftarrow C]
\]

**Definition 14.** Binary relation \( \triangleright_{gc} \) is defined by the following rules.

\[
\begin{align*}
(t, H) & \triangleright_{gc} (u, H) & \text{rdx}_{gc} \\
C & \triangleright_{gc} D & \text{ctx}_{gc} \\
(t, C) & \triangleright_{gc} (t, D) & \text{ctx2}_{gc} \\
(t, C) & \triangleright_{gc} (t, D) & \text{ctx2}_{gc} \\
\end{align*}
\]

\[
\begin{align*}
\text{tok}_{gc} & \quad \text{tok}_{gc} \\
\text{tok1}_{gc} & \quad \text{tok1}_{gc} \\
\text{tok2}_{gc} & \quad \text{tok2}_{gc} \\
\text{tok3}_{gc} & \quad \text{tok3}_{gc} \\
\end{align*}
\]

\[
(t, C, x, D) & \triangleright_{gc, \Pi} \xi & \text{sig}_{gc} \\
(t, C, x, D) & \triangleright_{gc, \Pi} \xi & \text{sig}_{gc} \\
(t, C, x, D) & \triangleright_{gc, \Pi} \xi & \text{sig}_{gc} \\
(t, C, x, D) & \triangleright_{gc, \Pi} \xi & \text{sig}_{gc} \\
(t, C, x, D) & \triangleright_{gc, \Pi} \xi & \text{sig}_{gc} \\
\end{align*}
\]

\[
\begin{align*}
(t, C) & \triangleright_{gc} (u, D) & \text{p}_{gc, \Phi} \\
(t, C, \Psi, \Sigma, p) & \triangleright_{gc} (u, D, \Phi, \Theta, q) & \text{state}_{gc}
\end{align*}
\]

**Theorem 4.** \( \triangleright_{gc} \) is an improvement between IALM states.

7.4 The Soundness Theorem

Let’s go back to soundness. We have to show the improvements of the previous sections do imply soundness. Consider \( \triangleright = \triangleright_{ab} \cup \triangleright_{ls} \cup \triangleright_{gc} \), that is an improvement because its components are. Consequently, if \( t \rightarrow u \), then the IALM run on \( u \) improves the one on \( t \), that is,

\[
s_{t,k} = (t, \langle \cdot \rangle, p^k, \epsilon) \triangleright (u, \langle \cdot \rangle, p^k, \epsilon) = s_{u,k}.
\]

Improvements transfer more than termination/divergence along linear head evaluation \( \rightarrow \). They also give bisimilar, structurally equivalent IO stacks, proving the invariance of the semantics, that is, soundness—the statement here refines the one at the end of Sect. [x]
Theorem 5 (Soundness, via improvements). If \( t \to u \), then \( \llbracket t \rrbracket_k = \llbracket u \rrbracket_k \) for each \( k \geq 0 \).

Proof. Since \( t \to u \), then \( s = (t, \langle \cdot \rangle, p^k, \epsilon) \) \( \triangleright (u, \langle \cdot \rangle, p^k, \epsilon) = q \) by the results about improvements (Theorem 2, Theorem 3, or Theorem 1). First of all, since improvements transfer termination/divergence (Proposition 1), we have \( \llbracket t \rrbracket_k = \perp \) if and only if \( \llbracket u \rrbracket_k = \perp \). Now, assume that \( \llbracket t \rrbracket_k \neq \perp \). Let us call \( s' = (r, C, \Psi, \Sigma, p) \) the terminal state of \( s \). Since \( \triangleright \) is an improvement, there exists a state \( q' = (r', C', \Psi', \Sigma', p) \) such that \( q \rightarrow^*_{IALM} q' \) and \( s' \triangleright q' \). We consider different cases according to the different possible shapes of \( s' \):

1. \( s' = (\lambda x.w, C, \epsilon, \Sigma) \), i.e. \( r = \lambda x.w, \Psi = \epsilon \) and \( p = \downarrow \). Then, since \( s' \triangleright q' \), \( \Psi' = \epsilon \). Moreover, either \( t \to u \), and thus \( r' = \lambda x.w' \) or \( C \to D \) and thus \( r = r' \). Then, \( \llbracket t \rrbracket_k = \llbracket u \rrbracket_k = \downarrow \).
2. \( s' = (t, \langle \cdot \rangle, p^m \cdot e \cdot p', \epsilon) \). Then, since \( s' \triangleright q' \), \( C' = \langle \cdot \rangle \), because the hole cannot \( \sigma \)-reduce. Moreover, the structure of the IO-stack is preserved by \( \triangleright \) and thus \( \llbracket t \rrbracket_k = \llbracket u \rrbracket_k = (m, l) \).
3. \( s' = (x, C, p^m, \Sigma) \). Since a variable cannot \( \sigma \)-reduce, also \( r' = x \). Then \( \llbracket t \rrbracket_k = \llbracket u \rrbracket_k = x \).

\( \square \)

8 Adequacy

We established soundness with respect to linear head evaluation \( \to \), that is, the semantics \( \llbracket \cdot \rrbracket \) is an invariant of \( \to \). Soundness however is not enough when defining a denotational semantics. A trivial semantics, where every object is mapped on the same element, for example, would be trivially sound.

Here we are going to prove that \( \llbracket \cdot \rrbracket \) is also adequate for \( \to \), i.e. that \( \llbracket t \rrbracket \) reflects some observable aspects of \( t \) and vice versa. For a head strategy in an untyped calculus, one usually observes termination, and, if it holds, the identity of the head variable. And this is exactly what \( \llbracket t \rrbracket \) reflects, or is it adequate for. The statement of the adequacy theorem is the following.

Theorem 6 (Adequacy). Let \( t \) be a LSC term. Then \( t \) has \( \to \)-normal form if and only if there exists \( k > 0 \) such that either \( \llbracket t \rrbracket_k = (m, l) \) for some \( m, l \geq 0 \) or \( \llbracket t \rrbracket_k = x \) for some \( x \in \mathcal{V} \).

Direction IALM to \( \to \). The only if direction of the statement is easy to prove. Since \( \llbracket t \rrbracket_k \) is invariant by \( \to \) (soundness) and \( \to \) terminates on \( t \) we can as well assume that \( t \) is normal. The rest is given by the following proposition.

Proposition 4 (Reading the head variable on \( \to \)-normal forms). Let \( t = \lambda x_0 \ldots \lambda x_n.y_1 \ldots u_1 \) be a head linear normal form up to substitution. If \( y = x_m \) where \( 0 \leq m \leq n \), then \( \llbracket t \rrbracket_{n+1} = (m, l) \), otherwise, if \( y \) is free, then \( \llbracket t \rrbracket_{n+1} = y \).

Proof. We proceed computing \( \llbracket t \rrbracket_{n+1} \) explicitly, according to the definition.

\[
(t, \langle \cdot \rangle, p^{n+1}, \epsilon) \rightarrow^*_{IALM} (y u_1 \ldots u_1, \lambda x_0 \ldots \lambda x_{n-1}, \langle \cdot \rangle, \epsilon, \epsilon) \\
\rightarrow^*_{IALM} (y, \lambda x_0 \ldots \lambda x_n, \langle \cdot \rangle) u_1 \ldots u_1, p', \epsilon)
\]
If \( y \) is free, the IALM stops and \( \llbracket t \rrbracket_{n+1} = y \). Otherwise, if \( y \) is bound by a \( \lambda \)-abstraction, \( i.e. \ y = x_m \) for \( 0 \leq m \leq n \), the computation continues.

\[
(t, \langle \cdot \rangle, p^{n+1}, \epsilon) \xrightarrow{n+1+1_k} IALM(x_m, \lambda x_0 \ldots \lambda x_n, \langle \cdot \rangle u_1 \ldots u_t, p^l, \epsilon) \\
\xrightarrow{IALM} (\lambda x_0 \ldots \lambda x_n x_m u_1 \ldots u_t, \lambda x_0 \ldots \lambda x_{m-1}, \langle \cdot \rangle, e \cdot p^l, \epsilon) \\
\xrightarrow{n_k} IALM(u, \langle \cdot \rangle, p^m \cdot e \cdot p^l, \epsilon).
\]

where \( e = (x_m, \lambda x_0 \ldots \lambda x_n, \langle \cdot \rangle u_1 \ldots u_t, \epsilon) \). Therefore \( \llbracket t \rrbracket_k = \langle m, l \rangle \).

**Direction \( \rightarrow \) to IALM.** The proof of the \( if \) direction of the adequacy theorem is by contra-position, that is, we prove that if the \( \rightarrow \) diverges on \( t \) then no run of the IALM on \( t \) ends in a successful state—careful: the run does not necessarily diverge, we shall come back to this point after the proof.

The proof is obtained via a quantitative analysis of the improvements used to prove soundness, showing that the length of runs strictly decreases along \( \rightarrow \). Note that improvements guarantee that the length of runs does not increase. To prove that it actually decreases one needs an additional **global** analysis of runs—improvements only deal with **local** bisimulation diagrams.

On proof nets, this decreasing property correspond to the standard fact that IAM paths passing through a cut have shorter residuals after that cut.

We recall that we write \( |t|_k \) for the length of the IALM run \( (t, \langle \cdot \rangle, p^k, \epsilon) \), with the convention that \( |t|_k = \infty \) if the machine diverges.

**Lemma 5 (The length of terminating runs strictly decreases along \( \rightarrow \)).**

Let \( t \rightarrow u \) and \( |t|_k \neq \infty \). There exists \( k \geq 0 \) such that \( |t|_h > |u|_h \) for each \( h \geq k \).

**Proof.** We treat the case of \( t \rightarrow_{\text{db}} u \), the others are obtained via similar diagrams.

If \( t \) has a \( \rightarrow_{\text{db}} \)-redex then it has the shape \( t = H(\langle \lambda x.r \rangle Lw) \) and \( u \) in the form \( u = H(\langle r[x-w] \rangle L) \). By induction on the structure of \( H \) one can prove that there exist \( k, n \geq 0 \) such that \( (t, \langle \cdot \rangle, p^k, \epsilon) \rightarrow_{IALM} (\langle \lambda x.r \rangle Lw, H, \epsilon, \epsilon) \) and \( (u, \langle \cdot \rangle, p^k, \epsilon) \rightarrow_{IALM} (\langle r[x-w] \rangle L, H, \epsilon, \epsilon) \). Given such \( n \) and \( k \) by the pumping lemma (Lemma 2) also the following holds: for any \( j \geq 0 \), \( (t, \langle \cdot \rangle, p^j \cdot p^k, \epsilon) \rightarrow_{IALM} (\langle \lambda x.r \rangle Lw, H, p^j, \epsilon) \) and \( (u, \langle \cdot \rangle, p^j \cdot p^k, \epsilon) \rightarrow_{IALM} (\langle r[x-w] \rangle L, H, p^j, \epsilon) \). Moreover, by definition of the improvement \( \text{db} \) we have the following diagram.

\[
\begin{align*}
(H(\langle \lambda x.r \rangle Lw), \langle \cdot \rangle, p^j \cdot p^k, \epsilon) & \xrightarrow{\text{db}} (H(\langle r[x-w] \rangle L), \langle \cdot \rangle, p^j \cdot p^k, \epsilon) \\
\downarrow^n \quad & \downarrow^n \\
(\langle \lambda x.r \rangle Lw, H, p^j, \epsilon) & \xrightarrow{\text{db}} (\langle r[x-w] \rangle L, H, p^j, \epsilon) \\
\downarrow & \downarrow |L| \\
(\lambda x.r, H(\langle \cdot \rangle Lw), p \cdot p^j, \epsilon) & \quad (r[x-w], H(\langle \cdot \rangle L), p^j, \epsilon) \\
\downarrow & \downarrow \\
\quad s_1 = (r, H(\langle \lambda x.\langle \cdot \rangle \rangle Lw), p^j, \epsilon) & \text{db}(r, H(\langle \langle r[x-w] \rangle \rangle L), p^j, \epsilon) = s_2
\end{align*}
\]
From $s_1 \triangleright s_2$, the hypothesis $|t|_k \neq \infty$, and the properties of improvements (Lemma 2), we obtain $|s_1| \geq |s_2|$. Then, by setting $h := k + j$, we have

$$|t|_h = n + 1 + |L| + 1 + |s_1| > n + |L| + 1 + |s_2| = |u|_h. \square$$

Using the lemma, we prove the if direction of adequacy.

**Proposition 5 (→o-divergence implies that the IALM never succeeds).** Let $t$ be a →o-divergent LSC term. There is no $k \geq 0$ such that $|t|_k$ is successful.

**Proof.** By contradiction, suppose that there exists $k$ such that $|t|_k$ is successful. Then by soundness $|t|_k = n \in \mathbb{N}$ and it ends on a successful state. By monotonicity of runs (Lemma 3), $|t|_k = |t|_h = n$ for every $h > k$. Since $t \rightarrow o$-divergent, then there exists an infinite reduction sequence $\rho: t = t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots t_k \rightarrow \cdots$.

Since the length of terminating runs strictly decreases along $\rightarrow o$ for sufficiently long inputs (Lemma 5), for each $i \in \mathbb{N}$ if $t_i \rightarrow t_{i+1}$ then there exists $k_i$ such that $|t_i|_{k_i} > |t_{i+1}|_{k_i}$. Now, consider $h = \max\{k_0, \ldots, k_n, k_{n+1}\}$. We have that $|t_j|_h > |t_{j+1}|_h$ for every $j \in \{0, 1, \ldots, n+1\}$. Then $|t_0|_h \geq |t_{n+1}|_h + n + 1$. Since the length of runs is non-negative, we obtain that $|t_0|_h \geq n + 1$, which is absurd because $h \geq k_0$ and so $|t_0|_h = n$. \square

**Terminating without Succeeding.** The previous proposition leaves open the case where $\rightarrow o$ diverges on $t$ and the runs of the IALM terminate on $t$ without ever succeeding. Such a case can indeed happen. The idea is that the IALM performs a fine analysis of the linear head evaluation of $t$, approximating $\rightarrow o$ while incrementally building the Lévy-Longo tree of $t$. Let us consider the non-terminating term $\Omega = (\lambda x.xx)(\lambda x.xx)$. Starting from $\langle \Omega, (\cdot), p^h, \epsilon \rangle$ the IALM diverges, i.e. $|\Omega|_k = \bot$ for each $k \geq 0$. Instead, if one considers the also non-terminating term $\Lambda = (\lambda x.\lambda y.xx)(\lambda x.\lambda y.xx)$, then the IALM does not diverge and in particular $|\Lambda|_k = \bot$ for each $k \geq 0$—this is exactly a case of perpetual termination without success. This corresponds to the fact that $\Omega \rightarrow_h \Omega$ while $\Lambda \rightarrow_h \lambda x.\Lambda$, i.e. $\Lambda$ has an infinite number of abstractions in its limit normal form, while $\Omega$ has none.

9 Conclusions

**This Work.** We provide an abstract machine based on Geometry of Interaction, that for the first time is defined in a genuinely inductive fashion on $\lambda$-terms, with no reference to proof nets. Soundness of the IALM, our machine, is proved directly by bisimulations, with no reference to game semantics. Soundness holds also for open terms and erasing steps, which is positively unusual.

**What’s Next?** This work paves the way to the study of the (relative) complexity of the IALM and similar token machines, both in time and in space, that we are already pursuing. Notably, the IALM can be optimized as Danos and Regnier do with the JAM, a variant of the IAM introduced in [26]. Our setting easily accommodates the optimization, turning the IALM into the JALM. Moreover, soundness of the JALM can be shown via an improvement of the IALM with a new proof, exploiting the exhaustible state invariant introduced here. This is, however, another story, soon to be told.
References


Appendix

A Proofs of Section 4

Proof of Lemma 1 the proper state invariant Lemma.

Proof. By induction on the execution $s_0 \xrightarrow{k} s_{IALM}$ $s$ from the initial state $s_0$. If $k = 0$, $s = i = (t, (\cdot), p^k, e)$. Clearly $(\cdot)$ is a level 0 context, and $|\Sigma| = 0$. Moreover, $|\Psi| = 0$ and $\downarrow^0 = \downarrow$. Now, let us consider a IAM run of length $k > 0$ and let $\{s_h\}_{0 \leq h \leq k}$ be the sequence of states of this run. By induction hypothesis $s_{k-1} = (t, C_n, \Psi, \Sigma, p)$ is proper i.e $|\Sigma| = n$ and $\downarrow^{|\Psi|} = p$. We can show, by cases, that $s_k$ is also proper.

- $p = \downarrow$.
  - $t = u\cdot r$. Then $s_k = (u, C(\langle \cdot \rangle r), p \cdot \Psi, \Sigma)$. $C(\langle \cdot \rangle r)$ is a context of level $n = |\Sigma|$ and both $|\Psi| = p$ are unchanged.
  - $t = \lambda x.u$ and $\Psi = p \cdot \Psi'$. Then $s_k = (u, C(\lambda x.(\langle \cdot \rangle), \Psi', \Sigma)$. $C(\lambda x.(\cdot))$ is a context of level $n = |\Sigma|$ and both $|\Psi| = p$ are unchanged.

- $t = \lambda x.u$ and $\Psi = (x, C(\lambda x.D_m), \Theta) \cdot \Psi'$. Then $s_k = (x, C(\lambda x.D_m), \Psi', \Theta \cdot \Sigma)$. $C(\lambda x.D_m)$ is a context of level $n + m = |\Sigma| + |\Theta|$ and since $\downarrow^{|\Psi|} = \downarrow$, then $\downarrow^{|\Psi'|} = \downarrow^{|\Psi'| + 1} = \uparrow$.

- $t = D_i \cdot \Sigma_l$. Then $s_k = (x, C_m(D_i \cdot \langle \cdot \rangle), x, C_m(\lambda x.D_i, \Sigma_l) \cdot \Psi, \Sigma')$. Since $m + l = |\Sigma|$, then $|\Sigma'| = m$ and since $\downarrow^{|\Psi|} = \downarrow$, then $\downarrow^{|\Psi| + 1} = \uparrow$.

- $t = \lambda x.u \cdot r$. Then $s_k = (u, C(\langle \cdot \rangle x \leftarrow r), \Psi, \Sigma)$. $C(\langle \cdot \rangle x \leftarrow r)$ is context of level $n = |\Sigma|$ and both $|\Psi| = p$ are unchanged.

- $p = \uparrow$. The proof is equivalent to the one above.

Proof of Lemma 2 the Pumping Lemma.

Proof. We proceed by induction on $n$. Thus we have that if $(t, C, e, \Sigma, p) \xrightarrow{n-1} s_{IALM}$ $(u, D, \Psi, \Theta, q)$, then $(t, C, \Gamma, \Sigma, p) \xrightarrow{n-1} s_{IALM}$ $(u, D, \Psi \cdot \Gamma, \Theta, q)$. The proof now proceeds analyzing all possible transitions from $(u, D, \Psi, \Theta, q)$ and $(u, D, \Psi \cdot \Gamma, \Theta, q)$. The key point is that every transition of the IALM consumes at most 1 element of the IO-stack. This is why the pushed stack $\Gamma$ gets never touched.

Proof of Lemma 3 the Monotonicity Lemma.
Proof. Let \( s_{t,k} \rightarrow^{n}_{IALM} (u,C,\Psi,\Sigma,p) = s'_{t,k} \). By the pumping lemma (Lemma 2), if \( s_{t,k+1} \rightarrow^{n}_{IALM} (u,C,\Psi \cdot p,\Sigma,p) = s''_{t,k+1} \). If \(|t|_{k} = \infty\) then \( s_{t,k} \rightarrow^{n}_{IALM} s''_{t,k+1} \) for every \( n \in \mathbb{N} \) and so \( s_{t,k+1} \rightarrow^{n}_{IALM} s''_{t,k+1} \), that is, \(|t|_{k+1} = \infty = |t|_{k}\).

If \(|t|_{k} = n \in \mathbb{N}\) then \( s''_{t,k} \) is final. Two cases. If \( s''_{t,k} \) is an approximating final state \((\lambda x.t,C,\epsilon,\Sigma)\) then \( s''_{t,k+1} = (\lambda x.t,C,p,\Sigma)\) which can make a transition, that is, \(|t|_{k} < |t|_{k+1}\). If instead \( s''_{t,k} \) is a bound successful final state \((t,\langle\cdot\rangle,p^{m} \cdot e \cdot p^{n},\Sigma)\) then \( s''_{t,k+1} = (t,\langle\cdot\rangle,p^{m} \cdot e \cdot p^{n+1},\Sigma)\) which is also a successful final state, and \(|t|_{k} = |t|_{k+1}\). Similarly for an open successful final state. A straightforward induction then shows that the same holds for every other \( h > k \).

\( \square \)

B Proofs of Section 5

We need a further Lemma in order to prove Lemma 4.

Lemma 6. Let \( s = (t,C_{n},\Psi,\Sigma_{n},p) \) be a balanced state. Then:
1. Polarity: the dual \((t,C_{n},\Psi,\Sigma_{n},p)\) of \( s \) induces the same outer states;
2. Input-Output stack: the state \((t,C_{n},\Phi,\Sigma_{n},p)\) obtained from \( s \) replacing \( \Psi \) with an arbitrary IO stack \( \Phi \) induces the same outer states;
3. Head translation: if \( t = H(r) \) then the head translation \((r,C_{n}(H),\Phi,\Sigma_{n},p)\) of \( s \) induces the same outer states.
4. Inclusion: if \( C_{n} = C_{n}(C_{i}) \) and \( \Sigma_{n} = \Sigma_{i} \cdot \Sigma_{m} \) then the outer states of \((C_{i}(t),C_{m},\Phi,\Sigma_{m},p)\) are outer states of \( s \).

Proof. Immediate consequences of the definition of outer state. \( \square \)

Proof of Lemma 4 the exhaustible invariant.

Proof. Let \( s = (t,\langle\cdot\rangle,p^{h},\epsilon) \). By induction on \( k \). For \( k = 0 \) there is nothing to prove because the IO stack has no signatures (so it does not decompose) and \( s \) has no outer state. Then suppose \( s \rightarrow^{k-1}_{IALM} s'' \rightarrow^{k}_{IALM} s' \). By i.h., \( s'' = (u,C,\Psi,\Sigma,p) \) is exhaustible, and with this hypothesis we need to conclude that \( s' \) is exhaustible, too. There are many cases to take into account, depending on the transition used to move from \( s'' \) to \( s' \). First, suppose that \( p = \downarrow \). Cases of \( s'' \rightarrow^{k}_{IALM} s' \):

1. Application, i.e. \( u = rw \) and

\((rw,C,\Psi,\Sigma) \rightarrow^{1}_{IALM} (r,C(\langle\cdot\rangle w),p \cdot \Psi,\Sigma)\).

We have to show that the obtained state \( s' \) is exhaustible. For outer decomposition, it follows from Lemma 6 and the i.h.: \( s'' \) is a head translation of \( s' \), and the lemma states that they have the same outer states, which are exhaustible because \( s'' \) is exhaustible by i.h.

For IO decomposition, consider a decomposition \( \Psi = \Phi \cdot e \cdot \Phi' \). Two cases, depending on the parity of \(|\Phi|_{e} |\):

1. \(|\Phi|_{e} | \) is odd. Then the exponential length of the IO stack \( p \cdot \Phi \cdot e \) is even (multiplicative constants are ignored) and so the polarity of the corresponding IO-state \( s'_{e} \) is \( \uparrow \). Note that \( s'_{e} \) reduces to a IO-state \( s''_{e} \) for \( s'' \):

\[ s'_{e} = (r,C(\langle\cdot\rangle w),p \cdot \Phi \cdot e \cdot \Sigma) \rightarrow^{1}_{IALM} (rw,C,\Phi \cdot e,\Sigma) = s''_{e} \]
By i.h., $s''$ is exhaustible, and so $s''_e$ evolves to an exhaustible state generated by $e$, call it $q_e$. We conclude by observing that $s'_e$ evolves to $q_e$.

2. $\| \Phi \|_e$ is even. Then $[p \cdot \Phi \cdot e|_e$ is odd, and the polarity of the corresponding IO-state $s'_e$ os $s'$ is $\downarrow$. Note that the corresponding IO-state $s''_e$ of $s''$ reduces to $s'_e$:

$$s''_e = (rw, C, \Phi' \cdot e, \Sigma) \rightarrow_{\uparrow \uparrow} (r, C(\langle \cdot \rangle w), p \cdot \Phi \cdot e, \Sigma) = s'_e$$

By i.h., $s''$ is exhaustible, then $s''_e$ evolves to an exhaustible state generated by $e$, call it $q_e$. The IAM is deterministic, so $s'_e$ itself reduces to $q_e$, which proves IO decomposition.

3. Variable bound by an abstraction, i.e. $u = x$ and

$$s'' = (x, C(\lambda x.D_n), C, (x, C(\lambda x.D_n), \Sigma_n) \cdot \Phi, \Sigma) = s'$$

The proof that $s'$ is exhaustible is divided in two parts:

1. Outer decomposition. By Lemma [6], all outer states of $s'$ are also outer-states of $s''$. Since the latter is exhaustible by i.h., then outer decomposition holds for $s'$, too.

2. IO decomposition. We need to consider various cases, corresponding to the various decompositions of the IO stack $e' \cdot \Phi$ where $e' = (x, C(\lambda x.D_n), \Sigma_n)$:

   1. The signature to test is $e = e'$, i.e. the first one. We are then considering a prefix of odd length of $e' \cdot \Phi$, so the polarity of the corresponding IO-state $s'_e$ is $\downarrow$. Observe, however, that by definition

   $$s'_e = (\lambda x.D_n(x), x, C(\lambda x.D_n), \Sigma_n, \Sigma) \rightarrow_{\downarrow \downarrow} (x, C(\lambda x.D_n), \epsilon, \Sigma_n, \Sigma) = s''_e$$

   where $s''_e$ is trivially generated by $e$. Moreover, by i.h., $s''$ is exhaustible, a property which is easily transferred to $s''_e$: the outer states are the same by Lemma [6], while $s''_e$ satisfies IO decomposition trivially, because the IO stack is empty.

2. The prefix $\Phi \cdot e$ of the IO stack has even length and the polarity of the corresponding IO state $s'_e$ is $\uparrow$. Let $\Phi = (x, C(\lambda x.D_n), \Sigma_n) \cdot \Phi'$. Note that the corresponding IO-state $s''_e$ of $s''$ reduces to $s'_e$:

   $$s''_e = (x, C(\lambda x.D_n), \Phi' \cdot e, \Sigma_n, \Sigma) \rightarrow_{\downarrow \downarrow} (x, C(\lambda x.D_n), \Phi' \cdot e, \Sigma) = s'_e$$

   By i.h., $s''$ is exhaustible, then $s''_e$ evolves to an exhaustible state generated by $e$, call it $q_e$. The IAM is deterministic, so $s'_e$ itself reduces to $q_e$, which proves IO decomposition.

3. The prefix $\Phi \cdot e$ of the IO stack has odd strictly positive length and the polarity of the corresponding outer state $s'_e$ is $\downarrow$. Let $\Phi = (x, C(\lambda x.D_n), \Sigma_n) \cdot \Phi'$. Note that $s'_e$ reduces to the corresponding outer state $s''_e$ of $s''$:

   $$s'_e = (\lambda x.D_n(x), x, C(\lambda x.D_n), \Sigma_n) \cdot \Phi' \cdot e, \Sigma) \rightarrow_{\downarrow \downarrow} (x, C(\lambda x.D_n), \Phi' \cdot e, \Sigma) = s''_e$$

   We can then proceed as usual using the i.h.
4. Abstraction, i.e. \( u = \lambda x \cdot r \) and
\[
s'' = (\lambda x \cdot r, C, (x, C(\lambda x.D_n), \Theta) \cdot \Psi, \Sigma) \rightarrow_{\Delta 2} (x, C(\lambda x.D_n), \Psi, \Theta \cdot \Sigma) = s'
\]

1. **Outer Decomposition.** Let \( e = (x, C(\lambda x.D_n), \Theta) \) and note that
\[
s''_e = (\lambda x.\cdot r, C, (x, C(\lambda x.D_n), \Theta), \Sigma) \rightarrow_{\Delta 2} (x, C(\lambda x.D_n), e, \Theta \cdot \Sigma) = s'_e
\]
By i.h., \( s'' \) is exhaustible and so by IO decomposition \( s'_e \) is exhaustible. By Lemma 3.2, \( s'_e \) and \( s' \) have the same outer states, so outer decomposition for \( s' \) holds because it does for \( s'_e \).

2. **IO decomposition.** As usual, we have to consider various cases, corresponding to the possible decompositions \( \Psi = \Phi \cdot e \cdot \Phi' \) of the IO stack.
   1. \( \Phi_e \) is odd, so that the prefix \( \Phi \cdot e \) of the IO stack has even length and the polarity of the IO-state \( s'_e \) corresponding to \( e \) is \( \uparrow \). Note that the IO-state \( s''_e \) of \( s'' \) reduces to the corresponding IO-state \( s'_e \) of \( s' \):
      \[
s''_e = (\lambda x.D_n(x), C, (x, C(\lambda x.D_n), \Theta) \cdot \Phi.e, \Sigma) \rightarrow_{\Delta 2} (x, C(\lambda x.D_n), \Phi.e, \Theta \cdot \Sigma) = s'_e
\]
      We can then proceed as usual, exploiting the determinism of the IALM and the i.h.
   2. \( \Phi_e \neq 0 \) is even, so that the prefix \( \Phi \cdot e \) of the IO stack has odd length and the polarity of the IO-state \( s'_e \) corresponding to \( e \) is \( \downarrow \). Note that \( s'_e \) reduces to the corresponding IO-state \( s''_e \) of \( s'' \):
      \[
s'_e = (x, C(\lambda x.D_n), \Phi.e, \Theta \cdot \Sigma) \rightarrow_{\Delta \varnothing \cdot \lambda} (\lambda x.D_n(x), C, (x, C(\lambda x.D_n), \Theta) \cdot \Phi.e, \Sigma) = s''_e
\]
      Again, we can then proceed as usual using the i.h.

3. **Explicit Substitution**, i.e. \( u = r[x\leftarrow w] \) and
\[
s'' = (r[x\leftarrow w], C, \Psi, \Sigma) \rightarrow_{\Delta x e s} (r, C(\langle \cdot \rangle[x\leftarrow w]), \Psi, \Sigma) = s'
\]
For outer decomposition, it follows from Lemma 3.3, and the i.h.: \( s'' \) is a head translation of \( s' \), and the lemma states that they have the same outer states, which are exhaustible because \( s'' \) is exhaustible by i.h.
For IO decomposition it goes exactly as the application case. We spell it out anyway. Consider a decomposition \( \Psi = \Phi \cdot e \cdot \Phi' \). Two cases, depending on the parity of \( \Phi_e \):
   1. \( \Phi_e \) is odd. Then the exponential length of the IO stack \( \Phi \cdot e \) is even and so the polarity of the corresponding IO-state \( s'_e \) is \( \uparrow \). Note that \( s'_e \) reduces to a IO-state \( s''_e \) for \( s'' \):
      \[
s'_e = (r, C(\langle \cdot \rangle[x\leftarrow w]), \Phi \cdot e, \Sigma) \rightarrow_{\Delta x e s 1} (r[x\leftarrow w], C, \Phi \cdot e, \Sigma) = s''_e
\]
      Again, we then proceed as usual using the i.h.
   2. \( \Phi_e \) is even. Then \( \Phi \cdot e \) is odd, and the polarity of the corresponding IO-state \( s'_e \) is \( \downarrow \). Note that the corresponding IO-state \( s''_e \) of \( s'' \) reduces to \( s'_e \):
      \[
s''_e = (r[x\leftarrow w], C, \Phi \cdot e, \Sigma) \rightarrow_{\Delta x e s} (r, C(\langle \cdot \rangle[x\leftarrow w]), \Phi \cdot e, \Sigma) = s'_e
\]
Again, we then proceed as usual, exploiting the determinism of the IALM and the \textit{i.h.}

3. \textit{Variable bound by an explicit substitution}, i.e. \( u = x \) and

\[
s'' = (x, C(D_n[x\leftarrow r]), \Psi, \Sigma_n \cdot \Sigma, \varnothing) \rightarrow_{\text{var}} (r, C(D_n(x)[x\leftarrow \cdot]), \Psi, (x, C(D_n[x\leftarrow r]), \Sigma_n \cdot \Sigma)) = s'
\]

1. \textit{Outer decomposition}: let \( e := (x, C(D_n[x\leftarrow r]), \Sigma) \) and \( m = |e \cdot \Sigma| \). The \( m \)-outer state of \( s' \) is

\[
\text{out}_m(s) = (r, C(D_n(x)[x\leftarrow \cdot]), \epsilon, (x, C(D_n[x\leftarrow r]), \Sigma_n \cdot \Sigma))
\]

which makes a transition

\[
\rightarrow_{\text{en2}} (x, C(D_n[x\leftarrow r]), \epsilon, \Sigma_n \cdot \Sigma) = (s''')^\perp
\]

that is a state generated by \( e \), as required by outer decomposition. We have to prove that \((s''')^\perp\) is exhaustible. \textit{IO decomposition} is trivial, because the IO stack is empty. Outer decomposition follows from the \textit{i.h.} and the fact that \((s''')^\perp\) is \( s'' \) with reversed polarity and without the IO stack, and so by Lemma \textbf{6.1} and Lemma \textbf{6.2} they have the same outer states.

Note that the \( i \)-outer states of \( s' \) for \( i < m \) are the \( i \)-outer states of \( s' \), and so they satisfy the outer decomposition clause by the \textit{i.h.}

2. \textit{IO decomposition}: it goes exactly as in the previous ordinary cases. We spell it out anyway. Consider a decomposition \( \Psi = \Phi \cdot e \cdot \Phi' \). Two cases, depending on the parity of \(|\Phi|_e\):

1. \( |\Phi|_e \) is odd. Then the exponential length of the IO stack \( \Phi \cdot e \) is even and so the polarity of the corresponding IO-state \( s'_e \) is \( \uparrow \). Note that \( s'_e \) reduces to a IO-state \( s''_e \) for \( s'' \):

\[
s'_e = (r, C(D_n(x)[x\leftarrow \cdot]), \Phi \cdot e, (x, C(D_n[x\leftarrow r]), \Sigma_n \cdot \Sigma)) \rightarrow_{\text{en2}} (x, C(D_n[x\leftarrow r]), \Phi \cdot e, \Sigma_n \cdot \Sigma) = s''_e
\]

Again, we then proceed as usual using the \textit{i.h.}

2. \( |\Phi|_e \) is even. Then \(|\Phi \cdot e|_e|_e\) is odd, and the polarity of the corresponding IO-state \( s'_e \) of \( s' \) is \( \downarrow \). Note that the corresponding IO-state \( s''_e \) of \( s'' \) reduces to \( s'_e \):

\[
s''_e = (x, C(D_n[x\leftarrow r]), \Phi \cdot e, \Sigma_n \cdot \Sigma) \rightarrow_{\text{var}} (r, C(D_n(x)[x\leftarrow \cdot]), \Phi \cdot e, (x, C(D_n[x\leftarrow r]), \Sigma_n \cdot \Sigma)) = s'_e
\]

Again, we then proceed as usual, exploiting the determinism of the IALM and the \textit{i.h.}

Now, suppose that \( p = \uparrow \). Cases of \( s'' \rightarrow_{\text{IALM}} s' \):

1. \textit{Coming from the left of an application}, i.e. \( C = D(\cdot) \) and

\[
s'' = (u, D(\cdot)(r), e \cdot \Psi, \Sigma, \varnothing) \rightarrow_{\text{en1}} (r, D(u(\cdot)), \Psi, e \cdot \Sigma) = s'.
\]

The proof that \( s' \) is exhaustible is divided in two parts:
1. **Outer decomposition.** The outer states of $s'$ are those of $s''$ plus $(r, D(u⟨⟩), e, e·Σ)$. The former are fine because of the i.h., while about the latter, observe that $(r, D(u⟨⟩), e, e·Σ)$ evolves to $(u, D⟨⟨·⟩⟩r, e, Σ)$ which is an IO-state of $s''$. The thesis easily follows.

2. **IO decomposition.** Let $Φ$ be a prefix of $Ψ$ such that $Φ = Φ'·e'$. Two cases:
   1. $|Φ|_e$ is odd, and the polarity is ↓. Note that the IO-state $s''$ of $s''$ corresponding to $e$ reduces to an IO-state $s'_e$ of $s'$: $s'' = (r, D(u⟨⟩), e·Φ, Σ) \overset{↓}{\Rightarrow} (r, D⟨⟨·⟩⟩u, e·Φ, Σ) = s'_e$

      We can then proceed as usual, using the i.h. and determinism of the IAM.
   2. $|Φ|_e$ is even, and the polarity is ↑. Note that $s'_e$ reduces to the corresponding IO-state $s''_e$ of $s'':$ $s'_e = (r, D(u⟨⟩), Φ, e·Σ) \overset{↑}{\Rightarrow} (r, D⟨⟨·⟩⟩u, e·Φ, Σ) = s''_e$

      Again, we can proceed as usual, using the i.h.

3. **Coming from the right of an application,** i.e. $C = D⟨⟨·⟩⟩r$ and

   $s'' = (u, D⟨⟨r⟩⟩, Ψ, e·Σ) \overset{↑}{\Rightarrow} (r, D⟨⟨·⟩⟩u, e·Ψ, Σ) = s'$

The proof that $s'$ is exhaustible is divided in two parts:

1. **Outer decomposition:** the outer states of $s'$ are among the outer states of $s''$, so outer decomposition follows from i.h.

2. **IO decomposition.** Let $Φ$ be a prefix of $Ψ$. Two cases:
   1. $Φ = Ψ$ is empty. So that the IO stack contains only $e$, its length is odd, and the polarity is ↓. The state to be proven exhaustible is $s'_e = (r, D⟨⟨·⟩⟩u, e, Σ)$

   Now, note that the outer state $\text{out}_{|e·Σ|}(s'')$ of $s''$ reduces in one step to $s'_e$:

   $\text{out}_{|e·Σ|}(s'') = (u, D⟨⟨r⟩⟩, e·e·Σ) \overset{↑}{\Rightarrow} (r, D⟨⟨·⟩⟩u, e·Σ)$

   By outer decomposition for $s''$, there is a state $q_e$ generated by $e$ such that $\text{out}_{|e·Σ|}(s'') \overset{*}{\Rightarrow}^{IAM} q_e$. By determinism of the IAM, $s'_e \overset{*}{\Rightarrow}^{IAM} q_e$.

   2. $Φ ≠ Ψ$ is non-empty. Then $Φ = Φ'·e'$ Two cases:
      1. $|Φ'·e'|_e$ is odd, so that the IO stack $e·Φ'·e'$ has odd length and the polarity is ↓. Note that the IO-state $s''_e$ corresponding to $e'$ of $s''$ reduces to the IO-state $s''_e$ corresponding to $e'$ of $s'$: $s''_e = (r, D⟨⟨·⟩⟩u, Φ'·e', e·Σ) \overset{↑}{\Rightarrow} (r, D⟨⟨·⟩⟩u, e·Φ'·e', Σ) = s''_e$

         In this case, as usual, we can conclude by determinism of the IAM.
2. \( |Φ|_e \) is odd, so that the IO stack \( e \cdot Φ' \cdot e' \) has even length and the polarity is \( \uparrow \). Note that \( s'_{e'} \) reduces to the corresponding IO-state \( s''_{e'} \): 

\[
s'_{e'} = (r, D(\langle \cdot \rangle u), e \cdot Φ' \cdot e', \Sigma) \rightarrow_{\uparrow \Theta \Pi 1} (r, D(\langle \cdot \rangle u), Φ' \cdot e', e \cdot \Sigma) = s''_{e'}
\]

Again, the usual scheme allows us to conclude that IO decomposition holds.

3. Explicit Substitution

\[
s'' = (u, C(\langle \cdot \rangle [x \leftarrow r]), \Psi, \Sigma) \rightarrow_{\uparrow \Theta \Pi 1} (u[x \leftarrow r], C, \Psi, \Sigma) = s'
\]

1. Outer decomposition: by Lemma \( \ref{lemma:head_translation} \) (head translation), the outer states of \( s' \) are outer states of \( s'' \), which satisfy outer decomposition by i.h.

2. IO decomposition: it goes exactly as for the other ordinary cases (i.h., plus determinism in one of the two sub-cases).

3. Coming from inside an explicit substitution:

\[
s'' = (u, C(D(x)[x \leftarrow \cdot]), \Psi, (x, C(D[x \leftarrow u]), \Theta) \cdot \Sigma) \rightarrow_{\uparrow \Theta \Pi 2} (x, C(D[x \leftarrow u]), (x, C(D[x \leftarrow u]), \Theta) \cdot \Sigma) = s'
\]

1. Outer decomposition: by i.h., \( s'' \) is exhaustible, and its \( |\Sigma| + 1 \)-outer state evolves to

\[
\text{out}_{|\Sigma|+1}(s'') = (u, C(D(x)[x \leftarrow \cdot]), (x, C(D[x \leftarrow u]), \Theta) \cdot \Sigma) \rightarrow_{\uparrow \Theta \Pi 2} (x, C(D[x \leftarrow u]), (x, C(D[x \leftarrow u]), \Theta) \cdot \Sigma) = s'_{e'}
\]

which is exhaustible. By Lemma \( \ref{lemma:head_translation} \), \( s'_{e'} \) and \( s' \) have the same outer states, that then verify the outer decomposition clause.

2. IO decomposition: since the IO stack is unaffected by the transition, this case goes exactly as the other ordinary ones.

C Proofs of Section \( \ref{section:proofs} \)

Proof of Proposition \( \ref{prop:improvements} \) about improvements.

Proof.

1. \( \Rightarrow \). Let us suppose \( s \in S_1 \) and let \( n \) be the number of steps that \( s \) needs to terminate. We proceed by induction on \( n \). If \( n = 0 \), \( s \in F_S \) and since \( sRq \), \( q \in F_Q \) and thus \( q \in Q_1 \). If \( n = h > 0 \), then \( s \rightarrow s' \), and thus there exists \( s'', q', k, j \) such that \( q \rightarrow^k q' \), \( s' \rightarrow^j s'' \), \( s''Rq' \) and \( k \leq j + 1 \). Since \( s'' \) terminates in less than \( h - 1 \) steps, by induction hypothesis \( q' \in Q_1 \) and thus also \( q \in Q_1 \).

\( \Leftarrow \). Let us suppose \( q \in Q_1 \) and let \( n \) be the number of steps that \( q \) needs to terminate. We proceed by induction on \( n \). If \( n = 0 \), \( q \in F_Q \) and since \( sRq \), \( s \in S_1 \). If \( n = h > 0 \), then \( q \rightarrow q' \), and thus there exists \( s', q'', k, j \) such that \( s \rightarrow^k s' \), \( q' \rightarrow^j q'' \), \( s''Rq'' \) and \( k \geq j + 1 \). Since \( q'' \) terminates in less than \( h \) steps, by induction hypothesis \( s'' \in S_1 \) and thus also \( s \in S_1 \).
2. If $s \in S_{\uparrow}$ and $q \in Q_{\uparrow}$, then $|s| = |q| = \infty$. Let us consider the other case, i.e. when $s \in S_{\downarrow}$ and $q \in Q_{\downarrow}$. We proceed by induction on $|s|$. If $|s| = 0$, then $q \in F_{Q}$ and thus also $|q| = 0$. If $|s| = n > 0$, then $s \rightarrow s'$ and there exists $s'', q', m, l$ such that $q \rightarrow^m q', s' \rightarrow^l s'', s'' \rightarrow q'$ and $m \leq l + 1$. By i.h., $|s''| \geq |q'|$. Thus, since $m \leq l + 1$, then $|s| = |s''| + l + 1 \geq |q'| + m = |q|$. $\square$