Intuitionistic Light Affine Logic

ANDREA ASPERTI
Università di Bologna
and
LUCA ROVERSI
Università di Torino

This article is a structured introduction to Intuitionistic Light Affine Logic (**ILAL**). **ILAL** has a polynomially costing normalization, and it is expressive enough to encode, and simulate, all **PolyTime** Turing machines. The bound on the normalization cost is proved by introducing the proof-nets for **ILAL**. The bound follows from a suitable normalization strategy that exploits structural properties of the proof-nets. This allows us to have a good understanding of the meaning of the § modality, which is a peculiarity of light logics. The expressive power of **ILAL** is demonstrated in full detail. Such a proof gives a hint of the nontrivial task of programming with resource limitations, using **ILAL** derivations as programs.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*proof theory*

General Terms: Languages, Theory

1. INTRODUCTION

This article deals with *implicit polytime computational systems* [Girard et al. 1998; Leivant and Marion 1993; Leivant 1994; Girard 1998]. The purpose of such systems is many-fold . On the theoretical side, they provide a better understanding about the *logical essence* of calculating with time restrictions. The systems that admit a Curry-Howard correspondence [Girard et al. 1989] yield sophisticated typing systems that, *statically*, provide an accurate upper bound on the complexity of the computation. The types give essential information on the strategy needed to reduce the terms they type efficiently.

A leading paper in this area is Girard's Light Linear Logic [Girard 1998] (**LLL**), a deductive system with cut elimination, that is, a logical system. In

Author's address: A. Asperti, Dipartmento di Scienze dell'Informazione, Via di Mura Anteo Zamboni, n. 7, 40127 Bolorigna, Italy, e-mail: asperti@cs.unibo.it, Web: http://www.cs.unibo.it/~asperti; L. Roversi, Dipartmento di Informatica, C.so Svizzera, n. 185, 10149 Torino, Italy, e-mail: rover@di.unito.it, Web: http://www.di.unito.it/~rover.

Asperti [1998], Light Affine Logic (**LAL**), a slight variation of **LLL**, was introduced. Roversi [1999] made some basic observations about how to build a proof of the representation power of **LAL**, and, indirectly, of **LLL** as well. This article is a monolithic reworking of both papers with the hope to make the subject more widely accessible. The article is designed for people acquainted with basics of Linear Logic [Girard 1995].

The main results of this article are two theorems about Intuitionistic Light Affine Logic (**ILAL**).

THEOREM 1 (COMPLEXITY BOUND OF **ILAL**). *There exists a function* $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *and a reduction strategy* $\triangleright_\rho$ *for a set of basic reduction steps such that, for every proof-net* Π, $\triangleright_\rho$ *reduces* Π *to its normal form in a number of elementary reduction steps bounded by* $f(\mathbf{D}(\Pi), \partial(\Pi))$, *being* $\mathbf{D}(\Pi)$ *the dimension of* Π, *and* $\partial(\Pi)$ *the* depth *of* Π. *Moreover,* $f(\mathbf{D}(\Pi), \partial(\Pi))$ *is a polynomial in* $\mathbf{D}(\Pi)$, *whose exponent is only function of* $\partial(\Pi)$.

THEOREM 2 (REPRESENTATION POWER). *There exists a translation* $\hat{}$ *from every* **PolyTime** *Turing machine* T *to a proof-net* $\hat{T}$ *of* **ILAL**, *so that* $\hat{T}$ *simulates* T. *The* depth $\partial(\hat{T})$ *of* $\hat{T}$ *is a function of the degree of the polynomial that bounds the length of the computations of* T, *and never depends on the dimension of the input.*

In more detail, **LAL** is introduced by adding full weakening to **LLL**. This modification both preserves the good complexity property that **LAL** inherits from **LLL**, and greatly simplifies **LLL** itself. Indeed, the number of rules decreases from 21 to just 11 rules, and **LAL** is endowed with additives, without adding them explicitly: "free weakening" sounds like "free additive computational behavior." Rephrasing Girard [1998], the slogan behind the design of **LAL** is: *the abuse of contraction may have damaging complexity effects, but the abstinence from weakening leads to inessential syntactical complications*.

1.1 Layout of Article

Section 2 recalls the single-side sequent calculus of **LAL**, as introduced in Asperti [1998], with some informal descriptions about its main principles. Section 3 restricts the sequent calculus of Section 2 to its intuitionistic form, so introducing **ILAL**. The sequent calculus of this section should be looked at as a traditional tool to speak about **ILAL**, but it is not our favorite language to prove the main properties of **ILAL** itself. To that purpose, we prefer proof-nets, as defined in Section 4, on which we shall develop computations in terms of elementary rewriting steps and garbage rewriting steps, as described in Section 5. Section 6 shows that the proof-nets of **ILAL**, endowed with the above rewriting steps, are a good programming language, that is, they are closed under the normalization, they are confluent, and their normalization has the expected polynomially bounded cost.

Our next step is to prove that **ILAL** is expressive enough to represent all the polynomially computable functions. As anticipated in the theorem above, this is accomplished by showing the existence of an encoding from the set of **PolyTime** Turing machines to **ILAL**. Of course, it would be possible to write the whole

$$\mathcal{T} = \{\alpha, \beta, \gamma, \ldots, \alpha^{\perp}, \beta^{\perp}, \gamma^{\perp}, \ldots\}$$

Fig. 1. Literals of **LAL**.

$$A ::= \mathcal{T} \mid A \otimes A \mid A \wp A \mid \forall \alpha.A \mid \exists \alpha.A \mid !A \mid ?A \mid \S A$$

Fig. 2. Formulas of **LAL**.

$$\begin{aligned}
(\alpha^{\perp})^{\perp} &= \alpha \\
(!A)^{\perp} &= ?(A^{\perp}) \\
(\S A)^{\perp} &= \S(A^{\perp}) \\
(?A)^{\perp} &= !(A^{\perp}) \\
(A \otimes B)^{\perp} &= A^{\perp} \wp B^{\perp} \\
(A \wp B)^{\perp} &= A^{\perp} \otimes B^{\perp} \\
(\forall \alpha.A)^{\perp} &= \exists \alpha.A^{\perp} \\
(\exists \alpha.A)^{\perp} &= \forall \alpha.A^{\perp}
\end{aligned}$$

Fig. 3. "De Morgan" laws on the formulas.

encoding, using the proof-nets. However, we found that this approach could not be proposed because the proof-nets hide their sequences of construction, making it more difficult to read out the programs they represent. So, Section 7 introduces an intuitionistic double-side sequent calculus, equivalent to the single-side one of **ILAL**, of Section 3. Then, the double-side sequent calculus is encoded by a functional language, which is an extended λ-Calculus, whose terms are typed by the formulas of **ILAL**. Section 8 is a first programming example with the functional notation. We develop a numerical system with a predecessor that is syntactically linear, up to weakening, and that obeys a general programming scheme, which we shall heavily exploit to encode the whole class of **PolyTime** Turing machines as well. Section 9 contains a second programming example. For the first time, we write all the details to encode the polynomials with positive degree and positive coefficients as derivations of **ILAL**. Section 10 shows the expressive power of **ILAL**. The proof is a further programming exercise. It consists of the definition of a translation from **PolyTime** Turing machines to terms of our functional language. For a simpler encoding, we make some simplifying, but not restricting assumptions, on the class of **PolyTime** Turing machine effectively encoded. Section 11 concludes the paper with some observations and hypothesis on future work.

2. LIGHT AFFINE LOGIC

Light Affine Logic (**LAL**) is both a variant and a simplification of Light Linear Logic (**LLL**). We recall it here below. Let $\mathcal{T}$ be a denumerable set of propositional variables, called *literals*, as in Figure 1. The literals with form $\alpha^{\perp}$, for some α, are *negative*. All the others are *positive*. The set $\mathcal{F}$ of formulas of **LAL** is the language, generated by the grammar in Figure 2, and partitioned by the equations in Figure 3.

The sequent calculus of **LAL** is in Figure 4. The judgments have form $\vdash \Gamma$, where Γ (and also Δ) denotes a, *possibly empty*, multi-set of $\mathcal{F}$, and $!\Gamma$ ($\S\Gamma$) denotes the distribution of ! ($\S$) along all the components of Γ.

$$(ax)\ \frac{}{\vdash A, A^{\perp}} \qquad (cut)\ \frac{\vdash \Gamma, A \quad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}$$

$$(\otimes)\ \frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \qquad (\wp)\ \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B}$$

$$(\forall)\ \frac{\vdash \Gamma, A \quad \alpha \notin FV(\Gamma)}{\vdash \Gamma, \forall \alpha.A} \qquad (\exists)\ \frac{\vdash \Gamma, A[^{B}/_{\alpha}]}{\vdash \Gamma, \exists \alpha.A}$$

$$(h)\ \frac{}{\vdash \Gamma} \qquad (w)\ \frac{\vdash \Gamma}{\vdash \Gamma, A}\ ,$$

$$(c)\ \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} \qquad (!)\ \frac{\vdash B_1, \ldots, B_m, A \quad m \leq 1}{\vdash ?B_1, \ldots, ?B_m, !A}$$

$$(\S)\ \frac{\vdash \Gamma, \Delta}{\vdash ?\Gamma, \S\Delta}$$

Fig. 4. Light Affine Logic.

$$A ::= \mathcal{T} \mid A \otimes A \mid A \wp A \mid A \multimap A \mid A \multimap^{\perp} A \mid \forall \alpha.A \mid \exists \alpha.A \mid !A \mid ?A \mid \S A$$

Fig. 5. Formulas of **ILAL**.

Like in Linear Logic (**LL**), we may only perform contraction on ?-modal assumptions $?A$. However, in **LAL** (**LLL**), the potential explosion of the computation, due to an explosion of the use of (c)-rules, is taken under control. This is achieved by means of two simultaneous restrictions on **LL**. First, every (!)-rule has at most one assumption. So, the duplication of an instance r of a (!)-rule, as effect of the cut elimination, can produce at most one instance of the duplicating (c)-rule, used to contract the assumption of r, if any. Second, the dereliction of an occurrence r of a (!)-rule, which keeps its premise, while deleting r, is completely banned from **LAL** (**LLL**). Namely, any of the logical rules of **LAL** (**LLL**) cannot change the number of instances of (!)-rules following it, as effect of the cut elimination. This mechanism becomes evident by adopting the proof-nets for the sequent calculus in Figure 4.

The two, just described, restrictions enormously decrease the overall expressive power, which is recovered by introducing a modality $\S$ by a $(\S)$-rule. Like in **LLL**, $\S$ is self-dual. A $(\S)$-rule may use *multiple* occurrences of ?-modal assumptions. To prevent the explosion of the occurrences of (c)-rules, as the cut elimination proceeds, **LAL** (**LLL**) cannot contract any $\S$-modal formula. Namely, any occurrence of a $(\S)$-rule can be duplicated.

The intuitive description about how **LAL** (**LLL**) controls the cut elimination complexity, never mentioned (w)-rule. The point about **LAL** is as follows: the unrestricted weakening does not falsify any of the previously described intuitions.

3. INTUITIONISTIC LIGHT AFFINE LOGIC

This section focuses our attention on Intuitionistic Light Affine Logic (**ILAL**), by restricting the sequent calculus in Figure 4. We start by the same set of literals as in Figure 1 to define the set $\mathcal{I}$, which extends $\mathcal{F}$. The grammar of $\mathcal{I}$ is in Figure 5, and $\mathcal{I}$ is partitioned by the set equations in Figure 6, which extends the set in Figure 3.

$$
\begin{array}{rcl}
(\alpha^{\perp})^{\perp} & = & \alpha \\
(!A)^{\perp} & = & ?(A^{\perp}) \\
(\S A)^{\perp} & = & \S(A^{\perp}) \\
(?A)^{\perp} & = & !(A^{\perp}) \\
(A \otimes B)^{\perp} & = & A^{\perp} \parr B^{\perp} \\
(A \parr B)^{\perp} & = & A^{\perp} \otimes B^{\perp} \\
(\forall \alpha.A)^{\perp} & = & \exists \alpha.A^{\perp} \\
(\exists \alpha.A)^{\perp} & = & \forall \alpha.A^{\perp} \\
(A \multimap B)^{\perp} & = & A^{\perp} \multimap^{\perp} B^{\perp} \\
(A \multimap^{\perp} B)^{\perp} & = & A^{\perp} \multimap B^{\perp}
\end{array}
$$

Fig. 6. "De Morgan" laws on the formulas of **ILAL**.

$$
(ax)\ \frac{}{\vdash A; A^{\perp}} \qquad (cut)\ \frac{\vdash \Gamma; A \qquad \vdash A^{\perp}, \Delta; B}{\vdash \Gamma, \Delta; B}
$$

$$
(\otimes)\ \frac{\vdash \Gamma; A \qquad \vdash \Delta; B}{\vdash \Gamma, \Delta; A \otimes B} \qquad (\parr)\ \frac{\vdash \Gamma, A, B; C}{\vdash \Gamma, A \parr B; C}
$$

$$
(\multimap)\ \frac{\vdash \Gamma, A; B}{\vdash \Gamma; A \multimap B} \qquad (\multimap^{\perp})\ \frac{\vdash \Gamma; A \qquad \vdash B, \Delta; C}{\vdash \Gamma, A \multimap^{\perp} B, \Delta; C}
$$

$$
(\forall)\ \frac{\vdash \Gamma; A \qquad \alpha \notin FV(\Gamma)}{\vdash \Gamma; \forall \alpha.A} \qquad (\exists)\ \frac{\vdash \Gamma, A[^{B}/_{\alpha}]; C}{\vdash \Gamma, \exists \alpha.A; C}
$$

$$
(h)\ \frac{}{\vdash \Gamma; A} \qquad (w)\ \frac{\vdash \Gamma; B}{\vdash \Gamma, A; B}
$$

$$
(c)\ \frac{\vdash \Gamma, ?A, ?A; B}{\vdash \Gamma, ?A; B} \qquad (!)\ \frac{\vdash B_1, \ldots, B_m; A \qquad m < 1}{\vdash ?B_1, \ldots, ?B_m; !A}
$$

$$
(\S)\ \frac{\vdash \Gamma, \Delta; A}{\vdash ?\Gamma, \S\Delta; \S A}
$$

Fig. 7. Intuitionistic Light Affine Logic.

The sequent calculus of **ILAL** is in Figure 7. Its judgments have form $\vdash \Gamma;\ A$, where Γ may be empty. Of course, the single-side sequent calculus in Figure 7 can be related to a single-side calculus as follows. Define the *injection* $\star : \mathcal{I} \to \mathcal{I}$, as in Figure 8. Then, prove:

LEMMA 3.1. $\vdash \Gamma; A$ *if, and only if,* $(\Gamma^{\star})^{\perp} \vdash A^{\star}$, *provided that both* $(\Gamma^{\star})^{\perp}$ *and* $A^{\star}$ *are free of occurrences of the negation* $\perp$.

Observe that requiring $(\Gamma^{\star})^{\perp}$ and $A^{\star}$ to be free of $\perp$ means that the single-side sequent calculus we have in mind inductively builds usual Linear logic formulas, (of course, extended with those containing the logical operator $\S$), starting from a set of positive literals $\{\alpha, \beta, \ldots\}$, using $\otimes$, $\multimap$, $\forall$, $!$, $\S$. So, any issue related to the self-duality of $\S$ gets forgotten. However, at least for the moment, we stick to the intuitionistic single-side sequents because it is directly related to the structure of the proof-nets for **ILAL** of Section 4. Recall that the proof-nets are our favorite language to prove that **ILAL** can be used as a programming language with a polynomial normalization cost. In Section 7, we shall move

$$
\begin{aligned}
\alpha^\star &= \alpha \\
(A^\perp)^\star &= (A^\star)^\perp \\
(A \otimes B)^\star &= A^\star \otimes B^\star \\
(A \multimap B)^\star &= (A^\star)^\perp \multimap B^\star \\
(!A)^\star &= !(A^\star) \\
(\S A)^\star &= \S(A^\star) \\
(\forall \alpha.A)^\star &= \forall \alpha.(A^\star)
\end{aligned}
$$

Fig. 8. From double to single-side sequents, and vice-versa.

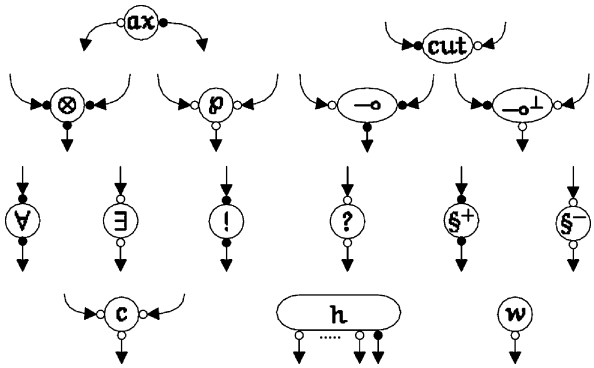

Fig. 9. The vertices of the oriented graphs of $\mathcal{G}$.

to the more traditional double-side sequent calculus to design a term calculus, which we shall exploit as a compact representation of the proof-nets.

4. THE PROOF-NETS

This section adapts the technology of proof-nets, given for usual **LL**, to our purposes. Our references are mainly Tortora [2000b; 2000a], which give an exhaustive reworking and extensions of previous results about proof-nets for **LL**.

Focus on the language $\mathcal{G}$ of *oriented graphs* such that:

—the vertices of the graphs are in Figure 9. The solid and hollow circles are the *polarized ports* of the vertices: a solid circle denotes *positive* ports. The others are *negative*. The arcs entering a port of a node are its *assumptions*. Those exiting a port of a node are its *conclusions*. The number of *premises* of a node is its indegree. The *conclusions* are as many as the value of the outdegree.

—the oriented arcs of the graphs are labeled by the formulas of $\mathcal{I}$, and they only connect ports with the same polarity, according to the indegree and to the outdegree of the vertex.

Note that, from positive ports, it does not necessarily exit an arc. “Node” and “vertex” are two interchangeable words, like “arc” and “edge.” The notion of ports/premises/conclusions obviously extends to any graph of $\mathcal{G}$. When drawing graphs, if the label of a port is a multi-set of formulas, that port is, in fact, a set of ports.

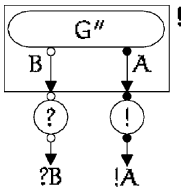

Fig. 10. A !-box.

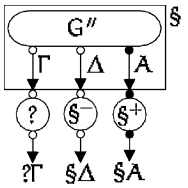

Fig. 11. A §-box.

Definition 4.1. A *proof-structure* of **ILAL** is a graph G of $\mathcal{G}$ such that:

(1) every edge of G is conclusion of a unique node, and is premise of at most one node;

(2) G may contain !-boxes. A !-box is a subgraph G' of G that can be depicted as in Figure 10, where:
—the conclusion of exactly one !-node is the *principal* port of G';
—at most one ?-node can be the *secondary* port of G';
—the graph G'' in G' is a proof-structure;

(3) G may contain §-boxes. A §-box is a subgraph G' of G that can be depicted as in Figure 11, where:
—the conclusion of exactly one $\S^+$-node is the *principal* port of G';
—the remaining ports of G' are *secondary* and must be partitioned into two, possibly empty, sets. Every port of the first set must be the conclusion of a $\S^-$-node. Every port of the other set, must be the conclusion of a ?-node;
—the graph G'' in G' is a proof-structure;

(4) G cannot contain !-nodes, ?-nodes, $\S^+$-nodes and $\S^-$-nodes not associated to a !-box or to a §-box;

(5) every pair of boxes is either disjoint or one is included into the other;

(6) every $\forall$-node of G binds a distinct variable, its *eigenvariable*. Every eigenvariable is subject to the following constraints:
—it cannot occur free in the formulas labeling the conclusions of G;
—if α is the eigenvariable of some $\forall$-node, contained in a box B, then every occurrence of α must be in B;

(7) every occurrence of a w-node has a *jump* associated with it. A jump is an occurrence of any of the nodes of G.

Definition 4.2. Any node n of a given proof-structure *depends (on an eigenvariable)* α if α is free in the formula labeling one of the premises of n.

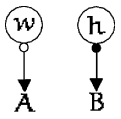

Fig. 12. A net peculiar of **ILAL**.

Note that an $\exists$-node depends on α if its premise is labeled by $A[^{B}/_{\beta}]$, and α is free in B.

Definition 4.3. Any node n of a given proof-structure *is at depth* d if n is enclosed in d boxes.

The *maximal depth* ∂ of a proof-structure is the maximal depth of its nodes.

Obviously, the depth extends to the edges and the boxes of a proof-structure.

Definition 4.4. A *switching graph* $S(\Pi)$ of a proof-structure Π:

(1) contains all the nodes of Π at level 0, and
(2) in place of every box (either ! or §) at level 0 in Π, contains an instance of an h-node with as many ports as the box.
(3) the edges of $S(\Pi)$ are *unoriented* and:
   (a) every node $\multimap$, $\wp$, c of $S(\Pi)$ has only one of its two premises it had in Π;
   (b) for every w-node n, the jump from n to a node m of Π becomes an edge between n and m in $S(\Pi)$;
   (c) for every $\forall$-node n, $S(\Pi)$ has either the edge premise of n, or its has an edge between n and one of the nodes of Π that depend on the eigenvariable of n.

Definition 4.5. A proof-structure Π is a *proof-net* when:

(1) every $S(\Pi)$ is *both connected and acyclic*, and
(2) the proof-structure of every box of Π at level 0 *is* a proof-net.

THEOREM 4.6 (SEQUENTIALIZATION). *Let* Π *be a proof-net whose ports are labeled by the formulas in* Γ, A, *being* A *positive, and, every formula of* Γ, *negative. Then,* $\vdash \Gamma; A$ *is derivable in the sequent calculus of* **ILAL**.

The proof is step-wise coincident to the proof of sequentialization for **LL**, as presented in Tortora [2000b; 2000a], except for the additives, which are not present in **ILAL**. We just want to comment about what happens, relatively to the differences between the proof-nets of **ILAL** and of **LL**. Every jump in **LL** is between a w-node and an ax-node. In **ILAL**, we allow jumps from an instance m of a w-node to any other node n. First, forgetting about the additives of **LL**, this results in a larger set of legal proof-structures: Figure 12 shows an **ILAL** proof-net, without a correspondent in **LL**. Liberalized jumps simply say that, in the sequent calculus, a (w)-rule can be used immediately after the rule that introduces n, during the sequentialization. Another difference between **LL** and **ILAL** is that the premises and the conclusions of the ?-nodes in a box of **LL** are labeled by the same formula, unlike **ILAL**. However, the sequentialization

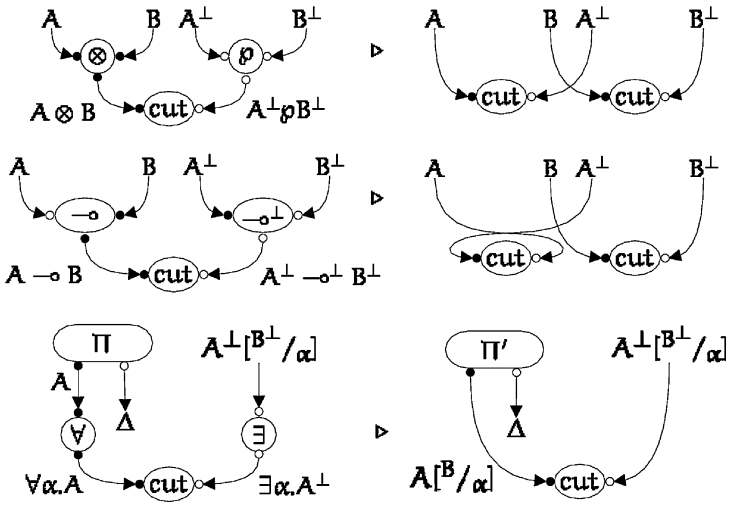

Fig. 13. The linear *ers*.

proceeds inductively: every !-box B corresponds to a well-formed derivation of **ILAL**, that concludes by an instance r of a (!)-rule. The form of the premises of the proof-net inside B, which coincides with those of r, are uninfluential to the sequentialization that keeps building a well-formed derivation below r. The same is true for §-boxes. Finally, the polarization only needs to map a positive port to the unique positive formula of the sequent and negative ports to the negative formulas, when sequentializing.

5. THE NORMALIZATION OF THE PROOF-NETS

The *elementary reduction steps*, also called *elementary rewriting steps*, and abbreviated as *ers*, are in Figures 13, 14, and 15. Figures 16 and 17, instead, introduce the *garbage collecting steps* (*gcs*).

Figure 13 introduces the *linear ers*. The first and second *ers* in Figure 13 describe how the pairs of nodes $\otimes/\wp$ and $\multimap/\multimap^{\perp}$ annihilates. The third one is the annihilation of the pair $\forall/\exists$, subject to: Π' is Π where B replaces every free occurrence of the eigenvariable α.

Figure 14 introduces the *shifting ers*, where $(\Box A)^{\perp} \equiv \star A^{\perp}$, and, if $\Diamond \equiv !$, then: (i) $\star \equiv ?$, (ii) the nodes represented by n cannot exist, and (iii) if m exists, it is a single !-node. Otherwise, if $\Diamond \equiv \S$, then: (i) $\star \in \{?, \S^{-}\}$, and (ii) the sequences of nodes, represented by both m and n, exist according to the formation rules of ! and §-boxes.

Figure 15 defines the *polynomial* ers that only duplicates !-boxes. Observe that the contraction on the ports labeled with $?B$ cannot exist if the !-box does not have such ports.

Figure 16 defines the first set of *gcs*. The first *gcs* replaces a *cut*-node, followed by an *ax*-node, with a link. The symmetric configuration, with an *ax*-node, followed by a *cut*-node, originates a *gcs* as well, with the same behavior. The

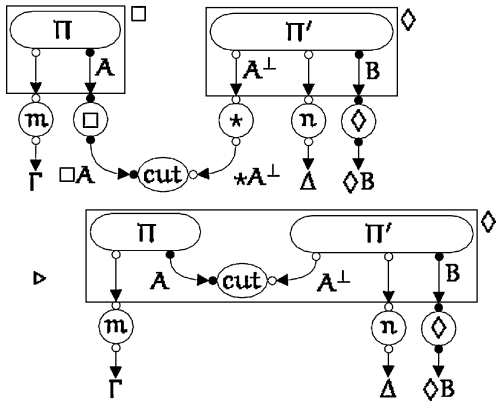

Fig. 14. The shifting *ers*.

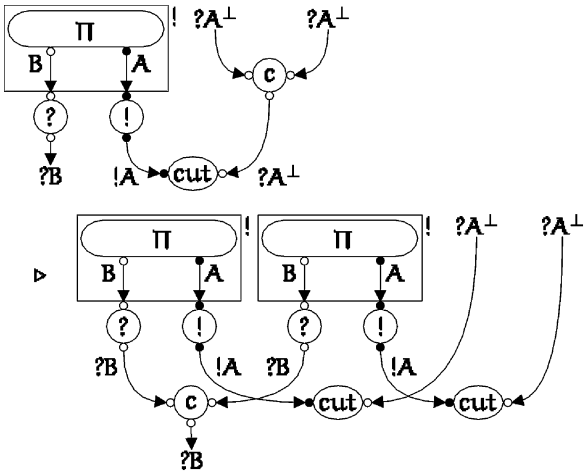

Fig. 15. The polynomial *ers*.

second *gcs* requires $(\Box A)^{\perp} \equiv \star A^{\perp}$. It generalizes the standard rewriting step that, in **LL**, only erases the !-boxes. Note that the graph resulting from the step is a sequence of as many w-nodes as the components of Γ. However, the unconstrained weakening of **ILAL** requires the simplification of a number of new configurations, with respect to the proof-nets of **LL**, to get to a *cut*-free proof-net of **ILAL**. In particular, for preserving the structural invariance that a *cut*-node plugs together positive and negative ports of a net, the third *gcs* in Figure 16 exploits an h-node with a *single* conclusion. Observe also the last *gcs*, where a pair h/w annihilates.

Figure 17 defines the second, and last, set of *gcs*, which can be viewed as complementary to those in Figure 16, and such that $(\star A)^{\perp} \equiv \Box A^{\perp}$.

As usual, a *redex* is the graph to the left of an *ers*, or of a *gcs*, while a *reduct* is the graph to its right. If not otherwise stated, we write *rewriting steps*, meaning all the *ers* and the *gcs* in the set *ers* ∪ *gcs*. A net Π is $\triangleright$-*normal*, or simply normal,

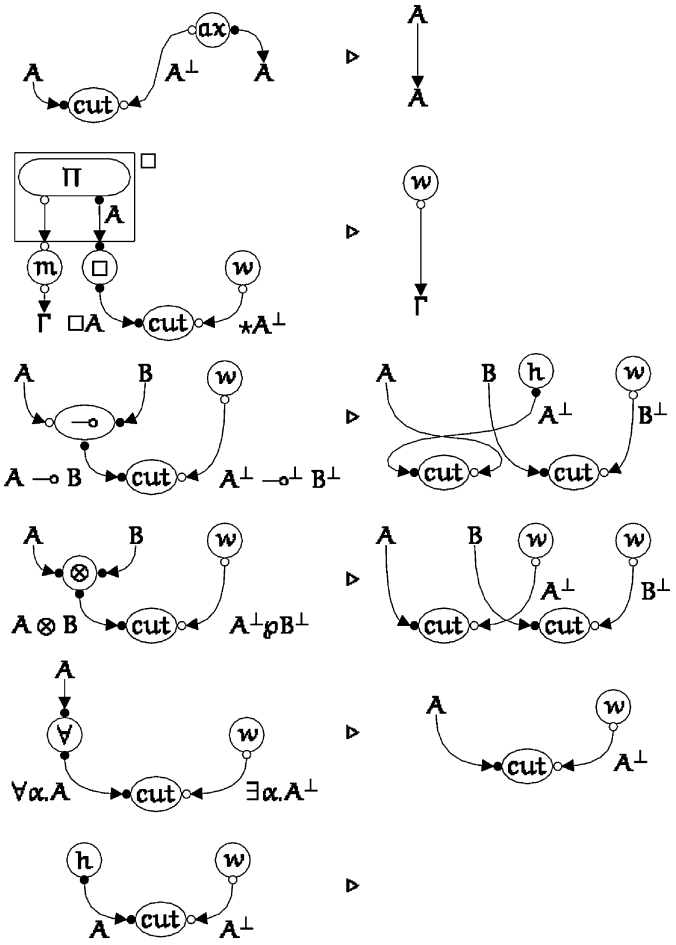

Fig. 16. The *gcs* relative to the *w*-nodes.

if it cannot be rewritten by any rewriting step. Also, a net is *linear-normal* if it cannot be rewritten by any linear *ers*. Analogous definitions apply to shift *ers*, poly *ers*, and to *gcs*.

6. THE PROPERTIES OF THE NORMALIZATION

This section shows that a *suitable* set of proof-nets can be used as a programming language with the expected costs. From now on:

"proof-nets" means the nets free of h-nodes with negative ports.

This restricted set of proof-nets is closed under $\triangleright$, which can reduce every proof-net Π to a unique normal form, in a number of $\triangleright$ steps bounded by a polynomial in the dimension of Π. We shall also explicit the relation between the proof-nets and the single-side sequent calculus of **ILAL**, with respect their dynamics.

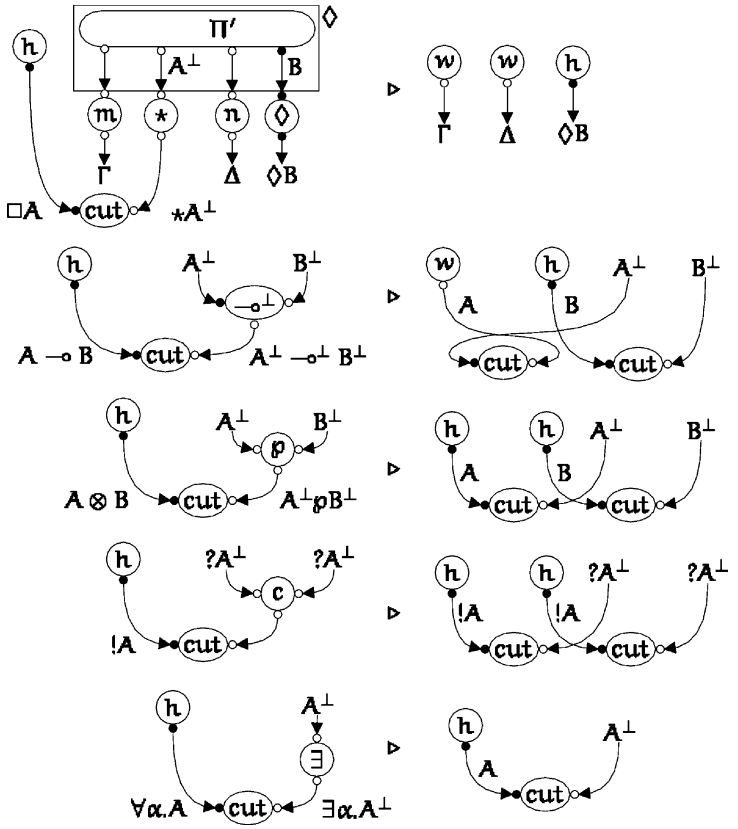

Fig. 17. The *gcs* relative to the *h*-nodes.

6.1 Stability under rewriting

THEOREM 6.1. *The set of proof-nets of* **ILAL** *is closed under* $\triangleright$.

PROOF: The proof is divided into two parts. The rewriting steps must produce proof-nets whose *h*-nodes are free from negative ports, and whose switching graphs are connected and acyclic. The first requirement holds by definition of rewriting steps. The proof relative to the switching graphs is step-wise coincident to the analogous proof for **LL**, as presented in Tortora [2000b; 2000a], except for both additives and *gcs*. Additives are not an issue, since they are not in **ILAL**. Let us focus on *gcs*, for proving that they transform proof-nets in proof-nets. For every *gcs* in Figure 16 and for the last three *gcs* in Figure 17, this can be proved by assuming the existence of a proof-structure, with a reduct in it, that cannot be a proof-net, because some of its switching graphs is cyclic or unconnected. This assumption, however, implies that the proof-structure, containing the corresponding redex, could not be a proof-net as well. Finally, focus on the first two *gcs* in Figure 17. They introduce *w*-nodes not initially present in the redex. The connectivity of the switching graphs holds due to the jump between the new *w*-nodes and the *h*-nodes, labeled $\Diamond B$ in the first *gcs* and B in

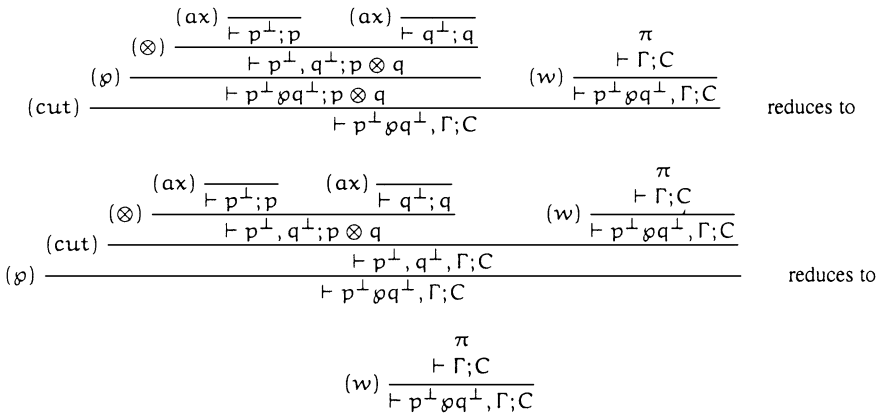

Fig. 18. Some cut-elimination steps at work.

the second one. Acyclicity follows from the acyclicity of the (switching graphs) of the proof-net, before the reduction of the *gcs*.

6.2 Relating the Normalization of the Proof-nets and of the Sequent Calculus

THEOREM 6.2. *Let π be any derivation of $\vdash \Gamma; A$. Assume Π_π be the proof-net that corresponds to π, in accordance with Theorem* 4.6. *Let also Π' be any proof-net such that Π_π reduces to, after some rewriting steps. If π' corresponds to Π', according to Theorem* 4.6, *then there exists a sequence of cut elimination steps that transforms π into π'.*

We leave as an exercise to the reader to make explicit all the cut-elimination steps on the sequent calculus, and the details of the proof. The statement can be proved by showing that it holds case by case, on the definition of the rewriting steps.

Here, we want to detail out an example to show why the following statement does not hold:

> *Let π and π' be two derivations of $\vdash \Gamma; A$. Call Π_π and Π'_π the two proof-nets, corresponding to π and π', respectively, in accordance with Theorem* 4.6. *If π reduces to π', by some cut elimination steps, then Π_π normalizes to Π'_π, by means of some rewriting steps.*

Look at Figures 18, and 19. The assumptions are that Π_π corresponds to π and that the uppermost derivation of Figure 18 corresponds to the uppermost proof-net in Figure 19. Observe that the normalization of the topmost proof-net in Figure 19 does not yield a proof-net corresponding to the lowermost derivation that we can obtain after some cut elimination steps, applied to the derivation in Figure 18. The reason is that the proof-nets erase structure using a node-by-node process, while the sequent calculus can also erase blocks of derivations at once. However, we want to remark that the sequent calculus can be as fine-grained as the proof-nets when erasing structure, due to its (h)-rule. This is why the Theorem 6.2 holds; namely, from the dynamic point of view, the sequent calculus can always copy what the proof-nets do.

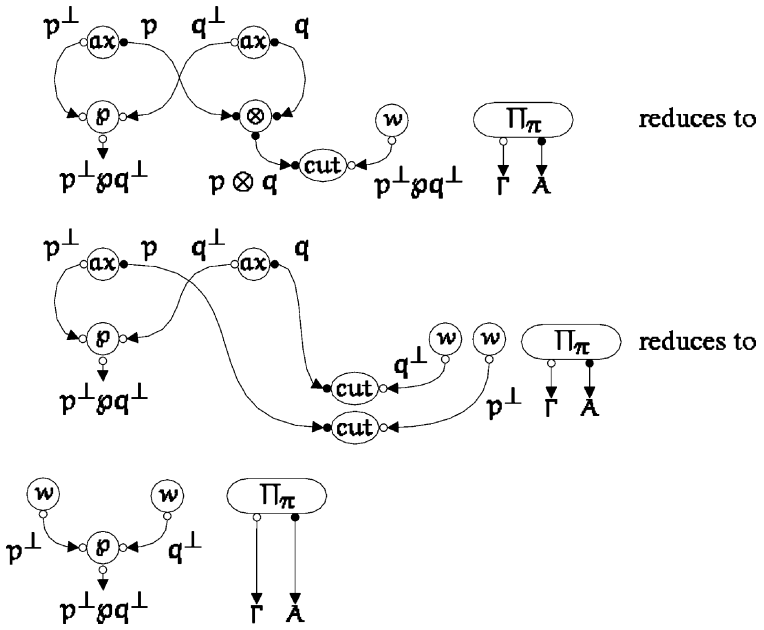

Fig. 19. Some garbage collecting steps at work.

6.3 Complexity Bounds

This section proves the existence of a strategy for the rewriting steps that normalizes every proof-net Π in a number of rewriting steps, bounded by a polynomial in the dimension of Π. The bound is:

$$O(\mathbf{D}^{6^{\partial}}(\Pi)),$$

being $\mathbf{D}(\Pi)$ the dimension of Π, and ∂ the maximal depth of Π.

Definition 6.3. Let Π be a proof-net, and $l \leq \partial$.

—The *dimension of* Π *at level* l, denoted by $d_l(\Pi)$, is the number of nodes in Figure 9 at level l;

—The *dimension* $\mathbf{D}(\Pi)$ is simply $\sum_{l=0}^{\partial} d_l(\Pi)$.

In absence of ambiguities, Π is omitted.

The complexity bound holds for a specific reduction strategy, which will be introduced after some lemma and definitions.

Definition 6.4. Let Π be a net, and $l \leq \partial$. Π *is* l*-normal* if:

—Π is linear-normal at every level $0 \leq i \leq l$,

—Π is both shift and poly-normal at every level $0 \leq i \leq l-1$,

—Π *gcs*-normal at every level $0 \leq i \leq l-1$.

The reason why we differentiate between the second and the third clause in the definition here above lies in the definition of the reduction strategy we shall

Node	lwgt
$\otimes, \wp, \multimap, \multimap^{\perp}, \mathrm{c}$	3
$\forall, \exists, !, ?, \S^{+}, \S^{-}, w, \mathrm{h}, \mathrm{cut}$	1

Fig. 20. Linear weight of the nodes.

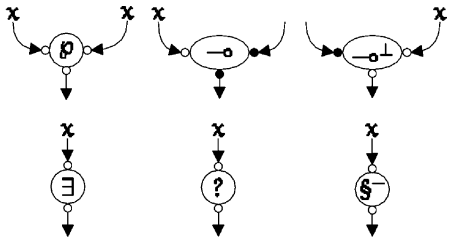

Fig. 21. The weight of the links: The base case.

find in Definition 6.10. Intuitively, the reduction strategy alternates blocks of linear reduction steps with blocks of both shifting and polynomial reduction steps. Every block may produce some garbage whose elimination simplifies the proof of the result about the complexity of the whole strategy. So, we need to explicitely express when a given level is *gcs*-normal.

Definition 6.5. Let Π be an $l-1$-normal proof-net. The leftmost column of the table in Figure 20 defines the *linear weight* (lwgt) of every (instance) of the nodes of Π, at level l.

The *linear weight at level* l *of* Π ($\mathrm{lwgt}_l(\Pi)$) is the sum of all the linear weights of its nodes at l, in accordance with Figure 20.

FACT 6.1. *Let* $l \leq \partial(\Pi)$, *for some* Π. *Assume that* Π *rewrites to* Π' *through a linear* ers *or through a* gcs *at level* l. *Then,* $\mathrm{lwgt}_l(\Pi) > \mathrm{lwgt}_l(\Pi')$.

It is enough to check that the relation holds for every *ers* and *gcs* in Figures 13, 16, and 17.

LEMMA 6.6. *Let* $l \leq \partial(\Pi)$, *for some* Π. *The linear* ers *at level* l, *together with the* gcs, *applied at the same level*, *are strongly normalizing*, *after*, *at most*, $3d_l(\Pi)$ *steps*.

Strong normalizability is the direct consequence of Fact 6.1. The bound holds because we cannot perform more linear *ers* and *gcs* than the value of $\mathrm{lwgt}_l(\Pi)$, which is at most $3d_l(\Pi)$.

Now, we focus on the normalization of shift and poly *ers*. We need to associate a *shift/poly weight* to every node *and arc*.

Definition 6.7. Let Π be an l-normal proof-net, with $l \leq \partial$. The *shift/poly weight of* Π *at level* l $\mathrm{spwgt}_l(\Pi)$ is a *partial* function from the nodes and the links of Π at level l to $\mathbb{N}$. Let x be any link of Π at level l:

—$\mathrm{spwgt}_l(\Pi)(x) = 0$ if either x is a port of Π, or it is the premise of one of the nodes $\wp, \exists, ?, \S^{-}$, or it is the negative premise of one of the nodes $\multimap, \multimap^{\perp}$, as in Figure 21;

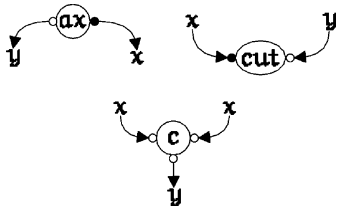

Fig. 22. The weight of the links: The identity case.

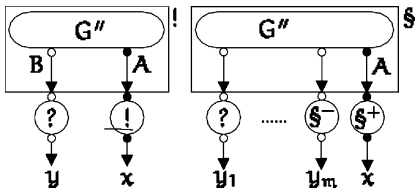

Fig. 23. The weight of the links: The box case.

—$\mathrm{spwgt}_l(\Pi)(x) = \mathrm{spwgt}_l(\Pi)(y)$ if x is the positive port of either an ax-node or of a cut-node, and y is the negative port of one of the same nodes, as in Figure 22;

—$\mathrm{spwgt}_l(\Pi)(x) = \mathrm{spwgt}_l(\Pi)(y)$ if x is one of the premises of a c-node and y is the conclusion of the same c-node, as in Figure 22;

—$\mathrm{spwgt}_l(\Pi)(x) = 1 + \mathrm{spwgt}_l(\Pi)(y)$ if y is a secondary port of some !-box, and x the principal port of the same !-box, as in Figure 23. If x is the unique port of the !-box, then $\mathrm{spwgt}_l(\Pi)(x) = 1$;

—$\mathrm{spwgt}_l(\Pi)(x) = 1 + \sum_{i=1}^{m} \mathrm{spwgt}_l(\Pi)(y_i)$ if y_i is a secondary port of some §-box, and x the principal port of the same §-box, as in Figure 23. If x is the unique port of the §-box, then $\mathrm{spwgt}_l(\Pi)(x) = 1$.

Let n be any node of Π at level l:

—$\mathrm{spwgt}_l(\Pi)(n) = \mathrm{spwgt}_l(\Pi)(x)$ if n is a $!/\S^+$-node and x its conclusion;

—$\mathrm{spwgt}_l(\Pi)(n) = (\mathrm{spwgt}_l(\Pi)(y))^2$ if n is a c-node and y its conclusion;

—$\mathrm{spwgt}_l(\Pi)(n) = 1$ if n is either an ax or a cut-node.

$\mathrm{spwgt}_l(\Pi)$ is undefined on any other link and node.

The definition here above tells us that every contraction node is as "heavy" as the shift/poly weight of the net, rooted at its conclusion. In particular, the links whose shift/poly weight is 0 are those where a c-node stops moving down, through a net, during the normalization.

Definition 6.8. Call "*poly* gcs" the two first *gcs* in Figure 16.

The meaning of the definition here above must be looked for in the details of the definition of the strategy, whose building blocks are the *rounds*, which are going to be introduced in Definition 6.10. Intuitively, the poly *gcs* are the only garbage collecting steps that a block of shifting and polynomial rewriting

steps, applied at level $l-1$ may originate, starting from a proof-net that is $l-1$-normal. All this is formalized by Fact 6.3 below.

By the definition of shift *ers*:

FACT 6.2. *Let* Π *be* $l-1$*-normal, with* $l \leq \partial$*. If the reduction of a shift ers in* Π *at* $l-1$ *creates a new redex, it is at level* l*.*

FACT 6.3. *Let* Π *be* $l-1$*-normal, with* $l \leq \partial$*. The reduction of poly ers at level* $l-1$ *can only create new shift/poly ers and new poly gcs at* $l-1$*.*

For proving this, it is enough to assume the possibility to create some other kind of redex in Π. This would contradict that Π was l-normal.

FACT 6.4. *Let* Π *be* $l-1$*-normal, with* $l \leq \partial$*. Assume that* Π *rewrites to* Π' *through a shift/poly ers or a poly gcs at level* $l-1$*. Then,* $\mathrm{spwgt}_{l-1}(\Pi) > \mathrm{spwgt}_{l-1}(\Pi')$.

For proving Fact 6.4, it is enough to check that the relation holds for every of the mentioned *ers* and *gcs* in Figures 14, 15, and 16, exploiting Facts 6.2 and 6.3 to assure that we shall never need to evaluate the undefined weight of some node.

LEMMA 6.9. *Let* Π *be* $l-1$*-normal, with* $l \leq \partial$*. The shift/poly ers at level* $l-1$*, together with the poly gcs, applied at the same level, are strongly normalizing in a number of steps which is* $O(d^3_{l-1}(\Pi))$*.*

Let rewrite Π *into* Π' *which is both shift/poly normal and gcs normal at level* $l-1$*. Then,* $d_j(\Pi')$ *is* $O(d_{l-1}(\Pi) \cdot d_j(\Pi))$*, for every level* $l-1 \leq j \leq \partial(\Pi')$*.*

Strong normalizability is the direct consequence of Fact 6.4. The bound holds because, *for every* c-node, we cannot perform more poly *ers* than the value of $\mathrm{spwgt}^2_{l-1}(\Pi)$, which cannot be greater than $d^2_{l-1}(\Pi)$. Since the c-nodes cannot be more than $d_{l-1}(\Pi)$, the total number of poly *ers* is, at most $d^3_{l-1}(\Pi)$. Now, firstly observe that every poly *ers* adds two nodes at level $l-1$. Due to Fact 6.2 and 6.3, at most $d_{l-1}(\Pi)$ c-nodes, each generating at most $2d_{l-1}(\Pi)$ redexes, yield a quadratic overhead of shift *ers* and poly *gcs* to be reduced, besides the cubic bound for poly *ers*. This gives the normalization bound $O(d^3_{l-1}(\Pi))$. Second, besides adding two nodes at l, every poly *ers* doubles a !-box. Iterating this duplication at most $d_{l-1}(\Pi)$ times for each c-node, it yields $d_j(\Pi') \leq d_{l-1}(\Pi) \cdot d_j(\Pi)$, for every $l-1 \leq j \leq \partial(\Pi')$.

Lemmas 6.6 and 6.9 justify the definition of the *reduction rounds*:

Definition 6.10. Let Π be $l-1$-normal, with $0 \leq l \leq \partial$. A *reduction round* $\triangleright^l_\rho$ on Π at l reduces the rewriting steps in the following order:

(1) *all* the shift/poly *ers* and poly *gcs* at $l-1$, in any order,

(2) *all* the linear *ers* and *gcs* at l, in any order.

Then, $\triangleright^l_\rho$ stops.

Note that $l=0$ in the definition above means that we do not apply point (1), because Π does not have negative levels.

LEMMA 6.11. *Let* Π *and* Π' *be such that* Π *is* $l-1$*-normal, with* $0 \leq l \leq \partial$, *and* $\Pi \rhd_\rho^l \Pi'$. *Then*:

(1) Π' *is* l*-normal*;

(2) *The length of the reduction* $\Pi \rhd_\rho^l \Pi'$ *is* $O(\mathbf{D}^3(\Pi))$;

(3) $d_j(\Pi')$ *is* $O(\mathbf{D}^2(\Pi))$, *for all* $l-1 \leq j \leq \partial$.

The first point is true by Definition 6.10 of $\rhd_\rho^l$, due to Lemmas 6.6 and 6.9. The second and the third points are a consequence of Lemma 6.9, where $d_{l-1}(\Pi)$ and $d_j(\Pi)$, for every $l-1 \leq j \leq \partial(\Pi')$, are certainly smaller than $\mathbf{D}(\Pi)$.

THEOREM 6.12. *Every proof-net* Π *of* **ILAL** *can normalize in* $O(\mathbf{D}^{6^\partial}(\Pi))$ *steps.*

Due to Lemma 6.11, iterating the rounds from the lower level to the upper one on Π, we have:

$$\Pi \equiv \Pi_0 \rhd_\rho^0 \cdots \rhd_\rho^{i-1} \Pi_i \rhd_\rho^i \Pi_{i+1} \rhd_\rho^{i+1} \cdots,$$

where Π_i rewrites in Π_{i+1}, using $O(\mathbf{D}^3(\Pi_i))$ steps. So, Π_i is obtained after:

$$O\left(\sum_{k=0}^{i-1} \mathbf{D}^{6^k}(\Pi)\right) \tag{1}$$

rewriting steps. The reduction sequence here above cannot be longer than $\partial(\Pi)$ because *ers* and *gcs* are, level by level, strongly normalizing. In particular, it is shorter if some *gcs* erases, at some point, the last box constituting the level ∂ of Π. So, the upper limit of (1) is $\partial(\Pi)$. Due to the following relations:

$$\sum_{k=0}^{\partial(\Pi)-1} \mathbf{D}^{6^k}(\Pi) \leq \sum_{k=0}^{6^{\partial(\Pi)}} \mathbf{D}^k(\Pi) \leq \frac{\mathbf{D}^{6^{\partial(\Pi)}+1}(\Pi)-1}{\mathbf{D}(\Pi)-1},$$

we can conclude that Π normalizes in, at most, $O(\mathbf{D}^{6^\partial}(\Pi))$ steps.

6.4 Confluence

Finally, we prove that the proof-nets can be used as a programming language:

THEOREM 6.13. *The set of normal forms of every proof-net of* **ILAL** *is a singleton.*

The proof follows a classical scheme, based on local convergence and strong normalizability. To that aim:

Definition 6.14. For every $0 \leq l \leq \partial(\Pi)$, Π' is *an* l*-normal form* of a proof-net Π, if $\Pi \equiv \Pi_0 \rhd_\rho^0 \cdots \rhd_\rho^l \Pi_{l+1} \equiv \Pi'$.

Observe that Π' here above is l-normal, due to Lemma 6.11. Moreover, it is (boring but) simple to check the local confluence of the rewriting steps in $ers \cup gcs$:

$$\mathsf{p} ::= \mathtt{T}_{\mathtt{variables}} \mid \mathsf{p} \otimes \mathsf{p}$$

Fig. 24. The patterns for the concrete syntax.

$$M, N ::= \mathtt{T}_{\mathtt{variables}} \mid (\lambda \mathsf{P}.M) \mid (MN) \mid M \otimes N \mid !M \mid \bar{!}M \mid \S M \mid \bar{\S} M$$

Fig. 25. The concrete syntax.

LEMMA 6.15. *If* Π *rewrites in a single rewriting step to both* Π_1 *and* Π_2, *then there is* Π_3 *to which both* Π_1 *and* Π_2 *rewrite after some*, *possibly none*, *steps.*

At this point, a simple inductive argument implies the proof of Theorem 6.13. Let $l = 0$, and assume the existence of two 0-normal forms Π_1 and Π_1' of Π. This would contradict the absence of critical pairs, among the linear *ers* and the *gcs*, as proved by Lemma 6.15. If $l > 0$, by induction, Π_l is the unique $l-1$-normal form of Π. The assumption of the existence of two l-normal forms Π_{l+1} and Π'_{i+1} such that both $\Pi_l \rhd_\rho^l \Pi_{i+1}$ and $\Pi_l \rhd_\rho^l \Pi'_{i+1}$ would again contradict Lemma 6.15, because such an hypothesis would require the existence of some critical pair among the whole set of rewriting steps. Hence, Π' in $\Pi \equiv \Pi_0 \rhd_\rho^0 \cdots \rhd_\rho^{\partial(\Pi)} \Pi_{\partial(\Pi)+1} \equiv \Pi'$ is unique.

This concludes the part of the paper devoted to proving the complexity property of **ILAL**. The next part is about showing it expressive power.

7. A FUNCTIONAL SYNTAX WITH ITS TYPE ASSIGNMENT

This section is about relating **ILAL** to a functional language. The goal is to supply a compact functional language to talk about proof-nets. The way to carry out this relation is to encode a double-side sequent calculus for **ILAL** by means of a functional syntax. Of course, the double-side sequent calculus is equivalent to the single-side one in Figure 7, due to the relation outlined by the injection in Figure 8 and Lemma 3.1.

We start by introducing the raw terms of our functional syntax which contains patterns for pattern matching. The grammar of the terms is in Figure 24. The convention is that the set of patterns is ranged over by P, while $\mathtt{T}_{\mathtt{variables}}$ is ranged over by x, y, w, z.

Figure 25 defines the set Λ of the terms which we take as functional syntax.

For any pattern $x_1 \otimes \cdots \otimes x_n$, the set $\mathtt{FV}(x_1 \otimes \cdots \otimes x_n)$ of its free variables is $\{x_1, \ldots, x_n\}$. As usual, λ binds the variables of M so that $\mathtt{FV}(\lambda \mathsf{P}.M)$ is $\mathtt{FV}(M) \setminus \mathtt{FV}(\mathsf{P})$. The free variable sets of all the remaining terms are obvious as the constructors $\otimes$, $!$, $\S$, $\bar{!}$, and $\bar{\S}$ do not bind variables. Both $!$ and $\S$ build $!$-boxes and $\S$-boxes, respectively, being M the *body*. The term, constructor $\bar{!}$, can mark one of the *entry points*, namely the inputs, of both $!$-boxes, and $\S$-boxes, while $\bar{\S}$ can mark only those of $\S$-boxes.

We shall adopt the usual shortening for λ-terms: $\lambda x_1 \cdots \lambda x_n.M$ is abbreviated by $\lambda x_1 \cdots x_n.M$, and $(M_1 \cdots (M_n N) \cdots)$ by $M_1 \cdots M_n N$, that is, by default, the application is left-associative.

The elements of Λ are considered up to the usual α-equivalence. It allows the renaming of the bound variables of a term M. For example, $!(\lambda x.(\bar{!}\, y)\, x)$ and $!(\lambda z.(\bar{!}\, y)\, z)$ are each other α-equivalent.

The substitution of M for x in N is denoted by $N[{}^{M}/_{x}]$. It is the obvious extension to Λ of the capture-free substitution of terms for variables, defined for the λ-Calculus. For example, $y[{}^{x}/_{y}]$ yields x.

The substitutions can be generalized to $[{}^{M_1}/_{x_1} \cdots {}^{M_n}/_{x_n}]$, which means the simultaneous replacement of M_i for x_i, for every $1 \leq i \leq n$.

We shall use $\equiv$ as syntactic coincidence.

7.1 The Type Assignment

The goal of the type assignment is to associate a logical formula to a term that belongs to a suitable subset of Λ. The logical formula must be inductively built by $\otimes, \multimap, \forall, !, \S$, starting from the set of positive propositional variables $\{\alpha, \beta, \gamma, \ldots\}$. The previously mentioned subset of Λ contains the terms that encode a deduction of a double-side sequent calculus, equivalent to the one in Figure 7, under the injection in Figure 8 and Lemma 3.1.

To formally introduce the type assignment, we need to set some terminology.

Call *basic set of assumptions* any set of pairs $\{x_1 : A_1, \ldots, x_n : A_n\}$ that can be seen as a function with finite domain $\{x_1, \ldots, x_n\}$. Namely, if $i \neq j$, then $x_i \neq x_j$.

An *extended set of assumptions* is a basic set, containing also pairs $\mathsf{P} : A$, that satisfies some further constraints. A pattern $\mathsf{P} \equiv x_1 \otimes \cdots \otimes x_m : A$ belongs to an extended set of assumptions:

(1) if A is $A_1 \otimes \cdots \otimes A_p$, with $p \geq m$, and
(2) if $\{x_1 : B_1, \ldots, x_m : B_m\}$ is a basic set of assumptions, where every B_i is either a single formula, or tensor of formulas.

For example, $\{x : \gamma, y : \beta\}$ is a legal extended set, while $\{z \otimes x : \gamma, y : \beta\}$ is not.

Talking about "assumptions", we generally mean "extended set of assumptions." Meta-variables for ranging over the assumptions are Γ, and Δ.

The *substitutions* on formulas replace formulas for variables in the obvious way.

Figure 26 introduces the double-side sequent calculus of **ILAL**, with term decorations. As a remark, the two rules for the second-order formulas are not encoded by any term. Namely, we introduce a system analogous to Mitchell's language *Pure Typing Theory* [Mitchell 1988]. In our case, the logical system of reference is second order **ILAL**, in place of System $\mathcal{F}$ [Girard et al. 1989].

7.2 The Dynamics for the Functional Syntax

We introduce Λ because we want to use it as a program notation in place of the proof-nets. The reason is to have a more readable syntax. Indeed, the syntax is essentially a standard λ-Calculus enriched with notation to take care of the box borders.

The correct way to use the functional language is formalized by Theorem 6.2, and is summarized in Figure 27. The functional language constitutes the readable form of input and output in a programming session, where the computations are developed by exploiting the proof-nets.

$$(ax)\ \frac{}{x : B \vdash x : B} \qquad (cut)\ \frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : B}{\Gamma, \Delta \vdash N[^{M}/_{x}] : B}$$

$$(w)\ \frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B} \qquad (c)\ \frac{\Gamma, x : !A, y : !A \vdash M : B}{\Gamma, z : !A \vdash M[^{z}/_{x}\ ^{z}/_{y}] : B}$$

$$(\multimap_l)\ \frac{\Gamma \vdash M : A \qquad \Delta, y : B \vdash N : C}{\Gamma, \Delta, x : A \multimap B \vdash N[^{xM}/_{y}] : C} \qquad (\multimap_r)\ \frac{\Gamma, P : B_1 \otimes \ldots \otimes B_n \vdash M : B}{\Gamma \vdash \lambda P.M : B_1 \otimes \ldots \otimes B_n \multimap B}$$

$$(\otimes_l)\ \frac{\Gamma, x_1 : B_1, x_2 : B_2 \vdash M : B}{\Gamma, x_1 \otimes x_2 : B_1 \otimes B_2 \vdash M : B} \qquad (\otimes_r)\ \frac{\Gamma \vdash M : B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash M \otimes N : B \otimes A}$$

$$(!)\ \frac{\ldots x_i : A_i \ldots \vdash M : B \qquad 0 \leq i \leq n \leq 1}{\ldots x_i : !A_i \ldots \vdash !M[\ldots\ ^{\bar{!}x_i}/_{x_i} \cdots] : !B}$$

$$(\S)\ \frac{\ldots x_i : B_i \ldots x'_j : A_j \ldots \vdash M : B \qquad 0 \leq i \leq m \quad 0 \leq j \leq n}{\ldots x_i : !B_i \ldots x'_j : \S A_j \ldots \vdash \S M[\ldots\ ^{\bar{!}x_i}/_{x_i} \cdots\ ^{\bar{\S}x'_j}/_{x'_j} \cdots] : \S B}$$

$$(\forall_l)\ \frac{\Gamma, x : [^{B}/_{\alpha}]A \vdash M : B}{\Gamma, x : \forall\alpha.A \vdash M : B} \qquad (\forall_r)\ \frac{\Gamma \vdash M : A \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall\alpha.A}$$

Fig. 26. Double-side sequent calculus and functional terms.

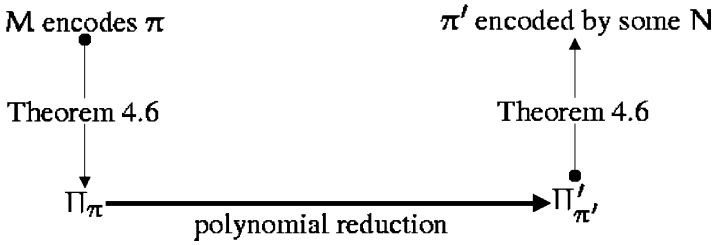

Fig. 27. How to use the functional terms to program.

$$(\lambda x_1 \otimes \ldots \otimes x_{m \geq 1}.M)M_1 \otimes \ldots \otimes M_m \ \rhd_\beta\ M[^{M_1}/_{x_1} \ldots {}^{M_m}/_{x_m}]$$
$$\bar{!}!M \ \rhd_!\ M$$
$$\bar{\S}\S M \ \rhd_\S\ M$$

Fig. 28. The rewriting relations for the functional syntax.

However, to keep things at a more intuitive level, we propose a detour with respect to the path in Figure 27, by introducing a dynamics directly on the functional language.

Figure 28 defines the basic rewriting relations on Λ.

The first relation is the trivial generalization of the β-rule of λ-Calculus to abstractions that bind patterns that represent tuples of variables. The α-equivalence must be used to avoid variable clashes when rewriting terms. The second rewriting relation *merges* the borders of two boxes.

Finally, define the rewriting system $\rightsquigarrow$ as the contextual closure on Λ of the rewriting relations in Figure 28. Its reflexive, and transitive closure is $\rightsquigarrow^*$. The

$$(c)\ \dfrac{(\S)\ \dfrac{x_1 : !\alpha, x_2 : !\alpha \vdash K\ x_1\ x_2 : !\alpha}{w_1 : !!\alpha, w_2 : !!\alpha \vdash \S(K\ \bar{\S} w_1\ \bar{\S} w_2) : \S!\alpha}}{z : !!\alpha \vdash \S(K\ \bar{\S} z\ \bar{\S} z) : \S!\alpha}$$

$$(\S)\ \dfrac{(c)\ \dfrac{x_1 : !\alpha, x_2 : !\alpha \vdash K\ x_1\ x_2 : !\alpha}{w : !\alpha \vdash K\ w\ w : !\alpha}}{z : !!\alpha \vdash \S(K\ \bar{\S} z\ \bar{\S} z) : \S!\alpha}$$

Fig. 29. Two derivations for the same term.

$$\begin{aligned}
\mathbf{Int} &= \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)\\
\bar{0} &= \lambda x.\S\lambda y.y : \mathbf{Int}\\
\bar{n} &= \lambda x.\S(\lambda y.\underbrace{\bar{!}x(\ldots(\bar{!}x}_{n}\ y)\ldots)) : \mathbf{Int}\ .
\end{aligned}$$

Fig. 30. The tally integers.

pair $(\Lambda, \rightsquigarrow)$ is the functional language we shall use to show the expressive power of **ILAL**. We shall generally adopt an abuse of notation by referring to such a language only with Λ.

7.3 Comments on the Concrete Syntax

We are going to use Λ to encode the **PolyTime** Turing machines in **ILAL**. As, to develop examples, we shall compute on the terms of Λ, we insist recalling that:

> *the functional language we are going to introduce is by no means a substitute for the proof-nets.*

Indeed, the mismatch between the normalization of the single-side sequent calculus and the proof-nets, as outlined in both Figure 18 and Figure 19, still exists, when adopting the functional language in place of the single-side sequent calculus. However, the mismatch is limited to the parts of the programs that get erased by the computation. So, we can safely keep observing the transformations of the functional terms, under $\rightsquigarrow$, to get the main intuitions about how the polynomially costing computation effectively develop on the proof-nets. However, we insist saying that all those who want to appreciate the details must program directly with the proof-nets.

Moreover, we have to pay for the notational economy of Λ, that, for example, represents the logical contraction by multiple occurrences of the same variable. This introduces an ambiguous representation of **ILAL** by means of Λ. For example, see Figure 29. The same term $\S(K\ \bar{\S}z\ \bar{\S}z)$ "encodes" two radically different derivations of the sequent calculus, which is equivalent to saying that $\S(K\ \bar{\S}z\ \bar{\S}z)$ "encodes" two structurally different proof-nets.

Such an ambiguity is *not* an issue for us, since, once more, the concrete syntax is not meant to be a real calculus, as outlined in Figure 27. We only need to agree about the translation from Λ to the nets. We choose the one putting the contractions as deeply as possible. This choice reduces the computational complexity of the translation.

$$
\begin{aligned}
\mathit{succ} &= \lambda zx.\S(\lambda y.\bar{!}x(\,\bar{\S}(z\,x)\,y)) : \mathbf{Int} \multimap \mathbf{Int}\\
\mathit{sum} &= \lambda wzx.\S(\lambda y.\,\bar{\S}(w\,x)(\,\bar{\S}(z\,x)\,y)) : \mathbf{Int} \multimap \mathbf{Int} \multimap \mathbf{Int}\\
\mathit{iter} &= \lambda xyz.\S(\,\bar{\S}(x\,y)\;\bar{\S}z) : \mathbf{Int} \multimap\, !(A \multimap A) \multimap \S A \multimap \S A\\
\mathit{mult} &= \lambda xy.\mathit{iter}\;x\;!(\lambda w.\mathit{sum}\;\bar{!}y\;w)\;\S\bar{0} : \mathbf{Int} \multimap\, !\mathbf{Int} \multimap \S\mathbf{Int}\\
\mathit{coerc} &= \lambda x.\S(\,\bar{\S}(x\;!\mathit{succ})\;\bar{0}) : \mathbf{Int} \multimap \S\mathbf{Int}\ .
\end{aligned}
$$

Fig. 31. Some combinators on the tally integers.

$$
\begin{aligned}
&\mathit{succ}\;\overline{n}\\
&\rightsquigarrow \lambda x.\S(\lambda y.\bar{!}x(\,\bar{\S}(\overline{n}\,x)\,y))\\
&\rightsquigarrow \lambda x.\S(\lambda y.\bar{!}x(\,\bar{\S}\S(\lambda w.\overbrace{\bar{!}x(\ldots(\bar{!}x}^{n}\;w)\ldots))\,y))\\
&\rightsquigarrow \lambda x.\S(\lambda y.\bar{!}x((\lambda w.\bar{!}x(\ldots(\bar{!}x\,w)\ldots))\,y))\\
&\rightsquigarrow \lambda x.\S(\lambda y.\underbrace{\bar{!}x(\bar{!}x(\ldots(\bar{!}x}_{n+1}\;y)\ldots)))\\
&= \overline{n+1}\ .
\end{aligned}
$$

Fig. 32. Calculating the numeral next to $\bar{n}$.

8. ENCODING A NUMERICAL SYSTEM

The numerical system adopted on Λ is the analogous of Church numerals for λ-Calculus.

The type and the terms of the *tally* integers are in Figure 30. Observe that there is a translation from Λ to λ-Calculus that, applied to $\bar{0}$ and $\bar{n}$, yields λ-Calculus Church numerals:

$$\lambda f x.\,\underbrace{f(\ldots(f}_{n\geq 0}\;x)\ldots)).$$

The translation just erases all the occurrences of $!$, $\S$, $\bar{!}$, and $\bar{\S}$.

Figure 31 introduces some further combinators on the numerals. To support intuition about how the computation develops, the numeral next to $\bar{n}$ can be calculated as in Figure 32. *sum* adds two numerals. *iter* takes as arguments a numeral, a *step* function, and a *base* where to start the iteration from. Observe that *iter* $\bar{2}$ $!\bar{n}$ $\S\bar{0}$ cannot have type, for any numeral $\bar{n}$. This is because the step function is required to have identical domain and co-domain. This should not surprise. Taking the λ-Calculus Church numeral $\bar{2}$, and applying it to itself, we get an exponentially costing computation.

mult is defined as an iterated sum, for multiplying two numerals.

Finally, *coerc*(ion) embeds a numeral into a $\S$-box, preserving its value. See Figure 33 for an example.

8.1 Encoding a Predecessor

The predecessor of the numerical system for Λ is an instance of a general computation scheme that iterates the template function in Figure 34. $\mathcal{T}$ takes a pair of functions h, g as arguments, and has f as its parameter. If

$$
\begin{aligned}
&\mathit{coerc}\ \overline{n}\\
&\rightsquigarrow\ \S(\,\bar{\S}(\overline{n}\,!\mathit{succ})\,\overline{0})\\
&\rightsquigarrow\ \S(\,\bar{\S}(\S(\lambda w.\bar{!}!\mathit{succ}(\ldots(\bar{!}!\mathit{succ}\,w)\ldots)))\,\overline{0})\\
&\rightsquigarrow\ \S((\lambda w.\mathit{succ}(\ldots(\mathit{succ}\,w)\ldots)))\,\overline{0})\\
&\rightsquigarrow\ \S(\overbrace{\mathit{succ}(\ldots(\mathit{succ}}^{n}\,\overline{0})\ldots))\\
&\rightsquigarrow^{*}\ \S\overline{n}\ .
\end{aligned}
$$

Fig. 33. Coercion of $\bar{n}$ to $\S\bar{n}$.

$$\mathcal{T}_f(g,h)\ =\ (f,gh)\ .$$

Fig. 34. Template function for the predecessor.

$$\overbrace{\mathcal{T}_f(\ldots\mathcal{T}_f}^{n}(gh)\ldots)\ =\ (f,\overbrace{f(\ldots f}^{n-1}(gh)\ldots))$$

Fig. 35. Iterating the template function.

$$
\begin{aligned}
I\ &=\ \lambda x.x : \forall\alpha.\alpha \multimap \alpha\\
\pi_2\ &=\ \lambda x \otimes y.y : \forall\alpha.\alpha \otimes \alpha \multimap \alpha\\
T\ &=\ \lambda f.\lambda g \otimes h.(f \otimes (gh)) : \forall\alpha.(\alpha \multimap \alpha) \multimap ((\alpha \multimap \alpha) \otimes \alpha) \multimap (\alpha \multimap \alpha) \otimes \alpha\\
\mathit{step}\ z\ &=\ T\ z : ((\mathbf{Int} \multimap \mathbf{Int}) \otimes \mathbf{Int}) \multimap (\mathbf{Int} \multimap \mathbf{Int}) \otimes \mathbf{Int}\\
\mathit{base}\ y\ &=\ T\ I\ (I \otimes y) : (\mathbf{Int} \multimap \mathbf{Int}) \otimes \mathbf{Int}\\
\mathit{pred}\ &=\ \lambda w x.\S(\lambda y.\pi_2(\,\bar{\S}(w\,!(\mathit{step}\,\bar{!}x))(\mathit{base}\ y))) : \mathbf{Int} \multimap \mathbf{Int}\\
\text{where}\quad & y, z : \mathbf{Int}
\end{aligned}
$$

Fig. 36. The predecessor.

T	$\mathcal{T}$
$!(\mathit{step}\,\bar{!}x)$	f
I	g
$I \otimes y$	h

Fig. 37. Correspondence between $\mathcal{T}$ and T.

$h : X \to Y$, $g : Y \to Z$, and $f : Z \to Z$, for some domains X, Y, Z, then $\mathcal{T}_f$ can be iterated. An example of an n-fold iteration of $\mathcal{T}_f$ from (g, h) is in Figure 35, where it is simple to recognize the predecessor of n, if we let $f : \mathbb{N} \to \mathbb{N}$ be the identity, $g : \mathbb{N} \to \mathbb{N}$ be the successor, h be 0, and if we assume to erase the first component of the result. Recasting everything in Λ, we get the definitions in Figure 36.

The term, *pred*, iterates w times $!(\mathit{step}\,\bar{!}x)$ from $I \otimes y$, exploiting the correspondence between $\mathcal{T}$ and T in Figure 37. Observe also that our predecessor does not make any explicit use of the encoding of the additive types by means of the second-order quantification. In Asperti [1998], the predecessor has a somewhat more intricate form that we recall here:

$$\lambda n x y.(n\ (\lambda p.(U\ I\ x\ (p\ snd)))\ (U I I\ \ y)\ fst), \tag{2}$$

$$(\multimap_l)\ \frac{!\Gamma \vdash \mathcal{T}_f : !(A \multimap A) \qquad \Delta, y : \S(A \multimap A) \vdash M : B}{(\forall_l)\ \dfrac{!\Gamma, \Delta, n : (!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha))[^A/_\alpha] \vdash M[^{\bar{\S}(n\ \mathcal{T}_f)}/_y] : B}{!\Gamma, \Delta, \mathbf{Int} \vdash M[^{\bar{\S}(n\ \mathcal{T}_f)}/_y] : B}}$$

Fig. 38. General iteration scheme: the logical structure.

$$(\multimap_l)\ \frac{\Theta \vdash N : A \qquad x : A \vdash x : A}{(\S)\ \dfrac{\Theta, y : A \multimap A \vdash y\ N : A}{\S\Theta, w : \S(A \multimap A) \vdash \S(\ \bar{\S} w\ N) : \S A}}$$

Fig. 39. Getting the standard iteration scheme.

where:

$$\begin{aligned} U\ P\ Q\ R &= \lambda z.(z\ P\ Q\ R) \\ fst &= \lambda x\, y\, z.(x\ z) \\ snd &= \lambda x\, y\, z.(y\ z). \end{aligned}$$

pred is obtained by eliminating the nonessential components of (2) above.

Both *pred* and (2) are *syntactically linear*, and so also their complexity is readily linear. On the contrary, the usual encoding of the predecessor, that, using λ-Calculus syntax with pairs $\langle M, N\rangle$, is:

$$\lambda n x\, y\, .fst(n\ (\lambda p.\langle snd(p), x\ snd(p)\rangle)\ \langle y, y\rangle), \tag{3}$$

has also an exponential strategy. Such a strategy exists because the term is not syntactically linear. However, both *pred* and (2) witness that the nonlinearity of (3) is inessential. In particular, in Girard [1998], where Girard embeds (3) in **LLL**, the sub-term $\lambda p.\langle snd(p), x\ snd(p)\rangle$ here above has the *additive* type $(\alpha\&\alpha) \multimap (\alpha\&\alpha)$. This means that, at every step of the iteration $n\ (\lambda p.\langle snd(p), x\ snd(p)\rangle)\ \langle y, y\rangle$, only one of the multiple uses of p is effectively useful to produce the result.

Remark 8.1

—The procedural iteration scheme in Figure 35, our predecessor is an instance of, was already used in Roversi [1998]. However, by reading Danos and Joinet [1999], we saw that the iteration in Figure 35 actually "implements" a general logical iteration scheme, which we adapt to **ILAL** in Figure 38. There, the term M must contain $g\ h$, the argument of $n\,\mathcal{T}_f$. The more traditional iteration scheme can be obtained from Figure 38 by letting $!\Delta, y : \S(A \multimap A) \vdash M : B$ be the conclusion of the derivation in Figure 39. Observe that the instance of M we use for our predecessor is not as simple as $\S(\ \bar{\S} w\ N)$.

—We want to discuss a little more about the linearity of the additive structures. Not sticking to any particular notation, let $fxy = fst\langle x\, y, x\, y\rangle$. The function f is just the identity, and it would get a linear type in **ILAL**. Now, consider an n-fold iteration of f by means of a Church numeral $\bar{n}$. Then, let us apply the result to a pair of identities. We have just defined $g_f\ n = ((n\ f)\ I)\ I$. This term is typable in **ILAL**. So, in **ILAL**, $g_f\ n$ normalizes with a polynomial (in fact linear) cost. However, try to reduce $g_f\ n$ in most traditional lazy

$$\vec{y}_3 = \begin{array}{ccc} y_0^1 & & \\ y_0^2 & y_1^2 & \\ y_0^3 & y_1^3 & y_2^3 \end{array}$$

Fig. 40. Vector of vectors of variables.

call-by-value implementations of functional languages, like SML, CAML, Scheme, etc. and you will discover that the reduction has an exponential cost. So, first, if a usual λ-term M can be embedded in **ILAL**, then, in general, it is *not* true that M normalizes with a polynomial cost under *any* reduction strategy. We only know that *there exists* an effective way to normalize M with a polynomial cost. The polynomial reduction, in general, is *not compatible* with the *lazy call-by-value reduction*.

However, consider again $g_f\ n = ((n\ f)\ I)\ I$. Its lazy call-by-name evaluation has a linear cost. We leave the following open question: is it true that, taking a typable term M, having a polynomially costing reduction strategy, then that strategy can be the lazy call-by-name?

9. ENCODING THE POLYNOMIALS

In this section, we show how to encode the elements of $\mathcal{P}$, that is, the polynomials with positive degrees, and positive coefficients, as terms of Λ. This encoding is based on the numerical system of Section 8. It will serve to represent and simulate all **PolyTime** Turing machines using the terms of **ILAL**.

We use p_x^ϑ to range over the polynomials $\sum_{i=0}^{\vartheta} a_i x^i \in \mathcal{P}$ with maximal nonnull degree ϑ, and indeterminate x.

The result of this section is:

THEOREM 9.1. *There is a translation* $\hat{}\ : \mathcal{P} \to \Lambda$, *such that, for any* $p_x^\vartheta \in \mathcal{P}$:

—$\hat{p}_x^\vartheta$: **Int** $\multimap \S^{\vartheta+3}$**Int**, *and*
—$p_n^\vartheta = m$, *if, and only if,* $\hat{p}_{\bar{n}}^\vartheta \rightsquigarrow^* \S^{\vartheta+3}\bar{m}$.

In the following, we develop the proof of the theorem, and an example about how the encoding works.

First of all, some useful notations.

Let p_x^ϑ be the polynomial $\sum_{i=0}^{\vartheta} a_i x^i$ describing the computational bound of the Turing machine being encoded. Let $\kappa = \frac{\vartheta(\vartheta+1)}{2}$.

Abbreviate with $\vec{y}_n$ an n-long vector of all vectors with length 1 through n, each containing variables $y_i^j \in \mathtt{T_{variables}}$, where $0 \le i \le n-1$, and $1 \le j \le n$. Figure 40 gives $\vec{y}_3$ as an example. As usual, $\vec{y}_3[i][j]$ picks y_i^j out of the vector $\vec{y}$.

Figure 41, where $p, q \ge 0$, and $n \ge 1$, introduces both a type abbreviation, and some generalizations of the operations on Church numerals in Section 8.

Figure 42 encodes the polynomial p_x^ϑ, on which we can note some simple facts. $tuple_n$ makes n copies of the numeral it is applied to. Every "macro" $\langle\!\langle a_i \cdot x^i \rangle\!\rangle_{\vec{y}_\vartheta[i]}$ represents the factor $a_i x^i$ so that x^i is a product of as many variables of $\mathtt{T_{variables}}$ as the degree i. The coercion applied to each of them just adds as many $\S$-boxes as necessary to have all the arguments of $sum_{\vartheta+1}^{\vartheta+2}$ at the same depth $\vartheta + 2$.

$$
\begin{array}{rcl}
\mathbf{Int}_n & = & \mathbf{Int} \otimes \ldots \otimes \mathbf{Int} \qquad \text{with } n \text{ components} \\
\overline{0}_n & = & \overline{0} \otimes \ldots \otimes \overline{0} \qquad \text{with } n \text{ components} \\
\Diamond^n M & = & \underbrace{\Diamond(\ldots(\Diamond}_{n} M)\ldots) \qquad \text{with } \Diamond \in \{!, \S, \bar{!}, \bar{\S}\} \\
\overline{0}^{p,q} & = & \S^p !^q \overline{0} : \S^p !^q \mathbf{Int} \\
sum_n & = & \lambda x_1 \otimes \ldots \otimes x_n z.\S(\lambda y.\, \bar{\S}(x_1\ z)(\ldots(\,\bar{\S}(x_n\ z)\ y)\ldots)) : \mathbf{Int}_n \multimap \mathbf{Int} \\
sum_n^p & = & \lambda x_1 \otimes \ldots \otimes x_n.\S^p(sum_n\ \bar{\S}^p x_1 \otimes \ldots \otimes \bar{\S}^p x_n) : (\S^p \mathbf{Int})_n \multimap \S^p \mathbf{Int} \\
succ^{p,q} & = & \lambda x.\S^p(!^q(succ\ \bar{!}^q(\,\bar{\S}^p x))) : \S^p !^q \mathbf{Int} \multimap \S^p !^q \mathbf{Int} \\
coerc^{p,q} & = & \lambda x.\S(\,\bar{\S}(x\ !succ^{p,q})\ \overline{0}^{p,q}) : \mathbf{Int} \multimap \S^{p+1} !^q \mathbf{Int} \\
mult^p & = & \lambda x y.\S^p(mult\ \bar{\S}^p x\ \bar{\S}^p y) : \S^p \mathbf{Int} \multimap \S^p !\mathbf{Int} \multimap \S^{p+1} \mathbf{Int} \\
tuple_n & = & \lambda x.\S(\,\bar{\S}(x\ !(\lambda x_1 \otimes \ldots \otimes x_n.succ\ x_1 \otimes \ldots \otimes succ\ x_n))\ \overline{0}_n) : \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{Int} \multimap \S(\mathbf{Int}_n)
\end{array}
$$

Fig. 41. Generalizations of operations on the numerals.

$$
\begin{array}{rcl}
\hat{p}_x^\vartheta & = & \lambda x.\S((\lambda\, y_0^1 \\
 & & \qquad \otimes \ldots \otimes \\
 & & \qquad y_0^i \otimes \ldots \otimes y_{i-1}^i \\
 & & \qquad \otimes \ldots \otimes \\
 & & \qquad y_0^\vartheta \otimes \ldots \otimes y_{\vartheta-1}^\vartheta \cdot \\
 & & \quad sum_{\vartheta+1}^{\vartheta+2}\ \S^1(coerc^{\vartheta,0}\ \bar{\S}^1 \langle\!\langle a_0 x^0 \rangle\!\rangle_{\vec{y}_\vartheta[0]}) \\
 & & \qquad\qquad \vdots \\
 & & \qquad \otimes \S^{i+1}(coerc^{\vartheta-i,0}\ \bar{\S}^{i+1} \langle\!\langle a_i x^i \rangle\!\rangle_{\vec{y}_\vartheta[i]}) \\
 & & \qquad\qquad \vdots \\
 & & \qquad \otimes \S^{\vartheta+1}(coerc^{0,0}\ \bar{\S}^{\vartheta+1} \langle\!\langle a_\vartheta x^\vartheta \rangle\!\rangle_{\vec{y}_\vartheta[\vartheta]}) \\
 & &)\ \bar{\S}(tuple_\kappa\ x)) : \mathbf{Int} \multimap \S^{\vartheta+3} \mathbf{Int}
\end{array}
$$

where:

$$
\begin{array}{rcll}
\langle\!\langle a x^0 \rangle\!\rangle_{\vec{z}} & \mapsto & coerc^{0,0}\ \overline{a} : \S\mathbf{Int} & \\
\langle\!\langle a x^n \rangle\!\rangle_{\vec{z}} & \mapsto & mult^n\ \langle \vec{z}, n-1 \rangle\ (coerc^{n-1,1}\ \overline{a}) : \S^{n+1} \mathbf{Int} & (n \geq 1)
\end{array}
$$

$$
\begin{array}{rcll}
\langle \vec{z}, 0 \rangle & \mapsto & coerc^{0,0}\ \vec{z}[0] : \S\mathbf{Int} & \\
\langle \vec{z}, n \rangle & \mapsto & mult^n\ \langle \vec{z}, n-1 \rangle\ (coerc^{n-1,1}\ \vec{z}[n]) : \S^{n+1} \mathbf{Int} & (n \geq 1)
\end{array}
$$

Fig. 42. Encoding of the polynomial.

We conclude this section with an example. Figure 43 fully develops the encoding of the polynomial x^2+1. Assume we want to evaluate $\hat{p}_{\bar{2}}^2$, from which we expect $\S^5\bar{5}$. Figure 44 gives the main intermediate steps to get to such a result.

10. EXPRESSIVE POWER OF **ILAL**

We are now in the position to show the expressive power of **ILAL** by encoding **PolyTime** Turing machines in Λ. We shall establish some notations

$$
\begin{aligned}
\hat{p}_2^2 \;=\; & \S((\lambda y_0^1 \otimes y_0^2 \otimes y_1^2 . \\
& \quad sum_3^4\ \S^1(coerc^{2,0}\ \bar{\S}^1\langle\langle 1 \cdot x^0\rangle\rangle_{\vec{y}_2[0]}) \\
& \quad \otimes \\
& \quad \S^2(coerc^{1,0}\ \bar{\S}^2\langle\langle 0 \cdot x^1\rangle\rangle_{\vec{y}_2[1]}) \\
& \quad \otimes \\
& \quad \S^3(coerc^{0,0}\ \bar{\S}^3\langle\langle 1 \cdot x^2\rangle\rangle_{\vec{y}_2[2]}) \\
&)\ \bar{\S}(tuple_3\ \bar{2})) : \mathbf{Int} \multimap \S^5\mathbf{Int}
\end{aligned}
$$

where:

$$
\begin{aligned}
\langle\langle 1 \cdot x^0\rangle\rangle_{\vec{y}_2[0]} &= coerc^{0,0}\ \bar{1} : \S\mathbf{Int} \\
\langle\langle 0 \cdot x^1\rangle\rangle_{\vec{y}_2[1]} &= mult^1\ (coerc^{0,0}\ \vec{y}_2[0][1])\ (coerc^{0,1}\ \bar{0}) : \S^2\mathbf{Int} \\
\langle\langle 1 \cdot x^2\rangle\rangle_{\vec{y}_2[2]} &= mult^2\ (mult^1\ (coerc^{0,0}\ \vec{y}_2[0][2]) \\
&\qquad (coerc^{0,1}\ \vec{y}_2[1][2]) \\
&\quad)\ (coerc^{1,1}\ \bar{1}) : \S^3\mathbf{Int}\ ,
\end{aligned}
$$

and

$$
\begin{aligned}
\vec{y}_2[0][1] &= y_0^1 \\
\vec{y}_2[0][2] &= y_0^2 \\
\vec{y}_2[1][2] &= y_1^2
\end{aligned}
$$

Fig. 43. Encoding the polynomial $x^2 + 1$.

$$
\begin{aligned}
\bar{\S}(tuple_3\ \bar{2}) &= \bar{2}\otimes\bar{2}\otimes\bar{2} \\
\bar{\S}^1(coerc^{0,0}\ \bar{1}) &= \bar{1} \\
\bar{\S}^2(mult^1\ (coerc^{0,0}\ \bar{2})\ (coerc^{0,1}\ \bar{0})) &= \bar{0} \\
\bar{\S}^3(mult^2\ (mult^1\ (coerc^{0,0}\ \bar{2}) & \\
\qquad (coerc^{0,1}\ \bar{2}) & \\
\quad)\ (coerc^{1,1}\ \bar{1})) &= \bar{4} \\
\S^1(coerc^{2,0}\ \bar{1}) &= \S^4\bar{1} \\
\S^2(coerc^{1,0}\ \bar{0}) &= \S^4\bar{0} \\
\S^3(coerc^{0,0}\ \bar{4}) &= \S^4\bar{4} \\
\S(sum_3^4\ (\S^4\bar{1} \otimes \S^4\bar{0} \otimes \S^4\bar{4})) &= \S^5\bar{5}
\end{aligned}
$$

Fig. 44. Intermediate evaluation steps of $\hat{p}_2^2$.

together with some simplifying, but not restricting, assumptions on the class of **PolyTime** Turing machines we want to encode.

Every machine we are interested in is a tuple $\langle \mathcal{S}, \Sigma \cup \{\star, \bot, \top\}, \delta, \mathsf{s}_0, \mathsf{s}_a\rangle$, where:

—$\mathcal{S}$ is the set of states with cardinality $|\mathcal{S}|$,

—Σ is the *input alphabet*,

—$\star, \bot, \top \notin \Sigma$ are "*blank*" symbols,

—$\Sigma \cup \{\star, \bot, \top\}$ is the *tape alphabet*,

—δ is the *transition function*,
—s_0 is the *starting state*, and
—s_a is the *accepting state*.

In general, we shall use s to range over $\mathcal{S}$.

The transition function has type $\delta\colon (\Sigma \cup \{\star, \bot, \top\}) \times \mathcal{S} \to (\Sigma \cup \{\star, \bot, \top\}) \times \mathcal{S} \times \{L, R\}$, where $\{L, R\}$ is the set of directions the head can move.

Both $\bot$ and $\top$ are special "blank" symbols. They delimit the leftmost and the rightmost tape edge. This means that we only consider machines with a finite tape that, however, can be extended at will. For example, suppose the head of the machine is reading $\top$, that is, the rightmost limit of the tape. Assume also the head needs to move rightward, and that, before moving, it needs to write the symbol 1 on the tape. Since the head is on the edge of the tape, the *control* of the machine firstly writes 1 for $\top$, then adds a new $\top$ to the right of 1, and, finally, it shifts the head one place to its right, so placing the head on the just added $\top$. The same can happen to $\bot$ when the head is on the leftmost edge of the tape.

Obviously, the machines whose finite tape can be extended at will are perfectly equivalent to those that, by assumption, have infinite tape. These latter have a control that does not require to recognize the borders of the tape, in order to extend it, when necessary.

Taking only machines with finite tape greatly simplifies our encoding, because Λ contains only finite terms.

Recall now that we want to encode ***PolyTime*** Turing machines. For this reason, we require that every machine come with a polynomial p_x^{ϑ}, with maximal nonnull degree ϑ. The polynomial characterizes the maximal running time. So, every **PolyTime** machine accepts an input of *length* l if, after at most p_l^{ϑ} steps, it enters state s_a. Otherwise, it rejects the input.

Without loss of generality, we add some further simplifying assumptions. First, whenever the machine is ready to accept the input, before entering s_a, it shifts its head to the leftmost tape character, different from $\bot$. We agree that the output is the portion of tape from $\bot$, excluded, through the first occurrence of $\star$ to its right. (Of course, for any **PolyTime** Turing machine, there is one behaving like this with a polynomial overhead.) Second, we limit ourselves to **PolyTime** Turing machines with $\Sigma = \{0, 1\}$.

Definition 10.1. $\mathcal{T}^{\vartheta}_{\mathbf{PolyTime}}$ is the set of all PTime Turing machines, described here above.

The next sections introduce the parts of the encoding of a generic **PolyTime** Turing machine, using an instance of Λ, built from the set of variable names $\mathsf{T}_{\mathtt{variables}} = \{0, 1, \star, \bot, \top\}$. Namely, we use the symbols of the tape alphabet directly as *variable* names for the term of the encoding. We hope this choice will produce a clearer encoding. We shall try to give as much intuition as possible as the development of the encoding proceeds. However, some details will become clear only at the end, when all the components will be assembled together.

10.1 States

Recall that the set of states $\mathcal{S}$ has cardinality $|\mathcal{S}|$. Assume to enumerate $\mathcal{S}$. The i^{th} state is:

$$state_i \;=\; \lambda x_0 \otimes \cdots \otimes x_{|\mathcal{S}|-1} \otimes v.x_i\; v \qquad \text{with } 0 \leq i \leq |\mathcal{S}| - 1,$$

which has type:

$$\mathbf{state} = \forall \alpha\beta.(\overbrace{(\alpha \multimap \beta) \otimes \cdots \otimes (\alpha \multimap \beta)}^{|\mathcal{S}| \text{ times}} \otimes \alpha) \multimap \beta.$$

Every $state_i$ extracts a row from an array that, as we shall see, encodes the translation $\hat{\delta}$ of δ. So, every x_i stands for the i^{th} row of $\hat{\delta}$ which *must be* a closed term. The parameter v stands for the variables that the rows of $\hat{\delta}$ would *share* in case they *were not closed* terms. The point here is that the sharing is *additive* and not *exponential*. We can understand the difference by assuming to apply $state_i$ on a $\hat{\delta}$ with two rows R_1 and R_2. Once all the encoding will be complete, we shall see that, as the computation proceeds, for every instance of $\hat{\delta}$ that the computation generates, *only one* between R_1, R_2 is used. The other gets discarded. This has some interesting consequences on the form of $\hat{\delta}$ itself, if R_1, R_2 share some variables. Indeed, assume $x_1, \ldots, x_n$ be *all* the free variables, with *linear* types, *common* to R_1, R_2. Then $R_1 \otimes R_2$ can not be typed as it is: every x_j would require an *exponential* type, contrasting with the effective use of every x_j we are going to do: since we assume to use *either* R_1, *or* R_2, every x_j is eventually used linearly. For this reason, our instance of $\hat{\delta}$ is represented as the triple:

$$(\lambda x_1 \otimes \cdots \otimes x_n.R_1) \otimes (\lambda x_1 \otimes \cdots \otimes x_n.R_2) \otimes (x_1 \otimes \cdots \otimes x_n).$$

The leftmost component is extracted by means of $state_0$ that applies $\lambda x_1 \otimes \cdots \otimes x_n.R_1$ to $x_1 \otimes \cdots \otimes x_n$. The rightmost component is obtained analogously, by applying $state_1$ to $\lambda x_1 \otimes \cdots \otimes x_n.R_2$ to $x_1 \otimes \cdots \otimes x_n$. Giving linear types to the free variables of the rows in $\hat{\delta}$ allows their efficient, in fact *linear*, use.

10.2 Configurations

Each of them stands for the position of the head on an instance of tape, in some state. We choose the following term scheme to encode the configurations of **PolyTime** Turing machines:

$$\begin{aligned} config \;=\; & \lambda 01 \star \bot \top. \\ & \S(\lambda x x'.(\bar{!}\chi_1(\cdots(\bar{!}\chi_p(\bar{!}\bot\; x))\cdots)) \otimes (\bar{!}\chi'_1(\cdots(\bar{!}\chi'_q(\bar{!}\top\; x'))\cdots)) \otimes state_i), \end{aligned}$$

where $\chi_{1 \leq i \leq p}, \chi'_{1 \leq j \leq q} \in \{0, 1, \star\}$, with $p, q \geq 0$. Every *config* has type:

$$\begin{aligned} \mathbf{config} \;=\; & \forall \alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \\ & !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \\ & !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha \multimap (\alpha \otimes \alpha \otimes \mathbf{state})). \end{aligned}$$

As an example, take the following tape:

$$\bot \star 1 \top. \tag{4}$$

$$\begin{aligned}
\Pi_0 &= \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top \otimes x \otimes \nu.0\,\nu \\
\Pi_1 &= \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top \otimes x \otimes \nu.1\,\nu \\
\Pi_\star &= \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top \otimes x \otimes \nu.\star\,\nu \\
\Pi_\bot &= \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top \otimes x \otimes \nu.\bot\,\nu \\
\Pi_\top &= \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top \otimes x \otimes \nu.\top\,\nu \\
\Pi_\emptyset &= \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top \otimes x \otimes \nu.x\,\nu
\end{aligned}$$

Fig. 45. Projections representing the tape alphabet symbols.

Assume that the head is reading $\bot$, and that the actual state is s_i. Its encoding is:

$$\lambda 01 \star \bot \top.\S(\lambda x x'.x \otimes (\bar{!}\bot(\bar{!}\star(\bar{!}1(\bar{!}\top\, x')))) \otimes \mathit{state}_i). \tag{5}$$

The leftmost component of the tensor in the body of the λ-abstraction is the part of the tape to the left of the head, also called *left tape*. It is encoded in reversed order. The cell read by the head, and the part of the tape to its right, the *right tape*, is the central component of the tensor.

Any *starting configuration* has form:

$$\lambda 01 \star \bot \top.\S(\lambda x x'.(\bar{!}\bot\, x) \otimes (\bar{!}\chi_1(\ldots(\bar{!}\chi_q(\bar{!}\top\, x'))\ldots)) \otimes \mathit{state}_0),$$

where every χ_j ranges over $\{0, 1\}$, and state_0 encodes s_0. Namely, the tape has only characters of the input alphabet on it, the head is on its leftmost input symbol, the left part of the tape is empty, and the only reasonable state is the initial one.

10.3 Transition Function

The transition function δ is represented by the term $\hat{\delta}$, which is (almost) the obvious encoding of an array in a functional language. So, $\hat{\delta}$ is (essentially) a tuple of tuples. Every term representing a state can project a row out of $\hat{\delta}$. We have already seen the encoding of the states in Section 10.1. Since then, we know that every state_i needs as argument the set of variables additively shared by the components of the array it is applied to. So, $\hat{\delta}$ contains these variables as $(|\mathcal{S}| + 1)^{\text{th}}$ row. A column of a row is extracted thanks to the projections in Figure 45. The name of each projection obviously recalls the tape symbol to which it is associated. Every projection has type:

$$\begin{aligned}
\mathbf{proj}_{\alpha,\beta} = ((\alpha \multimap \beta) \otimes (\alpha \multimap \beta) \otimes (\alpha \multimap \beta) \otimes (\alpha \multimap \beta) \\
\otimes (\alpha \multimap \beta) \otimes (\alpha \multimap \beta) \otimes \alpha) \multimap \beta.
\end{aligned}$$

The transition function is in Figure 46. For example, we can extract the element $Q_{i,\star}$ from $\hat{\delta}$, by evaluating:

$$\Pi_\star\ (\mathit{state}_i\ (\hat{\delta}\ (0 \otimes 1 \otimes \star \otimes \bot \otimes \top))).$$

Finally, the terms $Q_{i,j}$, as expected, produce a triple in the codomain of the translation $\hat{\delta}$ of δ. Figure 47 defines 15 terms to encode the triples we need. The triples are somewhat hidden in the structure of these terms. However, such terms have the most natural form we found, once we choose to manipulate the configurations of Section 10.2.

$$
\begin{aligned}
\widehat{\delta} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& (\lambda x . Q_{0,0} \otimes Q_{0,1} \otimes Q_{0,\star} \otimes Q_{0,\bot} \otimes Q_{0,\top} \otimes Q_{0,\emptyset} \otimes x) \quad \otimes \\
& \vdots \\
& (\lambda x . Q_{|\mathcal{S}|-1,0} \otimes Q_{|\mathcal{S}|-1,1} \otimes Q_{|\mathcal{S}|-1,\star} \otimes Q_{|\mathcal{S}|-1,\bot} \otimes Q_{|\mathcal{S}|-1,\top} \otimes Q_{|\mathcal{S}|-1,\emptyset} \otimes x) \otimes \\
& 0 \otimes 1 \otimes \star \otimes \bot \otimes \top
\end{aligned}
$$

Fig. 46. Encoding the transition function δ.

$$
\begin{aligned}
left^{0}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . t_l \otimes (h_l \; (0 \; t_r)) \otimes state_{ij} \\
left^{1}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . t_l \otimes (h_l \; (1 \; t_r)) \otimes state_{ij} \\
left^{\star}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . t_l \otimes (h_l \; (\star \; t_r)) \otimes state_{ij} \\
left^{0}_{\bot ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . t_l \otimes (\bot \; (0 \; t_r)) \otimes state_{ij} \\
left^{1}_{\bot ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . t_l \otimes (\bot \; (1 \; t_r)) \otimes state_{ij} \\
left^{\star}_{\bot ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . t_l \otimes (\bot \; (\star \; t_r)) \otimes state_{ij} \\
right^{0}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . 0 \; (h_l \; t_l) \otimes t_r \otimes state_{ij} \\
right^{1}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . 1 \; (h_l \; t_l) \otimes t_r \otimes state_{ij} \\
right^{\star}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . \star \; (h_l \; t_l) \otimes t_r \otimes state_{ij} \\
right^{0}_{\top ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . 0 \; (h_l \; t_l) \otimes (\top \; t_r) \otimes state_{ij} \\
right^{1}_{\top ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . 1 \; (h_l \; t_l) \otimes (\top \; t_r) \otimes state_{ij} \\
right^{\star}_{\top ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . \star \; (h_l \; t_l) \otimes (\top \; t_r) \otimes state_{ij} \\
stay^{0}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . (h_l \; t_l) \otimes (0 \; t_r) \otimes state_{a} \\
stay^{1}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . (h_l \; t_l) \otimes (1 \; t_r) \otimes state_{a} \\
stay^{\star}_{ij} \;=\; & \lambda 0 \otimes 1 \otimes \star \otimes \bot \otimes \top . \\
& \lambda h_l t_l t_r . (h_l \; t_l) \otimes (\star \; t_r) \otimes state_{a} \; .
\end{aligned}
$$

Fig. 47. The output triples of $\hat{\delta}$.

$$
\begin{array}{rcl}
\multimap_\alpha & = & (\alpha \multimap \alpha) \\
\otimes_\alpha & = & \multimap_\alpha \otimes \multimap_\alpha \otimes \multimap_\alpha \otimes \multimap_\alpha \otimes \multimap_\alpha \\
\tau_\alpha & = & (\multimap_\alpha) \multimap \alpha \multimap \alpha \multimap (\alpha \otimes \alpha \otimes \mathbf{state}) \\
\mathbf{row}_\alpha & = & \otimes_\alpha \multimap (\mathbf{shift}_\alpha \otimes \mathbf{shift}_\alpha \otimes \mathbf{shift}_\alpha \otimes \mathbf{shift}_\alpha \otimes \mathbf{shift}_\alpha \otimes \mathbf{shift}_\alpha \otimes (\otimes_\alpha)) \\
\mathbf{shift}_\alpha & = & \otimes_\alpha \multimap \tau_\alpha \\
\widehat{\delta} & = & \forall\alpha.\otimes_\alpha \multimap (\underbrace{\mathbf{row}_\alpha \otimes \ldots \otimes \mathbf{row}_\alpha}_{|\mathcal{S}|} \otimes (\otimes_\alpha)) \\
\\
\mathit{left}^{\chi}_{\chi'} & : & \mathbf{shift}_\alpha \\
\mathit{right}^{\chi}_{\chi'} & : & \mathbf{shift}_\alpha \\
\mathit{stay}^{\chi}_{\chi'} & : & \mathbf{shift}_\alpha \quad \text{with } \chi \in \{0, 1, \star\} \text{ and } \chi' \in \{\bot ij, ij\} \\
\widehat{\delta} & : & \widehat{\delta} \ .
\end{array}
$$

Fig. 48. Typing for $\hat{\delta}$.

The first three "left" terms move the head from the top h_l of the left tape to the top of the right tape. This move comes after the head writes one of the symbols among $\{0, 1, \star\}$ on the tape. For example, if the written symbol is $\star$, the new right tape becomes $h_l(\star\ t_r)$. We recall that $\star$ (or 0, or 1) replaces the symbol read before the move. However, if the character on top of the right tape, before the move, was $\bot$, one of the last three "left" terms must be used, instead. They put under the head the symbol that signals the end of the tape.

The "shifting to the right" behaves almost, but not perfectly, symmetrically. The main motivation is that the head is assumed to read the top of the right tape. So, when it shifts to the right, only the new character that the head writes has to be placed on the left tape. If the head was reading $\top$ before the move, another $\top$ must be added after it. This is done by the last three "right" shifts. The last three terms are used in two ways. When the actual state of the encoded machine is s_a, the head cannot move anymore. This is exactly the effect of every "stay" term. For example, stay^1_{ij} must be used when we have to simulate a head reading 1 in the actual state s_a: the head must rewrite 1 without shifting. The "stay" are also used as dummy terms in the "$\emptyset$-column" of $\hat{\delta}$. The elements of that column will never be used because they correspond to the move directions when the head is beyond the tape delimiters $\bot$ and $\top$. But this can never happen.

Of course, the choice of which term in Figure 47 we have to use as $Q_{i,j}$ in $\hat{\delta}$ must be coherent with the behavior of δ that we want to simulate. We shall see an explicit example about this later.

Figure 48 gives useful hints to those who want to check the well typing of $\hat{\delta}$. It may help also saying that, once the whole encoding will be set up, the projections Π_0, Π_1, $\Pi_\star$, $\Pi_\bot$, $\Pi_\top$, and $\Pi_\emptyset$ will be used in $\hat{\delta}$ with the type instantiated as $\mathbf{proj}_{\otimes_\alpha, \tau_\alpha}$.

10.4 The Qualitative Part

We shall use the definitions in Figure 49, which also recalls some of the already introduced abbreviations. Observe that $!\mathsf{P}^{\otimes}$ is not a term that represents a derivation of ILAL. However, it is perfectly sensible to associate it the logical

$$
\begin{aligned}
\multimap_\alpha &= (\alpha \multimap \alpha)\\
\otimes_\alpha &= \multimap_\alpha \otimes \multimap_\alpha \otimes \multimap_\alpha \otimes \multimap_\alpha \otimes \multimap_\alpha\\
!\otimes_\alpha &= (!\multimap_\alpha) \otimes (!\multimap_\alpha) \otimes (!\multimap_\alpha) \otimes (!\multimap_\alpha) \otimes (!\multimap_\alpha)\\
I &= \lambda x.x : \multimap_\alpha\\
P^{\otimes} &= 0 \otimes 1 \otimes \star \otimes \bot \otimes \top : \otimes_\alpha\\
!P^{\otimes} &= \bar{!}0 \otimes \bar{!}1 \otimes \bar{!}\star \otimes \bar{!}\bot \otimes \bar{!}\top : !\otimes_\alpha
\end{aligned}
$$

Fig. 49. Some useful definitions and abbreviations.

$$
\begin{aligned}
\mathit{config2config} &= \lambda c 0 1 \star \bot \top.\S(\lambda x x'.(\mathit{next\_config}\ !P^{\otimes})\\
&\qquad (\bar{\S}(c\ !(\mathit{step}\ \Pi_0\ \bar{!}0)\\
&\qquad\quad !(\mathit{step}\ \Pi_1\ \bar{!}1)\\
&\qquad\quad !(\mathit{step}\ \Pi_\star\ \bar{!}\star)\\
&\qquad\quad !(\mathit{step}\ \Pi_\bot\ \bar{!}\bot)\\
&\qquad\quad !(\mathit{step}\ \Pi_\top\ \bar{!}\top)\\
&\qquad\quad)(\mathit{base}\ \Pi_\emptyset\ x)\ (\mathit{base}\ \Pi_\emptyset\ x')\\
&\qquad)\\
&\quad)\\
\mathit{next\_config} &= \lambda P^{\otimes}.\lambda(h_l^l \otimes h_l^r \otimes t_l) \otimes (h_r^l \otimes h_r^r \otimes t_r) \otimes s.h_r^l(s(\hat{\delta}\ P^{\otimes}))\ h_l^r\ t_l\ t_r\\
\mathit{step} &= \lambda xy.\lambda u \otimes v \otimes z.x \otimes y \otimes (v\ z)\\
\mathit{base} &= \lambda xy.x \otimes I \otimes y
\end{aligned}
$$

Fig. 50. Terms producing a configuration from another configuration.

formula that we denote by $!\otimes_\alpha$. In particular, $!P^{\otimes}$ contributes to build a well-formed term, once inserted in a suitable context.

The key terms to encode a **PolyTime** Turing machine are in Figure 50. *config2config* takes a configuration c and yields a new one. Step by step, let us see the evaluation of *config2config* applied to the configuration (5). Substituting (5) for c, the evaluation of the whole sub-term in the scope of the $\bar{\S}$ operator yields:

$$(\Pi_\emptyset \otimes I \otimes x) \otimes (\Pi_\bot \otimes \bar{!}\bot \otimes (\bar{!}\star(\bar{!}1(\bar{!}\top\ x')))) \otimes \mathit{state}_i. \tag{6}$$

Observe that (6) is obtained because (5) *iterates* every *step* from *base* in order to extract what we call *head pairs* from the tape. In this example, the two head pairs are $\Pi_\emptyset \otimes I$, and $\Pi_\bot \otimes \bar{!}\bot$. The head pairs always have the same form: Π_0 will always be associated to $\bar{!}0$, Π_1 to $\bar{!}1$, $\Pi_\star$ to $\bar{!}\star$, $\Pi_\bot$ to $\bar{!}\bot$, $\Pi_\top$ to $\bar{!}\top$, and $\Pi_\emptyset$ to I.

Each of $\Pi_0, \Pi_1, \ldots$, together with state_i, extracts an element in a row of $\hat{\delta}$. This happens in *next config*. In its body, the actual state s extracts a row from $\hat{\delta}$, and the tape symbol h_r^l, read by the head, picks a move out of the row. In our running example, s is state_i, and h_r^l is $\Pi_\bot$. So, if $\delta(\mathsf{s}_i, \bot) = (\mathsf{s}_j, 1, L)$, then $Q_{i,\bot}$, producing $(\mathsf{s}_j, 1, L)$, must be $\mathit{left}^1_{\bot ij}$. The next computational steps are, internal to $\mathit{next}_{\mathit{config}}$ are:

$$
\begin{aligned}
&\Pi_\bot\ (\mathit{state}_i\ (\hat{\delta}\ !P^{\otimes}))\ I\ x\ (\bar{!}\star(\bar{!}1(\bar{!}\top\ x')))\\
&\qquad \rightsquigarrow^*\ \mathit{left}^1_{\bot ij}\ !P^{\otimes}\ I\ x\ (\bar{!}\star(\bar{!}1(\bar{!}\top\ x')))\\
&\qquad \rightsquigarrow^*\ (\lambda h_l t_l t_r.t_l \otimes (\bar{!}\bot(\bar{!}1\ t_r)) \otimes \mathit{state}_j)\ I\ x\ (\bar{!}\star(\bar{!}1(\bar{!}\top\ x')))\\
&\qquad \rightsquigarrow^*\ x \otimes (\bar{!}\bot\ (\bar{!}1(\bar{!}\star(\bar{!}1(\bar{!}\top\ x'))))) \otimes \mathit{state}_j
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{base} &= \forall\alpha\beta.\mathbf{proj}_{\alpha,\beta} \multimap \alpha \multimap \mathbf{proj}_{\alpha,\beta}\\
\mathbf{step} &= \forall\alpha\beta.\mathbf{proj}_{\alpha,\beta} \multimap \multimap_{\alpha} \multimap \mathbf{proj}_{\alpha,\beta} \multimap \mathbf{proj}_{\alpha,\beta}\\
\mathbf{next\_config} &= \forall\alpha.\otimes_{\alpha} \multimap ((\mathbf{proj}_{\otimes_{\alpha},\tau_{\alpha}} \otimes (\multimap_{\alpha}) \otimes \alpha) \otimes\\
&\qquad\qquad (\mathbf{proj}_{\otimes_{\alpha},\tau_{\alpha}} \otimes (\multimap_{\alpha}) \otimes \alpha) \otimes\\
&\qquad\qquad \mathbf{state}) \multimap (\alpha \otimes \alpha \otimes \mathbf{state})\\
\mathit{base} &: \mathbf{base}\\
\mathit{step} &: \mathbf{step}\\
\mathit{next\_config} &: \mathbf{next\_config}
\end{aligned}
$$

Fig. 51. Typing for *config2config*.

So, under the hypothesis of simulating $\delta(\mathsf{s}_i, \bot) = (\mathsf{s}_j, 1, L)$, the term, *config*, rewrites

$$\lambda 01 \star \bot\top.\S(\lambda xx'.x \otimes (\bar{!}\bot(\bar{!} \star (\bar{!}1(\bar{!}\top\ x')))) \otimes \mathit{state}_i) \tag{7}$$

into:

$$\lambda 01 \star \bot\top.\S(\lambda xx'.x \otimes (\bar{!}\bot(\bar{!}1(\bar{!} \star (\bar{!}1(\bar{!}\top\ x'))))) \otimes \mathit{state}_j) \tag{8}$$

by means of *config2config*. For those who want to check that *config2config* is *iterable*, that is, that *config2config* : **config** $\multimap$ **config**, Figure 51 gives some useful hints on the typing.

10.5 The Whole Encoding

We are, finally, in the position to complete our encoding of the machines in $\mathcal{T}^{\vartheta}_{\mathbf{PolyTime}}$, with a given ϑ, as derivations of **ILAL**.

Up to now, we have built the two main parts of the encoding. We call them *qualitative* and *quantitative*. The encoding $\hat{\delta}$ of the transition function, and the iterable term *config2config*, which maps configurations to configurations, belong to the first part. The encoding of the polynomials falls into the latter.

The whole encoding exploits the quantitative part to iterate the qualitative one, starting from the initial configuration. This is a suitable extension of the actual input. Every actual input of the encoding is a list, standing for a tape with the symbols $\{0, 1\}$ on it. The iteration is as much long as the value of the encoding of the polynomial, applied to the (unary representation) of the length of the actual input.

THEOREM 10.2. *There is a translation* $\hat{\ }$: $\mathcal{T}^{\vartheta}_{\mathbf{PolyTime}} \to \Lambda$ *such that, for any* $T \in \mathcal{T}^{\vartheta}_{\mathbf{PolyTime}}$, *and any input stream* x *for* T, *if* Tx *evaluates to* y, *then* $\hat{T}\hat{x} \rightsquigarrow^* \hat{y}$. *In particular*, $\hat{T}$: **tape** $\multimap \S^{\vartheta+6}$**tape**, *where*:

$$\mathbf{tape} = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).$$

The rest of this section develops the details about $\hat{\ }$: $\mathcal{T}^{\vartheta}_{\mathbf{PolyTime}} \to \Lambda$.

Figure 52 introduces the general scheme to encode any input for T as a term.

Figure 53 shows the encoding $\hat{T}$ of $T \in \mathcal{T}^{\vartheta}_{\mathbf{PolyTime}}$ which glues the quantitative and the qualitative parts together.

$$\lambda 01.\S(\lambda x.\bar{!}\chi_1(\ldots(\bar{!}\chi_p\ x)\ldots))\ ,$$
$$\text{where } \chi_{1\leq i\leq p},\in\{0,1\} \text{ and } p\geq 0\ .$$

Fig. 52. Encoding the input tapes.

$$\begin{aligned}\hat{T}\ =\ \lambda t.\mathit{config2tape}^{\vartheta+5}(\S((\lambda t_1\otimes t_2.\mathit{iter}^{\vartheta+3}\ (\hat{p}^{\vartheta}_{x}(\mathit{tape2int}\ t_1))\\ (!\mathit{config2config})\\ (\mathit{tape2config}\ t_2)\\ )\ \bar{\S}(\mathit{dbl\_tape}\ t)))&:\mathbf{tape}\multimap\S^{\vartheta+5}\mathbf{tape}\end{aligned}$$

Fig. 53. Encoding a **PolyTime** Turing machine in $\mathcal{T}^{\vartheta}_{\mathbf{PolyTime}}$.

$$\begin{aligned}\mathit{dbl\_tape}\ =\ \lambda t.\S(\ \bar{\S}(t\ !(\lambda xy.(\mathit{succ\_tape}_0\ x)\otimes(\mathit{succ\_tape}_0\ y))\\ !(\lambda xy.(\mathit{succ\_tape}_1\ x)\otimes(\mathit{succ\_tape}_1\ y))\\ )\ \mathit{empty\_tape}\ \mathit{empty\_tape})&:\mathbf{tape}\multimap\S(\mathbf{tape}\otimes\mathbf{tape})\end{aligned}$$
$$\text{where}$$
$$\mathit{succ\_tape}_\chi\ =\ \lambda t01.\S(\lambda x.\bar{!}\chi\ \bar{\S}(t\ 0\ 1)\ x):\mathbf{tape}\multimap\mathbf{tape}\qquad\text{with }\chi\in\{0,1\}$$
$$\mathit{empty\_tape}\ =\ \lambda 01.\S(\lambda x.x):\mathbf{tape}$$

Fig. 54. Doubling the contents of the actual input tape.

$$\begin{aligned}\mathit{config2tape}^{p}\ =\ \lambda c.\S^{p}((\lambda 01.\S((\lambda w\otimes y\otimes z.y)\\ (\ \bar{\S}(\ \bar{\S}^{p}\ c\ 0\ 1\ !(\lambda w.\mathit{empty\_tape})\ !I\ !I)\\ \mathit{empty\_tape}\ \mathit{empty\_tape}))\\ )\ !\mathit{succ\_tape}_0\ !\mathit{succ\_tape}_1\\ ):\S^{p}\mathbf{config}&\multimap\S^{p+1}\mathbf{tape}\end{aligned}$$

Fig. 55. Reading back a tape from a configuration.

Figures 54, 55, 56, 57, and 58 introduce the terms *dbl_tape*, *config2tape*, *tape2init_config*, *tape2int*, and the generalization iter^p of *iter*, with $1\leq p$, used by $\hat{T}$.

The term, *dbl_tape*, applied to a tape, doubles it. This is possible only by accepting that the result gets embedded into a §-box. For example:

$$\begin{aligned}&\mathit{dbl\_tape}\ (\lambda 01.\S(\lambda x.\bar{!}1(\bar{!}0\ x)))\\ &\leadsto^*\ \S((\lambda 01.\S(\lambda x.\bar{!}1(\bar{!}0\ x)))\otimes(\lambda 01.\S(\lambda x.\bar{!}1(\bar{!}0\ x)))).\end{aligned}$$

The term, *config2tape*, is used to erase the garbage, left by $\hat{T}$ on its tape, to produce the result. Recall, indeed, that we made some assumptions on the behavior of the elements of $\mathcal{T}^{\vartheta}_{\mathbf{PolyTime}}$ when entering s_a. The hypothesis was that the machines we encode enter s_a after their heads read the leftmost element of the tape, different from $\bot$. A further assumption is that the result is the portion of tape falling between the head position and the first occurrence of $\star$ to its right, once the machine is in state s_a. The term, *config2tape*, eliminates all the components of the encoding of a tape that is p §-boxes deep, but those between $\bot$ and the leftmost occurrence of $\star$. For example, if $\hat{T}$ reaches the configuration:

$$C\ =\ \S^{p}(\lambda 01\star\bot\top.\S(\lambda xx'.\bar{!}\bot\ x\ \otimes\ \bar{!}1(\bar{!}\star(\bar{!}0(\bar{!}\top x')))\otimes \mathit{state}_a)),$$

$$
\begin{array}{rcl}
\mathit{tape2config} & = & \lambda t.\mathit{coerc\_init\_config}\ (\mathit{tape2init\_config}\ t) : \mathbf{tape} \multimap \S\mathbf{config} \\
\text{where:} & & \\
\mathit{coerc\_init\_config} & = & \lambda c.\S(\ \bar{\S}(c\ !(\mathit{succ\_init\_config}_0) \\
 & & \qquad !(\mathit{succ\_init\_config}_1) \\
 & & \qquad !(\mathit{succ\_init\_config}_\top) \\
 & & \quad)\ \mathit{empty\_init\_config}\ \mathit{empty\_init\_config} \\
 & &)\ : \mathbf{config} \multimap \S\mathbf{config} \\
\mathit{empty\_init\_config} & = & \lambda 01 \star \bot\top.\S(\lambda xx'.(\ \bar{\S}\bot\ x) \otimes (\ \bar{\S}\top\ x')) : \mathbf{config} \\
\mathit{succ\_init\_config}_\chi & = & \lambda c01 \star \bot\top.\S(\lambda xx'.(\lambda w \otimes w' \otimes s.w \otimes (\bar{!}\chi\ w') \otimes s) \\
 & & \qquad\qquad (\ \bar{\S}(c\ 0\ 1 \star \bot\top)\ x\ x') \\
 & &)\ : \mathbf{config} \multimap \mathbf{config} \qquad \text{where } \chi \in \{0,1\} \\
\text{and:} & & \\
\mathit{tape2init\_config} & = & \lambda t01 \star \bot\top.\S(\lambda ww'.\mathit{step}\ (\bar{!}\bot\ w) \\
 & & \qquad\qquad \bar{!}(t\ !(\mathit{step}\bar{!}\ 0) \\
 & & \qquad\qquad\quad !(\mathit{step}\bar{!}\ 1) \\
 & & \qquad\qquad)\ I \otimes (\bar{!}\top\ w') \otimes \mathit{state}_0 \\
 & &)\ : \mathbf{tape} \multimap \mathbf{config} \\
\mathit{step} & = & \lambda x.\lambda y \otimes z.x \otimes (y\ w) : \beta \multimap ((\alpha \multimap \alpha) \otimes \alpha) \multimap (\beta \otimes \alpha)
\end{array}
$$

Fig. 56. The initial configuration out of the actual input tape.

$$\mathit{tape2int} \ = \ \lambda ts.\S(\lambda x.\ \bar{\S}(t\ s\ s)\ x) : \mathbf{tape} \multimap \mathbf{Int}$$

Fig. 57. transforming the actual input tape in to an integer.

$$\mathit{iter}^p \ = \ \lambda xyz.\S^p(\S(\ \bar{\S}(\ \bar{\S}^p x\ y)\ \bar{\S}z)) : \mathbf{Int}^p \multimap\ !(A \multimap A) \multimap \S A \multimap \S^{p+1}A$$

Fig. 58. Generalizing the iteration.

then $\mathit{config2tape}^{\,p}\ C \rightsquigarrow^* \S^{p+1}(\lambda 01.\S(\lambda x.\ \bar{\S}1\ x))$, that is, the result of the simulated machine is simply the tape with the single alphabet element 1, and embedded into $p+1$ §-boxes.

The term, *tape2config*, goes in the opposite direction than *config2tape*. Given the encoding of a tape t, *tape2config* t gives the initial configuration of the encoded machine, embedded into one §-box. For example:

$$
\begin{array}{l}
\mathit{tape2config}\ (\lambda 01.\S(\lambda x.\bar{!}1(\bar{!}0\ x))) \\
\quad \rightsquigarrow^*\ \S(\lambda 01 \star \bot\top.\S(\lambda xx'.\bar{!}\bot\ x\ \otimes\ \bar{!}1(\bar{!}0(\bar{!}\top\ x)) \otimes \mathit{state}_0)).
\end{array}
$$

The term, *tape2int*, applied to a tape, produces the numeral, which expresses the unary length of the tape itself. For example:

$$\mathit{tape2int}\ (\lambda 01.\S(\lambda x.\bar{!}1(\bar{!}0\ x))) \ \rightsquigarrow^*\ \lambda y.\S(\lambda x.\bar{!}y(\bar{!}y\ x)).$$

The term, $\mathit{iter}^{\,p}$, is the obvious generalization of *iter* to a first argument with type $\mathbf{Int}^p$.

As a summary, we rephrase the intuitive explanation we gave at the beginning of this subsection, to describe the behavior of the encoding. $\mathit{iter}^{\vartheta+3}$ iterates $\hat{p}_x^{\vartheta}\ (\mathit{tape2int}\ t_1)$ times the term $!\mathit{config2config}$, starting from the initial

$$\lambda o 1 \star \bot.\S(\lambda x.(!\bar{\chi}_1(\ldots(!\bar{\chi}_p(!\bot\ x))\ldots)))$$
$$\otimes$$
$$\lambda o 1 \star \top.\S(\lambda y.(!\bar{\chi}'_1(\ldots(!\bar{\chi}'_q(!\top\ y))\ldots)))$$
$$\otimes$$
$$state_i$$

where $\chi_{1\leq i\leq p}, \chi'_{1\leq j\leq q} \in \{0, 1, \star\}$, with $p, q \geq 0$.

Fig. 59. Obvious encoding of the configurations.

configuration given by *tape2config* t_2. The variables t_1, t_2 stand for the two copies of the input tape, produced by *dbl_tape* t, where t represents the input tape itself. Finally, *config2tape*$^{\vartheta+6}$ reads back the result.

11. CONCLUSIONS

Light Linear Logic [Girard 1998] is the first logical system with cut elimination, whose formulas can be used as program annotations to improve the evaluation efficiency of the reduction. In the remark concluding Section 8.1, we observed that the relation between the strategy to get such an efficiency and the more traditional strategies is not completely clear; we left an open problem.

By drastically simplifying Light Linear Logic sequent calculus, Intuitionistic Light Affine Logic helps to understand the main crucial issues of Girard's technique to control the computational complexity. Roughly, it can be summarized in the motto: *stress and take advantage of linearity whenever possible*. Technically, the simplification allows to see § as a *weak* version of dereliction in Linear Logic [Girard 1995]. It opens !-boxes while preserving the information on levels. Moreover, the proof about the expressive power of **ILAL** is not immediately trivial. In particular, some reader may have noticed that the configurations of the machines are not encoded obviously, like in Girard [1998], as recalled in Figure 59. Roversi [1999] discusses about why such an encoding can not work. Roughly, it does not allow to write an *iterable* function *config2config*, which is basic to produce the whole encoding.

The idea to consider full weakening in Light Linear Logic, to get Light Affine Logic, was suggested by the fact that in Optimal Reduction [Asperti and Guerrini 1998] we may freely erase any term. For the experts, the garbage nodes do not get any index.

Some attempts to extract a programming language with automatic polymorphic type inference, from **ILAL** are in Roversi [1998; 2000]. However, they must be improved in terms of expressive power and readability.

ACKNOWLEDGMENT

We warmly thanks the anonymous referee and Simone Martini who greatly helped us by suggesting how to correct and clarify this paper.

REFERENCES

ASPERTI, A. 1998. Light affine logic. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (*LICS'98*). IEEE Computer Society Press, Los Alamitos, Calif.

ASPERTI, A., AND GUERRINI, S. 1998. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, Cambridge, England.

DANOS, V., AND JOINET, J.-B. 1999. Linear logic & elementary time. In *Proceedings of the 1st International workshop on Implicit Computational Complexity* (*ICC'99*) (Santa Barbara, Calif.).

GIRARD, J.-Y. 1995. Proof nets: The parallel syntax for proof-theory. In *Logic and Algebra*. New York.

GIRARD, J.-Y. 1998. Light linear logic. *Inf. Comput. 143*, 175–204.

GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. 1989. *Proofs and Types*. Cambridge University Press, Cambridge, England.

GIRARD, J.-Y., SCEDROV, A., AND SCOTT, P. J. 1998. Light linear logic. *Inf. Comput. 143*, 175–204.

LEIVANT, D. 1994. A foundational delineation of poly-time. *Inf. Comput. 110*, 391–420.

LEIVANT, D., AND MARION, J.-Y. 1993. Lambda calculus characterizations of poly-time. *Funda. Inf. 19*, 167–184.

MITCHELL, J. 1988. Polymorphic type inference and containment. *Inf. Comput. 76*, 211–249.

ROVERSI, L. 1998. A polymorphic language which is typable and poly-step. In *Proceedings of the Asian Computing Science Conference (ASIAN'98)* (Manila, The Philippines). Lecture Notes in Computer Science, vol. 1538, Springer Verlag, New York, pp. 43–60.

ROVERSI, L. 1999. A P-Time Completeness proof for light logics. In *Proceedings of the 9th Annual Conference of the EACSL (CSL'99)* (Madrid, Spain), Lecture Notes in Computer Science, vol. 1683. Springer-Verlag, New York, pp. 469–483.

ROVERSI, L. 2000. Light affine logic as a programming language: a first contribution. *Inte. J. Found. Comput. Sci. 11*, 1 (Mar.), 113–152.

TORTORA, L. 2000a. Additives of linear logic and normalization-Part I: A (restricted) Church-Rosser property. *Theor. Comput. Sci.*, to appear.

TORTORA, L. 2000b. Réseaux, cohérence et expériences obsessionnelles. Ph.D. dissertation, Université Paris 7, Paris.

Received June 2000; revised November 2000; accepted January 2001