On the Computational Complexity of Cut-Elimination in Linear Logic

Harry G. Mairson[1] and Kazushige Terui[2]

[1] Computer Science Department, Brandeis University,
Waltham, Massachusetts 02454, USA
mairson@cs.brandeis.edu
[2] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku,
101-8430 Tokyo, Japan
terui@nii.ac.jp

Abstract. Given two proofs in a logical system with a confluent cut-elimination procedure, the *cut-elimination problem* (CEP) is to decide whether these proofs reduce to the same normal form. This decision problem has been shown to be PTIME-complete for Multiplicative Linear Logic (Mairson 2003). The latter result depends upon a restricted simulation of weakening and contraction for boolean values in **MLL**; in this paper, we analyze how and when this technique can be generalized to other **MLL** formulas, and then consider CEP for other subsystems of Linear Logic. We also show that while additives play the role of *nondeterminism* in cut-elimination, they are not needed to express *deterministic* PTIME computation. As a consequence, affine features are irrelevant to expressing PTIME computation. In particular, Multiplicative Light Linear Logic (**MLLL**) and Multiplicative Soft Linear Logic (**MSLL**) capture PTIME even without additives nor unrestricted weakening. We establish hierarchical results on the cut-elimination problem for **MLL**(PTIME-complete), **MALL**(coNP-complete), **MSLL**(EXPTIME-complete), and for **MLLL** (2EXPTIME-complete).

1 Introduction

Cut-elimination is naturally seen as a *function* from proofs to their normal form, and we can derive from it an equally natural *decision problem:* if L is a logical system with a confluent cut-elimination procedure, and we are given two proofs in L, do they reduce to the same normal form? Call this the *cut elimination problem (CEP).* When L has reasonable representations of boolean values as proofs, an even simpler decision problem is to ask: given a proof, does it reduce to the representation for "true"?

Through the Curry-Howard correspondence, we know that proofs in linear logics represent programs, typically in a functional programming langauge with highly specified forms of copying, where cut-elimination serves as an interpreter: normalization is evaluation. The cut-elimination problem is then a fundamental question about program equivalence, and how hard it is to decide. Moreover, the

correspondence facilitates our identification of particular logics with associated complexity classes, where our goal is to link the expressive power of proofs with a suitably powerful interpreter that can "run" representations of programs in that complexity class.

The cut-elimination problem is known to be non-elementary for simply typed λ-calculus [Sta79], and hence for linear logic. Low order fragments of simply typed λ-calculus are investigated in [Sch01]. In this paper, we consider the decision problem for various weak subsystems of linear logic that have no exponentials, or have very weak forms of them (i.e., the so-called "light" linear logics). Such an investigation suggests another way to characterize the complexity of linear logics: not only by the complexity of theorem proving (proof search)—see, for example, [Lin95]—but also by the complexity of theorem simplification (proof normalization).

Even in intuitionistic multiplicative linear logic (**IMLL**), which has no exponentials, it is possible to simulate weakening and contraction for a restricted set of formulas, including a formula whose proofs code boolean values. As a consequence, we derive PTIME-completeness for CEP in **IMLL**; see Section 2. This result contradicts folkloric intuitions that **MLL** proofnets could be normalized in logarithmic space—that is, with only a finite number of pointers into the proofnet, presumably following paths in the style of the geometry of interaction. Similar to the results for **IMLL**, in Section 3 we derive coNP-completeness results for **IMALL**, where we also have additives.

An alternative way to represent a complexity class by some logic is to consider the functions realizable (say, by a Turing machine) in the class, and show how each can be coded as a *fixed* proof (program) in the logic. For example, Light Linear Logic has been shown to so represent PTIME computations [Gir98], and the use of additives in that proof was replaced by unrestricted weakening in Light Affine Logic [Asp98,AR02]. We improve these results to show that such weakening is also unnecessary: Multiplicative Light Linear Logic is sufficient to capture PTIME (see Section 4), where we also prove that deciding CEP is complete for doubly-exponential time. Finally, in Section 5 we show similar characterizations of exponential time in Multiplicative Soft Linear Logic [Laf01].

2 Expressivity of Multiplicatives

2.1 Weakening in MLL

We restrict our attention to the intuitionistic $(\multimap, \forall)$ fragment **IMLL** of **MLL**, although all the results in this section carry over to the full classical **MLL** with no difficulty. Moreover, we omit type annotation from the proof syntax, and identify proofs of **IMLL** with type-free terms of linear λ-calculus.

A *term* (*proof*) of **IMLL** is either a variable x, or an application (tu) where t and u are terms such that $FV(t) \cap FV(u) = \emptyset$, or an abstraction $(\lambda x.t)$ where t is a term and $x \in FV(t)$. Terms are considered up to α-equivalence, and the variable convention is adopted. The substitution operation $t[u/x]$ and the β reduction relation are defined as usual. The *size* $|t|$ of a term t is the number of

nodes in its syntax tree. The *type assignment rules* are as follows[3]:

$$\frac{}{x:A \vdash x:A} \qquad \frac{\Gamma \vdash u:A \quad x:A, \Delta \vdash t:C}{\Gamma, \Delta \vdash t[u/x]:C} \qquad \frac{x:A, \Gamma \vdash t:B}{\Gamma \vdash \lambda x.t : A \multimap B}$$

$$\frac{\Gamma \vdash u:A \quad x:B, \Delta \vdash t:C}{\Gamma, y:A \multimap B, \Delta \vdash t[yu/x]:C} \qquad \frac{\Gamma \vdash t:A}{\Gamma \vdash t:\forall\alpha.A}\ \alpha \notin FV(\Gamma) \qquad \frac{x:A[B/\alpha], \Gamma \vdash t:C}{x:\forall\alpha.A, \Gamma \vdash t:C}$$

Here, $\Gamma, \Delta \ldots$ stand for finite multisets of *declarations* $x:A$ and $FV(\Gamma)$ denotes the set of all free type variables in Γ. We say that a term t is *of type* A (or t *is a proof of* A) if $\vdash t:A$ is derivable by the above rules. A type A is *inhabitied* if there is a term of type A.

Unit $\mathbf{1}$ and tensor product $\otimes$ are introduced by means of the second order definitions:

$$\mathbf{1} \equiv \forall\alpha.\alpha \multimap \alpha \qquad A \otimes B \equiv \forall\alpha.(A \multimap B \multimap \alpha) \multimap \alpha$$
$$\mathsf{I} \equiv \lambda x.x \qquad t \otimes u \equiv \lambda x.xtu$$
$$\mathsf{let}\ t\ \mathsf{be}\ \mathsf{I}\ \mathsf{in}\ u \equiv tu \qquad \mathsf{let}\ t\ \mathsf{be}\ x \otimes y\ \mathsf{in}\ u \equiv t(\lambda xy.u).$$

Tensor product is naturally extended to n-ary ones: $t_1 \otimes \ldots \otimes t_n$ and $\mathsf{let}\ u\ \mathsf{be}\ x_1 \otimes \cdots \otimes x_n\ \mathsf{in}\ t$. The expression $\lambda x_1 \otimes \cdots \otimes x_n.t$ stands for $\lambda z.\mathsf{let}\ z\ \mathsf{be}\ x_1 \otimes \cdots \otimes x_n\ \mathsf{in}\ t$. We also use shorthand notations such as $id \equiv \lambda x.x$, $t \circ u \equiv \lambda x.t(u(x))$, $A^n \equiv \underbrace{A \otimes \cdots \otimes A}_{n\ times}$, $A^{(n)} \multimap B \equiv \underbrace{A \multimap \cdots A \multimap}_{n\ times} B$.

Our first observation is that a version of weakening rule can be constructed for a certain restricted class of **IMLL** formulas.

Definition 1 (Π_1, Σ_1, $e\Pi_1$, $e\Sigma_1$). *A type A is Π_1 (Σ_1) if it is built from type variables by $\multimap$, $\mathbf{1}$, $\otimes$ (viewed as primitives) and positive (negative) occurrences of $\forall$. An $e\Pi_1$ ($e\Sigma_1$) type is like a Π_1 (Σ_1) type, but it may additionally contain negative (positive) occurrences of* inhabited $\forall$*-types.*

The above definition of Π_1 and $e\Pi_1$ involves $\mathbf{1}$ and $\otimes$ as primitives, but we may ignore them in practice, because negative occurrences of $\otimes$ and $\mathbf{1}$ can be removed by isomorphisms $((A \otimes B) \multimap C) \circ\!\!-\!\!\circ (A \multimap B \multimap C)$ and $(\mathbf{1} \multimap C) \circ\!\!-\!\!\circ C$, while positive occurrences can be replaced with their Π_1 definitions.

Finite data types are naturally represented by closed inhabited Π_1 types. A typical example is the boolean type: $\mathbf{B} \equiv \forall\alpha.\alpha \multimap \alpha \multimap \alpha \otimes \alpha$. Meanwhile, functional types over those finite data types, such as $(\mathbf{B} \multimap \mathbf{B}) \multimap \mathbf{B}$, are all included in the class $e\Pi_1$.

Theorem 1 ($e\Pi_1$-Weakening). *For any closed $e\Pi_1$ type A, there is a term w_A of type $A \multimap \mathbf{1}$.*

[3] Note that *any* term of linear λ-calculus has a propositional type [Hin89]; the role of second order quantifers is not to increase the number of typable terms, but to assign a uniform type to structurally related terms.

Proof. Without loss of generality, we may assume that A does not contain $\otimes$ and $\mathbf{1}$. Let $B[\mathbf{1}]$ be the type B with all free variables replaced with $\mathbf{1}$. By simultaneous induction, we prove: (i) for any $e\Pi_1$ type B, $B[\mathbf{1}] \vdash \mathbf{1}$ is provable; and (ii) for any $e\Sigma_1$ type B, $\vdash B[\mathbf{1}]$ is provable. When B is a variable, the claims are obvious. When B is $C \multimap D$, for (i) we derive $(C \multimap D)[\mathbf{1}] \vdash \mathbf{1}$ from $\vdash C[\mathbf{1}]$ and $D[\mathbf{1}] \vdash \mathbf{1}$, and for (ii) we derive $\vdash (C \multimap D)[\mathbf{1}]$ from $C[\mathbf{1}] \vdash \mathbf{1}$ and $\vdash D[\mathbf{1}]$. Let B be $\forall\alpha.C$. If B is $e\Pi_1$, we derive $(\forall\alpha.C)[\mathbf{1}] \vdash \mathbf{1}$ from $C[\mathbf{1}] \vdash \mathbf{1}$. If B is $e\Sigma_1$, $\vdash B$ is provable by definition, and so is $\vdash B[\mathbf{1}]$. ■

2.2 Encoding boolean circuits

Let A be an arbitrary type, and B be a type that supports weakening in the sense we have just described; we can then define a projection function $\mathsf{fst}_B : A \otimes B \multimap A$, given by $\mathsf{fst}_B \equiv \lambda x.\mathsf{let}\ x\ \mathsf{be}\ y \otimes z\ \mathsf{in}\ (\mathsf{let}\ \mathsf{w}_B(z)\ \mathsf{be}\ \mathsf{I}\ \mathsf{in}\ y)$. By using this coding, we can then specify boolean values, weakening, and operations (including duplication) as:

$$
\begin{array}{lll}
\mathsf{true} & \equiv \lambda xy.x \otimes y & : \mathbf{B} \\
\mathsf{false} & \equiv \lambda xy.y \otimes x & : \mathbf{B} \\
\mathsf{w}_{\mathbf{B}} & \equiv \lambda z.\mathsf{let}\ z\mathsf{I}\mathsf{I}\ \mathsf{be}\ x \otimes y\ \mathsf{in}\ (\mathsf{let}\ y\ \mathsf{be}\ \mathsf{I}\ \mathsf{in}\ x) & : \mathbf{B} \multimap \mathbf{1} \\
\mathsf{not} & \equiv \lambda Pxy.Pyx & : \mathbf{B} \multimap \mathbf{B} \\
\mathsf{or} & \equiv \lambda PQ.\mathsf{fst}_{\mathbf{B}}(P\,\mathsf{true}\,Q) & : \mathbf{B} \multimap \mathbf{B} \multimap \mathbf{B} \\
\mathsf{cntr} & \equiv \lambda P.\mathsf{fst}_{\mathbf{B}\otimes\mathbf{B}}(P(\mathsf{true} \otimes \mathsf{true})(\mathsf{false} \otimes \mathsf{false})) & : \mathbf{B} \multimap \mathbf{B} \otimes \mathbf{B}
\end{array}
$$

Recall that a language $X \subseteq \{0,1\}^*$ is *logspace reducible* to $Y \subseteq \{0,1\}^*$ if there exists a logspace function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that $w \in X$ iff $f(w) \in Y$. Language X is PTIME-*complete* if $X \in$ PTIME and each language $L \in$ PTIME is logspace reducible to X; a decision problem is said to be PTIME-complete when the language defined by that problem is PTIME-complete. The canonical PTIME-complete decision problem is the following:

Circuit Value Problem: Given a boolean circuit C with n inputs and 1 output, and truth values $\boldsymbol{x} = x_1, \ldots, x_n$, is $\boldsymbol{x}$ accepted by C? [Lad75]

Using the above coding of boolean operations, the problem is logspace reducible to CEP for **IMLL**:

Theorem 2 (PTIME-completeness of IMLL, [Mai03]). *There is a logspace algorithm which transforms a boolean circuit C with n inputs and m outputs into a term t_C of type $\mathbf{B}^n \multimap \mathbf{B}^m$, where the size of t_C is $O(|C|)$. As a consequence, the cut-elimination problem for* **IMLL** *is* PTIME-*complete.*

Since binary words of length n can be represented by $\mathbf{B}^n$, any finite function $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ can be represented by a term $t_f : \mathbf{B}^n \multimap \mathbf{B}^m$. In this sense, **MLL** captures all the finite functions.

2.3 Contraction in MLL

One of the key observations in proving Theorem 2 is that contraction is available for **B**. We now generalize this observation, and show that the same holds for all closed inhabited Π_1 types (i.e. finite data types). First we show that conditional is available in **IMLL**:

Lemma 1 (Conditional). *Let t_1 and t_2 be terms such that $x_1 : C_1, \ldots, x_n : C_n \vdash t_i : D$ for $i = 1, 2$, and the type $A \equiv C_1 \multimap \cdots C_n \multimap D$ is $e\Pi_1$ (not necessarily closed). Then there is a term* $\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ *such that*

$$b{:}\mathbf{B}, x_1{:}C_1, \ldots, x_n{:}C_n \vdash \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : D,$$

where $(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \longrightarrow t_1$ *and* $(\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \longrightarrow t_2$.

Proof. Define $\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ u \equiv \mathsf{fst}_{\forall\boldsymbol{\alpha}.A}(b(\lambda\boldsymbol{x}.t)(\lambda\boldsymbol{x}.u))\boldsymbol{x}$, where $\boldsymbol{x}$ abbreviates $x_1, \ldots, x_n$ and $\forall\boldsymbol{\alpha}.A$ is the universal closure of A. This term can be typed as required; $\lambda\boldsymbol{x}.t$ and $\lambda\boldsymbol{x}.u$ have type $\forall\boldsymbol{\alpha}.A$, thus $b(\lambda\boldsymbol{x}.t)(\lambda\boldsymbol{x}.u)$ has type $\forall\boldsymbol{\alpha}.A \otimes \forall\boldsymbol{\alpha}.A$, to which the projection $\mathsf{fst}_{\forall\boldsymbol{\alpha}.A}$ applies. The rest is obvious. ∎

Fix a quantifier-free type A of size k, built from a single type variable α. A *long normal form* of type A is a term t such that $\vdash t : A$ has a derivation in which all identity axioms are *atomic*, i.e., of the form $x{:}\alpha \vdash x{:}\alpha$. It is clear that every long normal form t of type A has size bounded by k, and we may assume that all variables occurring in it are from a fixed set of variables $\{x_1, \ldots, x_k\}$ (due to α-equivalence). Therefore, t can be written as a word in $\{0,1\}^n$, where $n = O(k \log k)$. Since $\{0,1\}^n$ can in turn be represented by $\mathbf{B}^n$, there must be a function $\lceil\ \rceil$ which maps a given term u of size bounded by k into a term $\lceil u \rceil$ of type $\mathbf{B}^n$. Furthermore, as a consequence of Theorem 2, we can associate to this coding two terms $\mathsf{abs}, \mathsf{app} : \mathbf{B}^n \multimap \mathbf{B}^n \multimap \mathbf{B}^n$ which satisfy

$$\begin{array}{ll} \mathsf{abs}\lceil y\rceil\lceil t\rceil \longrightarrow^* \lceil \lambda y.t\rceil, & \text{if } |\lambda y.t| \le k \text{ and } y \in \{x_1, \ldots, x_k\}; \\ \mathsf{app}\lceil t\rceil\lceil u\rceil \longrightarrow^* \lceil tu\rceil, & \text{if } |tu| \le k. \end{array}$$

We now show that the coding function $\lceil\ \rceil$ can be internalized in **IMLL**, as far as the long normal forms of a fixed type A is concerned. For each subtype B of A, define $\sigma_B(t)$ and $\tau_B(t)$ as follows:

$$\begin{array}{ll} \sigma_\alpha(t) \equiv t & \sigma_{B\multimap C}(t) \equiv \mathsf{abs}\lceil y\rceil\sigma_C(t\,\tau_B(\lceil y\rceil)); \\ \tau_\alpha(t) \equiv t & \tau_{B\multimap C}(t) \equiv \lambda z.\tau_C(\mathsf{app}t\,\sigma_B(z)); \end{array}$$

where y is from $\{x_1, \ldots, x_k\}$ and *fresh*, in the sense that $\lceil y\rceil$ does not occur in t. The term $\sigma_B(t)$ has type $\mathbf{B}^n$ whenever t has type $B[\mathbf{B}^n/\alpha]$, and $\tau_B(t)$ has type $B[\mathbf{B}^n/\alpha]$ whenever t has type $\mathbf{B}^n$. Finally, let $\mathsf{code}_A \equiv \lambda x.\sigma_A(x) : A[\mathbf{B}^n/\alpha] \multimap \mathbf{B}^n$.

Lemma 2 (Internal Coding). *Let A be as above. For each closed long normal form t of type A,* $\mathsf{code}_A(t) \longrightarrow^* \lceil t\rceil$.

For example, let A_1 be $((\alpha \multimap \alpha) \multimap \alpha) \multimap (\alpha \multimap \alpha) \multimap \alpha$, which has two long normal forms $t_1 \equiv \lambda Ff.f(F(\lambda y.y))$ and $t_2 \equiv \lambda Ff.F(\lambda y.fy))$. Then code_{A_1} is defined as follows:

$$\begin{aligned}
\tau_{\alpha\multimap\alpha}(\lceil f \rceil) &\equiv \lambda x.\mathsf{app}\lceil f\rceil x;\\
\tau_{(\alpha\multimap\alpha)\multimap\alpha}(\lceil F \rceil) &\equiv \lambda g.\mathsf{app}\lceil F\rceil(\mathsf{abs}\lceil y\rceil(g\lceil y\rceil));\\
\mathsf{code}_{A_1} &\equiv \lambda z.\mathsf{abs}\lceil F\rceil(\mathsf{abs}\lceil f\rceil(z\tau_{(\alpha\multimap\alpha)\multimap\alpha}(\lceil F\rceil)\tau_{\alpha\multimap\alpha}(\lceil f\rceil))).
\end{aligned}$$

It is easy to check that $\mathsf{code}_F(t_i)$ reduces to $\lceil t_i \rceil$ for $i = 1, 2$.

Theorem 3 (Π_1-Contraction). *Let A be a closed Π_1 type which is inhabited. Then there is a contraction map $\mathsf{cntr}_A : A \multimap A \otimes A$ such that for any normal form t of type A, $\mathsf{cntr}_A(t) \longrightarrow^* t' \otimes t'$, where t' is a long normal form η-equivalent to t.*

Proof. Without loss of generality, we may assume that A is free from $\otimes$ and $\mathbf{1}$. Let A^- be obtained from A by replacing all subtypes $\forall\beta.C$ by $C[\alpha/\beta]$ for a fixed variable α. Then, there is a canonical map $\mathsf{iso}_A : A \multimap A^-[D/\alpha]$ for any D which preserves the structure of terms up to η-equivalence. By applying Lemma 2 to the type A^- we obtain a coding map $\mathsf{code}_{A^-} : A^-[\mathbf{B}^n/\alpha] \multimap \mathbf{B}^n$.

Let $t_1, \ldots, t_l$ be the long normal forms of type A. By using the conditional in Lemma 1 several times, we can build a term $\mathsf{copy}_A : \mathbf{B}^n \multimap A \otimes A$ which satisfies

$$\begin{aligned}
\mathsf{copy}_A(u) &\longrightarrow^* t_i \otimes t_i, \quad \text{if } u \equiv \lceil t_i \rceil;\\
&\longrightarrow^* t_1 \otimes t_1, \quad \text{otherwise.}
\end{aligned}$$

Finally, define $\mathsf{cntr}_A \equiv \mathsf{copy}_A \circ \mathsf{code}_{A^-} \circ \mathsf{iso}_A$. ∎

3 Additives as Nondeterminism

3.1 Additive slices and nondeterministic cut-elimination

We now move on to the multiplicative additive fragment of Linear Logic. We again confine ourselves to the intuitionistic fragment **IMALL**, and furthermore, we only consider & as the additive connective, although $\oplus$ could be added harmlessly.[4]

The *terms of* **IMALL** are defined analogously to the terms of **IMLL**, but we have in addition: (i) if t and u are terms and $FV(t) = FV(u)$, then so is $\langle t, u\rangle$; (ii) if t is a term, then so are $\pi_1(t)$ and $\pi_2(t)$. The type assignment rules are extended with:

$$\frac{\Gamma \vdash t_1 : A_1 \quad \Gamma \vdash t_2 : A_2}{\Gamma \vdash \langle t_1, t_2\rangle : A_1 \,\&\, A_2} \qquad \frac{x : A_i, \Gamma \vdash t : C}{y : A_1 \,\&\, A_2, \Gamma \vdash t[\pi_i(y)/x] : C}\ i = 1, 2,$$

[4] However, we have to be careful when considering the classical system, which is *not* confluent as it stands. It could be overcome by adopting Tortora's proofnet syntax with generalized & boxes, which enjoys confluence [dF03]; see also [MR02].

and the reduction rules are extended with $\pi_i\langle t_1, t_2\rangle \longrightarrow t_i$, for $i = 1, 2$.

Note that some reductions cause duplication (e.g. $(\lambda x.\langle x, x\rangle)t \longrightarrow \langle t, t\rangle$), thus cut-elimination in **IMALL** is no more in linear steps.[5] However, duplication can be avoided by computing each component of $\langle t_1, t_2\rangle$ separately. To formalize this idea, we recall the notion of *slice* [Gir87].

Definition 2 (Slices). *A* slice *of a term t is obtained by applying the following operation to t as much as possible: $\langle u, v\rangle \mapsto \langle u\rangle_1$, or $\langle u, v\rangle \mapsto \langle v\rangle_2$.*

We say that two slices t and u (of possibly different terms) are *compatible* if there is no context (i.e. a term with a hole) Φ such that $t \equiv \Phi[\langle t'\rangle_i]$, $u \equiv \Phi[\langle u'\rangle_j]$, and $i \neq j$.

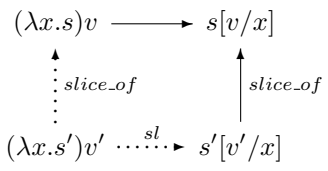
Lemma 3 (Slicewise checking). *Two terms t and u are equivalent if and only if for every compatible pair (t', u') of slices of t and u, we have $t' \equiv u'$.*

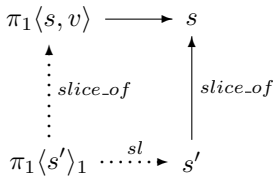
The reduction rules are naturally adapted for slices:

$$(\lambda x.t)u \xrightarrow{sl} t[u/x] \qquad \pi_i\langle t\rangle_i \xrightarrow{sl} t \qquad \pi_i\langle t\rangle_j \xrightarrow{sl} \mathsf{fail}, \text{ if } i \neq j.$$

Lemma 4 (Pullback). *Let $t \longrightarrow^* u$ and u' be a slice of u. Then there is a unique slice t' of t such that $t' \xrightarrow{sl}{}^* u'$.*

Proof. See the following diagrams





■

Note that there are exponentially many slices for a given term, but once a slice has been chosen, the computation afterwards can be done in linear steps, thus in quadratic time, since each slice is entirely a linear term. We therefore have a nondeterministic polynomial time cut-elimination procedure, viewing the slicing operation in Definition 2 as a nondeterministic reduction rule. Lemma 3 states that the equivalence of two normal forms can be checked slicewise, and Lemma 4 assures that every slice of a normal form can be obtained by the above nondeterministic procedure. Hence we may conclude that the cut-elimination problem for **IMALL** is in coNP.

[5] There *is* a linear step cut-elimination procedure for terms (proofnets) of *lazy* types, i.e., those which do not contain positive occurrences of & and negative occurrences of ∀.

3.2 Encoding a coNP-complete Problem

Now we show that the following coNP-complete problem is logspace reducible to CEP for **IMALL**:

Logical Equivalence Problem: Given two boolean formulas, are they logically equivalent? (cf. [GJ78])

By Theorem 2, every boolean formula C with n variables can be translated into a term t_C of type $\mathbf{B}^{(n)} \multimap \mathbf{B}$ in $O(\log |C|)$ space. For each $1 \leq k \leq n$, let

$$\mathsf{ta}_k \equiv \lambda f.\lambda x_1 \cdots x_{k-1}.\langle f\ \mathsf{true}\ x_1 \cdots x_{k-1}, f\ \mathsf{false}\ x_1 \cdots x_{k-1}\rangle,$$

which is of type $\forall\alpha.(\mathbf{B}^{(k)} \multimap \alpha) \multimap (\mathbf{B}^{(k-1)} \multimap \alpha \,\&\, \alpha)$, and define $\mathsf{ta}(t_C)$ by

$$\mathsf{ta}(t_C) \equiv \mathsf{ta}_1(\cdots(\mathsf{ta}_n t_C)\cdots) : \underbrace{\mathbf{B} \,\&\, \cdots \,\&\, \mathbf{B}}_{2^n\ times}.$$

It is clear that the term $\mathsf{ta}(t_C)$ can be built from t_C with the help of a counter of size $O(\log n)$.

The normal form of $\mathsf{ta}(t_C)$ consists of 2^n boolean values, each of which corresponds to a 'truth assignment' to the formula C. For example, $\mathsf{ta}(\mathsf{or})$ reduces to $\langle\langle \mathsf{or\ true\ true}, \mathsf{or\ true\ false}\rangle, \langle \mathsf{or\ false\ true}, \mathsf{or\ false\ false}\rangle\rangle$,

and thus to $\langle\langle \mathsf{true}, \mathsf{true}\rangle, \langle \mathsf{true}, \mathsf{false}\rangle\rangle$.

Therefore, two formulas C and D with n variables are logically equivalent if and only if $\mathsf{ta}(t_C)$ and $\mathsf{ta}(t_D)$ reduce to the same normal form.

Theorem 4 (coNP-completeness of IMALL). *The cut-elimination problem for* **IMALL** *is* coNP*-complete.*

Remark 1. We do not claim that *the* complexity of **MALL** is coNP. What we have shown is that a specific problem, CEP for **MALL**, is complete for coNP. If we had considered the complement of CEP, then the result would have been NP-completeness. Likewise, we could obtain a $\mathcal{C}$-completeness result for any class $\mathcal{C}$ in the polynomial time hierarchy by complicating the problem more and more.

However, we do claim that additives have something to do with *nondeterminism*, as they provide a notion of nondeterminsitic cut-elimination, as well as a very natural coding of nondeterministic Turing machine computation.

4 Multiplicative Light Linear Logic and 2EXPTIME

In this section, we show that the intuitionistic multiplicative fragment **IMLLL** of Light Linear Logic is already expressive enough to represent all polynomial time functions; it needs neither additives (as in [Gir98]) nor unrestricted weakening (as in [Asp98]).

Since our concern is not normalization but representation, we do not need to introduce a proper term calculus with the polynomial time normalization

property (see [Asp98] and [Ter01] for such term calculi). We rather use the standard λ-calculus and think of **IMLLL** as a typing system for it.

The type assignment rules of **IMLLL** are those of **IMLL** with the following:

$$\frac{x:B \vdash t:A}{x:!B \vdash t:!A} \qquad \frac{\Gamma \vdash t:C}{x:!B,\Gamma \vdash t:C} \qquad \frac{x:!A, y:!A, \Gamma \vdash t:C}{z:!A,\Gamma \vdash t[z/x,z/y]:C} \qquad \frac{\overline{x}:\overline{A},\overline{y}:\overline{B} \vdash t:C}{\overline{x}:\overline{!A},\overline{y}:\overline{\S B} \vdash t:\S C}$$

where $x:B$ may be absent in the first rule. Define $\mathbf{W}$ to be $\forall\alpha.!(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. Then each word $w = i_1 \cdots i_n$, where $n \geq 0$ and $i_k \in \{0,1\}$, is represented by $\underline{w} \equiv \lambda cx.c\underline{i_1} \circ \cdots \circ c\underline{i_n}(x) : \mathbf{W}$, where $\underline{i_k}$ is false if $i_k = 0$, and is true if $i_k = 1$. A function $f:\{0,1\}^* \longrightarrow \{0,1\}^*$ is *represented by* a term t if $f(w) = v \iff t\underline{w} \longrightarrow^* \underline{v}$.

Simulation of polynomial time Turing machines in Light Linear Logic [Gir98,AR02] consists of two parts; one for coding of polynomials and the other for simulation of one-step transition (as well as initialization and output extraction). Since the former is already additive-free in [Gir98], we focus on the latter here.

Let M be a Turing machine with two symbols[6] and 2^n states, and let

$$\delta : \mathit{Symbols} \times \mathit{States} \longrightarrow \mathit{Symbols} \times \mathit{States} \times \{\mathit{left}, \mathit{right}\}$$

be the associated instruction function. A *configuration* of M can be specified by a triple $\langle w_1, w_2, q\rangle$, where the stack $w_1 \in \{0,1\}^*$ describes the non-blank part of the tape to the left of the head, the stack $w_2 \in \{0,1\}^*$ describes the non-blank part of the tape to the right of the head, and $q < 2^n$ denotes the current state. By convention, w_1 is written in the reverse order, and w_1 includes the content of the cell currently scanned.

The configurations are represented by terms of type $ID[\mathbf{B}^n]$, where $ID[A]$ is defined by $ID[A] \equiv \forall\alpha.!(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha \multimap (\alpha \otimes \alpha \otimes A))$. Note that $ID[A]$ is a generalization of $\mathbf{W}$, which allows to encode two words and an additional datum of type A into one term. For example, the configuration $\langle 010, 11, 7\rangle$ is represented by

$$\underline{\langle 010, 11, q\rangle} \equiv \lambda c.\lambda x_1 x_2.(c\underline{0} \circ c\underline{1} \circ c\underline{0}(x_1)) \otimes (c\underline{1} \circ c\underline{1}(x_2)) \otimes \underline{q},$$

where $\underline{q}$ is a term of type $\mathbf{B}^n$ coding $q \in \mathit{States}$.

To simulate one-step transition, it is convenient to divide it into two parts: the decomposition part and the combination part.

Lemma 5 (Decomposition). *There is a term* $\mathsf{dec} : ID[\mathbf{B}^n] \multimap ID[\mathbf{B} \otimes \mathbf{B} \otimes \mathbf{B}^n]$ *such that for any configuration* $\langle i_1 \cdots i_n, j_1 \cdots j_m, q\rangle$,

$$\mathsf{dec}\underline{\langle i_1 \cdots i_n, j_1 \cdots j_m, q\rangle} \longrightarrow^* \underline{\langle i_2 \cdots i_n 0, j_2 \cdots j_m 0, i_1, j_1, q\rangle}.$$

[6] Although more than two symbols are required in general, we describe the two symbols version here for simplicity. The extension is straightforward.

Proof. Define $\mathsf{dec}(\mathit{conf})$ to be $\lambda c.G(\mathit{conf}F(c))$, where the 'step' function F and the 'basis' function G are defined as follows;

$$F(c) \equiv \lambda b_1.\lambda b_2 \otimes w.(b_1 \otimes (cb_2w))$$
$$G(y) \equiv \lambda x_1 x_2.\mathsf{let}\ (y(\underline{0} \otimes x_1)(\underline{0} \otimes x_2))\ \mathsf{be}\ (i_1 \otimes w_1) \otimes (j_1 \otimes w_2) \otimes q\ \mathsf{in}\ (w_1 \otimes w_2 \otimes i_1 \otimes j_1 \otimes q)$$
$$c{:}B \multimap \alpha \multimap \alpha \vdash F(c){:}\mathbf{B} \multimap D \multimap D$$
$$y{:}D \multimap D \multimap D \otimes D \otimes \mathbf{B}^n \vdash G(y){:}\alpha \multimap \alpha \multimap (\alpha \otimes \alpha \otimes \mathbf{B} \otimes \mathbf{B} \otimes \mathbf{B}^n).$$

Here, D stands for $\mathbf{B} \otimes \alpha$. The use of F may be illustrated by

$$(F(c)\underline{i_1}) \circ \cdots \circ (F(c)\underline{i_n})(\underline{0} \otimes x) \longrightarrow^* \underline{i_1} \otimes (c\underline{i_2} \circ \cdots \circ c\underline{i_n} \circ c\underline{0}(x)){:}D,$$

while G plays the roles of initialization and rearrangement of the output. ■

Lemma 6 (Combination). *There is a term* $\mathsf{com} : ID[\mathbf{B} \otimes \mathbf{B} \otimes \mathbf{B}^n] \multimap ID[\mathbf{B}^n]$ *such that for any* $\langle w_1, w_2, i_1, i_2, q\rangle$ *with* $\delta(i_1, q) = (s, q', m)$,

$$\mathsf{com}\underline{\langle w_1, w_2, i_1, i_2, q\rangle} \longrightarrow^* \underline{\langle w_1,\ si_2w_2, q'\rangle},\ \textit{if } m = \textit{left;}$$
$$\longrightarrow^* \underline{\langle i_2sw_1,\ w_2, q'\rangle},\ \textit{if } m = \textit{right.}$$

Proof. Let $\underline{\mathit{left}} \equiv \mathsf{true}$ and $\underline{\mathit{right}} \equiv \mathsf{false}$. By Theorem 2, there is a term delta such that $\mathsf{delta}\, \underline{i_1 q}$ reduces to $\underline{s} \otimes \underline{q'} \otimes \underline{m}$ when $\delta(i_1, q) = (s, q', m)$. Now the key trick is to use the boolean value $\underline{m}$ as "switcher." Observe that $\underline{msi_2}$ reduces to $\underline{s} \otimes \underline{i_2}$ $(\underline{i_2} \otimes \underline{s})$ and $\underline{mw_1w_2}$ reduces to $\underline{w_1} \otimes \underline{w_2}$ $(\underline{w_2} \otimes \underline{w_1})$ when m is *left* (*right*)—thus m can be used to determine on which side of the tape we push symbols, and in what order they are pushed.

Formally, let $\mathsf{cntr}^3 : \mathbf{B} \multimap \mathbf{B}^3$ be a generalized contraction which produces three copies of a given boolean value, and define $G(m, w_1, w_2, i_2, s, c_1, c_2)$ to be

$$\mathsf{let}\ \mathsf{cntr}^3(m)\ \mathsf{be}\ m_1 \otimes m_2 \otimes m_3\ \mathsf{in}\ (\mathsf{let}\ m_1 s i_2\ \mathsf{be}\ j_1 \otimes j_2\ \mathsf{in}\ (\mathsf{let}\ m_2 w_1 w_2\ \mathsf{be}\ v_1 \otimes v_2\ \mathsf{in}\ m_3 v_1(c_1 j_1 \circ c_2 j_2(v_2)))),$$

which is of type

$m : \mathbf{B}, w_1 : \alpha, w_2 : \alpha, i_2 : \mathbf{B}, s : \mathbf{B}, c_1 : \mathbf{B} \multimap \alpha \multimap \alpha, c_2 : \mathbf{B} \multimap \alpha \multimap \alpha \vdash G(m, w_1, w_2, i_2, s, c_1, c_2){:}\alpha \otimes \alpha$. Then, depending on the value of m, it reduces as follows:

$$G(\mathsf{true}, w_1, w_2, i_2, s, c, c) \longrightarrow^* w_1 \otimes (cs \circ ci_2(w_2));$$
$$G(\mathsf{false}, w_1, w_2, i_2, s, c, c) \longrightarrow^* (ci_2 \circ cs(w_1)) \otimes w_2.$$

Finally, the term com is defined to be

$$\lambda z.\lambda c x_1 x_2.\mathsf{let}\ z c x_1 x_2\ \mathsf{be}\ w_1 \otimes w_2 \otimes i_1 \otimes i_2 \otimes q\ \mathsf{in}\ (\mathsf{let}\ \mathsf{delta}\ i_1 q\ \mathsf{be}\ s \otimes q' \otimes m\ \mathsf{in}\ G(m, w_1, w_2, i_2, s, c, c) \otimes q').$$

Although the 'cons' variable c is used three times in com, it does not matter since it is assigned a type $!(\mathbf{B} \multimap \alpha \multimap \alpha)$. ■

The desired one-step transition function is obtained by composing dec and com.

Theorem 5 (IMLLL represents PTIME functions). *A function* $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ *is computable in* DTIME$[n^k]$ *if and only if it is represented by an* **IMLLL** *term* t *of type* $\mathbf{W} \multimap \S^d \mathbf{W}$*, where* $d = O(\log k)$.

In general, cut-elimination in Light Affine Logic, hence in **IMLLL**, requires of time $O(s^{2^{d+1}})$, where s is the size of a proof and d is its *depth*, which counts the nesting of ! and § inferences. The reason why we have a characterization of PTIME above is that we consider a fixed program t, so all the terms $t\underline{w}$ to be evaluated have a fixed depth. On the other hand, CEP allows the depth to vary, hence we get a characterization of doubly-exponential time as in [NM02].

Theorem 6 (2EXPTIME-completeness of IMLLL). *The cut-elimination problem for* **IMLLL** *is complete for* $2\text{EXPTIME} = \bigcup_k \text{DTIME}[2^{2^{n^k}}]$.

5 Multiplicative Soft Linear Logic and EXPTIME

In this section, we show that the intuitionistic mulitplicative fragment **IMSLL** of Soft Linear Logic is expressive enough to represent all polynomial time functions, as conjectured by Lafont [Laf01]. As before, we do not introduce a term calculus for **IMSLL**, thinking of it as a type assignment system for the standard λ-calculus.

The type assignment rules of **IMSLL** are those of **IMLL** with the following:

$$\frac{x_1 : B_1, \ldots, x_m : B_m \vdash t : A}{x_1 : !B_1, \ldots, x_m : !B_m \vdash t : !A}\ m \geq 0 \qquad \frac{x_1 : A, \ldots, x_n : A, \Gamma \vdash t : C}{z : !A, \Gamma \vdash t[z/x_1, \ldots, z/x_n] : C}\ n \geq 0$$

The former is called *soft promotion* and the latter is called *multiplexing*. A term which can be typed without multiplexing is called *generic*. Note that every generic term is a linear λ-term.

The policy of **MSLL** programming is to write each program in a generic way; multiplexing (i.e. duplication) is used only in data. Due to this restriction, simulation of Turing machines is more sophisticated than before. Let M and δ be as before. Let $ID_k[A]$ be $\forall\alpha.!(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap ((\alpha \multimap \alpha)^k \otimes A)$. Each term of type $ID_k[A]$ encodes k words as well as an element of type A. For instance, the configuration $\langle 010, 11, 7\rangle$ is represented by

$$\underline{\langle 010, 11, 7\rangle} \equiv \lambda c.(c\underline{0} \circ c\underline{1} \circ c\underline{0}) \otimes (c\underline{1} \circ c\underline{1}) \otimes \underline{7} : ID_2[\mathbf{B}^n].$$

Lemma 7 (Decomposition). *For every* $k \geq 1$*, there exists a generic term* dec *of type* $ID_k[\mathbf{B}^n] \multimap ID_{2k}[\mathbf{B} \otimes \mathbf{B}^n]$ *such that for any* $\langle i_1 w_1, \ldots, i_k w_k, q\rangle \in (\{0,1\}^+)^k \times 2^n$*,*

$$\mathsf{dec}\underline{\langle i_1 w_1, \ldots, i_k w_k, q\rangle} \longrightarrow^* \underline{\langle w_1, \ldots, w_k, i_1, \ldots, i_k, i_1, q\rangle}.$$

Note that the output contains two occurrences of i_1; the first is a word of length 1 which will be thrown away, while the second is a boolean value which will be used as the current symbol in the next combination part.

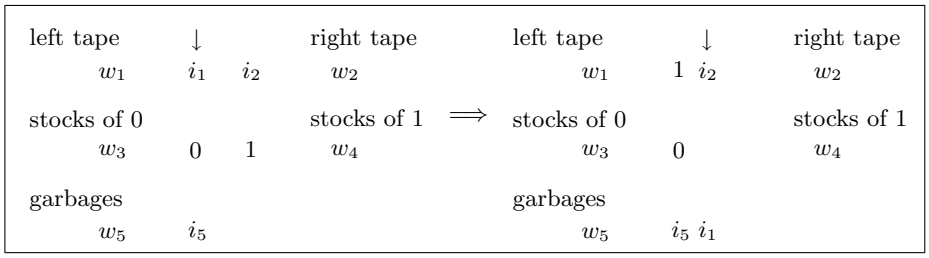


Fig. 1. "Write 1 and move right" (↓ indicates the head position)

Proof. Consider the case $k = 1$. The term dec is defined to be $\lambda z.\lambda c.\lambda z \otimes q.(zF(c)(id \otimes id \otimes \underline{0})) \otimes q$, where the step function F is defined by

$$F(c) \equiv \lambda b.\mathsf{let}\ \mathsf{cntr}(b)\ \mathsf{be}\ b_1 \otimes b_2\ \mathsf{in}\ (\lambda g \otimes h \otimes e.\mathsf{fst}(((h \circ g) \otimes cb_1 \otimes b_2) \otimes e))$$
$$c{:}\mathbf{B} \multimap \alpha \multimap \alpha \vdash F(c){:}\mathbf{B} \multimap ((\alpha \multimap \alpha)^2 \otimes \mathbf{B}) \multimap ((\alpha \multimap \alpha)^2 \otimes \mathbf{B}).$$

The behavior of F is illustrated as follows;

$$(F(c)\underline{i_1}) \circ \cdots \circ (F(c)\underline{i_n})(id \otimes id \otimes \underline{0}) \longrightarrow^* (c\underline{i_2} \circ \cdots \circ c\underline{i_n}) \otimes c\underline{i_1} \otimes c\underline{i_1}.$$

The case $k \geq 2$ is similar, except that we remove all redundant boolean values $i_2, \ldots, i_k$ by weakening for $\mathbf{B}$. ∎

Now let us move on to the combination part. Due to the genericity restriction, we face two difficulties: (i) we cannot create a new tape cell, since the 'cons' variable c of type $!(\mathbf{B} \multimap \alpha \multimap \alpha)$ cannot be used twice; (ii) we cannot simply remove an unnecessary tape cell of type $\alpha \multimap \alpha$, since we do not have weakening for the open type $\alpha \multimap \alpha$. To resolve the first difficulty, we prepare two additional stacks which are filled with $\underline{0}$'s and $\underline{1}$'s respectively, and instead of creating a new cell, we pick one from these two stacks according to the instruction δ. To resolve the second difficulty, we further prepare a 'garbage' stack where unneccesary tape cells are collected. Thus we associate five stacks in total with a configuration. The transition corresponding to "write 1 and move right" is illustrated in Figure 1.

Lemma 8 (Combination). *There is a generic term* com *of type* $ID_{10}[\mathbf{B} \otimes \mathbf{B}^n] \multimap ID_5[\mathbf{B}^n]$ *such that for any* $\langle w_1, \ldots, w_5, i_1, \ldots, i_5, b, q\rangle \in (\{0,1\}^+)^5 \times \{0,1\}^5 \times \{0,1\} \times 2^n$ *with* $\delta(b,q) = (s, q', m)$,

$$\begin{aligned}
&\mathsf{com}\underline{\langle w_1, \ldots, w_5, i_1, i_2, 0, 1, i_5, b, q\rangle} \\
&\longrightarrow^* \underline{\langle w_1,\ 0i_2w_2,\ w_3,\ 1w_4,\ i_1i_5w_5, q'\rangle} && \text{if } s = 0 \text{ and } m = \textit{left;} \\
&\longrightarrow^* \underline{\langle w_1,\ 1i_2w_2,\ 0w_3,\ w_4,\ i_1i_5w_5, q'\rangle} && \text{if } s = 1 \text{ and } m = \textit{left;} \\
&\longrightarrow^* \underline{\langle i_20w_1,\ w_2,\ w_3,\ 1w_4,\ i_1i_5w_5, q'\rangle} && \text{if } s = 0 \text{ and } m = \textit{right;} \\
&\longrightarrow^* \underline{\langle i_21w_1,\ w_2,\ 0w_3,\ w_4,\ i_1i_5w_5, q'\rangle} && \text{if } s = 1 \text{ and } m = \textit{right.}
\end{aligned}$$

Keep in mind that the third and the fourth stacks are to be filled with $\underline{0}$'s and $\underline{1}$'s, so that we always find $\underline{0}$ and $\underline{1}$ at positions i_3 and i_4, respectively.

Proof. As before, there is a term delta such that $\mathsf{delta}\,\underline{bq}$ reduces to $\underline{s} \otimes \underline{q'} \otimes \underline{m}$ when $\delta(b,q) = (s,q',m)$. Define $\mathsf{1Right}$ by

$$\mathsf{1Right} \equiv (i_2 \circ i_4 \circ w_1) \otimes w_2 \otimes (i_3 \circ w_3) \otimes w_4 \otimes (i_1 \circ i_5 \circ w_5)$$
$$w_1 : \alpha \multimap \alpha, \ldots, w_5 : \alpha \multimap \alpha, i_1 : \alpha \multimap \alpha, \ldots, i_5 : \alpha \multimap \alpha \vdash \mathsf{1Right} : (\alpha \multimap \alpha)^5,$$

which corresponds to the case $s = 1$ and $m = \mathit{right}$ (see Figure 1) and gives five stacks as output. $\mathsf{0Left}$, $\mathsf{1Left}$ and $\mathsf{0Right}$ are defined analogously. By using conditionals in Lemma 1 three times, we obtain

$$G(m,s,w_1,\ldots,w_5,i_1,\ldots,i_5) \equiv \begin{pmatrix} \mathsf{if}\ m\ \ \mathsf{then\ if}\ s\ \ \mathsf{then\ 0Left}\ \ \ \mathsf{else\ 1Left} \\ \qquad\qquad\ \ \mathsf{then\ if}\ s\ \ \mathsf{then\ 0Right\ else\ 1Right} \end{pmatrix}$$
$$\mathsf{com} \equiv \lambda z.\lambda c.\mathsf{let}\ zc\ \mathsf{be}\ w_1 \otimes \cdots \otimes w_5 \otimes i_1 \otimes \cdots \otimes i_5 \otimes b \otimes q\ \mathsf{in}$$
$$(\mathsf{let\ delta}\ bq\ \mathsf{be}\ s \otimes q' \otimes m\ \mathsf{in}\ G(m,s,w_1,\ldots,w_5,i_1,\ldots,i_5) \otimes q').$$

■

The rest of coding is basically the same as in [Laf01] except the initialization part, where we need to fill two stacks with $\underline{0}$'s and $\underline{1}$'s. As in [Laf01], we have no idea how to extract a single word as output from the final configuration consisting of five stacks. Instead, we can extract the boolean value which tells us whether the final configuration is accepting or not. Thus the representation theorem below is stated in terms of languages rather than functions in general. Furthermore, due to the genericity restriction, we need to relax the definition of representation slightly. The set $\{0,1\}^*$ is *represented by* $\mathbf{W} \equiv \forall\alpha.!(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap \alpha \multimap \alpha$. We say that a language $X \subseteq \{0,1\}^*$ is *represented by* a term $t : \mathbf{W}^l \multimap \mathbf{B}$ if $w \in X \iff t\underbrace{\underline{w}\cdots\underline{w}}_{l\ times} \longrightarrow^* \mathsf{true}$.

Theorem 7 (IMSLL captures PTIME). *A language $X \subseteq \{0,1\}^*$ is accepted in*

DTIME$[n^k]$ *if and only if it is represented by a generic term t of type $\mathbf{W}^l \multimap \mathbf{B}$, where $l = O(k)$.*

As in the case of **IMLLL**, the complexity of CEP exceeds polynomial time. A difference is that cut-elimination in **IMSLL** only requires exponential time $O(s^{d+2})$ [Laf01]. Hence we have:

Theorem 8 (EXPTIME-completeness of IMSLL). *The cut-elimination problem for* **IMSLL** *is complete for* EXPTIME *under logspace reducibility.*

Proof (sketch). Suppose that a language X be accepted by a Turing machine M in time $O(2^{n^k})$. For each word w of length n, the following terms (of suitable types) can be constructed in $O(k \log n)$ space: (1) the Church representation $\underline{w}$ of w; (2) the term $\mathsf{exp}(n^k)$ of size and depth $O(n^k)$, which reduces to the tally integer $\underline{2^{n^k}}$; (3) the term $\mathsf{M}_{n,k}(w,x)$ with two variables w and x, which outputs the result of x-steps computation on the input w, when w is of length n and

x is of the same type as $\mathsf{exp}(n^k)$. By putting them together, we obtain a term $\mathsf{M}_{n,k}(\underline{w}, \mathsf{exp}(n^k))$ which normalizes to true if and only if $w \in X$. ■

Acknowledgments. We wish to thank Patrick Baillot and Marco Pedicini for very stimulating discussions, and Jean-Yves Girard, Stefano Guerrini, Yves Lafont, Satoshi Matsuoka, Peter Neergaard, Peter Selinger, Izumi Takeuti and Rene Vestergaard for a lot of useful comments.

References

[Asp98] A. Asperti. Light affine logic. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 300–308, 1998.

[AR02] A. Asperti and L. Roversi. Intuitionistic light affine logic (proof-nets, normalization complexity, expressive power, programming notation). *ACM Transactions on Computational Logic*, 3(1):1–39, 2002.

[dF03] L. Tortora de Falco. The additive multiboxes. *Annals of Pure and Applied Logic*, 120(1):65–102, 2003.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 14(3):175–204, 1998.

[GJ78] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1978.

[Hin89] J. R. Hindley. BCK-combinators and linear λ-terms have types. *Theoretical Computer Science*, 64:97–105, 1989.

[Lad75] R. E. Ladner. The circuit value problem is logspace complete for P. *SIGACT News*, 7(1):18–20, 1975.

[Laf01] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, to appear.

[Lin95] P. D. Lincoln. Deciding provability of linear logic formulas. In *Advances in Linear Logic*, London Mathematical Society Lecture Notes Series, Volume 222, Cambridge University Press, 1995, 109–122.

[MR02] H. G. Mairson and X. Rival. Proofnets and context semantics for the additives. *Computer Science Logic (CSL) 2002*, 151–166.

[Mai03] H. G. Mairson. Linear lambda calculus and polynomial time. *Journal of Functional Programming*, to appear.

[NM02] P. M. Neergaard and H. G. Mairson. LAL is square: representation and expressiveness in light affine logic. Presented at the Fourth International Workshop on Implicit Computational Complexity, 2002.

[Sch01] A. Schubert. The Complexity of β-Reduction in Low Orders. *Typed Lambda Calculi and Applications (TLCA) 2001*, 400–414.

[Sta79] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.

[Ter01] K. Terui. Light affine lambda calculus and polytime strong normalization. In *Proceedings of the sixteenth annual IEEE symposium on Logic in Computer Science*, pages 209–220, 2001. The full version is available at http://research.nii.ac.jp/∼terui.