A PolyTime Functional Language from Light Linear Logic*

Patrick Baillot[1], Marco Gaboardi[2], and Virgile Mogbil[3]

[1] LIP - UMR 5668 CNRS ENS-Lyon UCBL INRIA - Université de Lyon
[2] Dipartimento di Informatica, Università degli studi di Torino
[3] LIPN - UMR 7030, CNRS - Université Paris 13
patrick.baillot@ens-lyon.fr, gaboardi@di.unito.it,
virgile.mogbil@lipn.univ-paris13.fr

Abstract. We introduce a typed functional programming language LPL (acronym for Light linear Programming Language) in which all valid programs run in polynomial time, and which is complete for polynomial time functions. LPL is based on lambda-calculus, with constructors for algebraic data-types, pattern matching and recursive definitions, and thus allows for a natural programming style. The validity of LPL programs is checked through typing and a syntactic criterion on recursive definitions. The higher order type system is designed from the ideas of Light linear logic: stratification, to control recursive calls, and weak exponential connectives §, !, to control duplication of arguments.

1 Introduction

Implicit computational complexity (ICC). This line of research aims at characterizing complexity classes not by external measuring conditions on a machine model, but instead by investigating restrictions on programming languages or calculi which imply a complexity bound. So for instance characterizing the class PTIME in such a framework means that all the programs of the framework considered can be evaluated in polynomial time (*soundness*), and that conversely all polynomial time *functions* can be represented by a program of the framework (*extensional completeness*).
The initial motivation was to provide new characterizations of complexity classes of functions to bring some insight on their nature [1–4]. In a second step, e.g. [5, 6], the issue was raised of using these techniques to design some ways of statically verifying the complexity of concrete programs. Some efforts in this direction have been done also following other approaches, e.g. [7–9]. For this point of view it is quite convenient to consider a general programming language or calculus, and to state the ICC condition as a *criterion* on programs, which can be checked statically, and which ensures on the validated programs a time or space complexity bound. In this respect the previous extensional completeness is of limited interest, and one is interested in designing criteria which are *intensionally expressive*, that is to say which validate as many interesting programs as possible. Note that for a Turing-complete language the class of

* Partially supported by project ANR-08-BLANC-0211-01 ”COMPLICE”.

PTIME programs is non recursively enumerable, and so an intensionally complete criterion would not be decidable. Actually we think that three aspects should be taken into consideration for discussing intensional expressivity:

1. what are the algorithmic schemes that can be validated by the criterion,
2. what are the features of the programming language: e.g. higher-order functional language, polymorphism, exceptions handling ...
3. how effective is the criterion: what is the complexity of the corresponding decision problem.

Results and methodology. The main contribution of the present work is the definition of LPL (acronym for Light linear Programming Language), a typed functional programming language inspired by Light linear logic satisfying an ICC criterion ensuring a PTIME bound. LPL improves with respect to previous PTIME linear logic inspired languages in aspects 1 and 2 above, since it combines the advantages of a user-friendly and expressive language and of modular programming. The distinguishing feature of LPL is the combination of

– higher-order types by means of a typed λ-calculus,
– pattern-matching and recursive definitions by means of a `LetRec` construction,
– a syntactic restriction avoiding intrinsically exponential recursive definitions and a light type system ensuring duplication control,

in such a way that all valid typed programs run in polynomial time, and all polynomial time functions can be programmed by valid typed programs.
A difficulty in dealing with λ-calculus and recursion is that we can easily combine apparently harmless terms to obtain exponential time programs like the following one

$$\lambda \mathtt{x}.\mathtt{x}(\lambda \mathtt{y}.\ \mathtt{mul}\ 2\ \mathtt{y})1$$

where `mul` is the usual recursive definition for multiplication. Such a term is apparently harmless, but for each Church numeral $\underline{\mathtt{n}} = \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}^n\mathtt{z}$ this program returns the (standard) numeral 2^n. In order to prevent this kind of programs, to achieve polynomial time soundness, a strict control over both the numbers of recursive calls and beta-reduction steps is needed. Moreover, the extension to higher order in the context of polynomial time bounded computations is not trivial. Consider the classical `foldr` higher order function; its unrestricted use leads to exponential time programs. E.g. let `ListOf2` be a program that given a natural number n returns a list of 2 of length n. Then, the following program is exponential in its argument:

$$\lambda \mathtt{x}.\mathtt{foldr}\ \mathtt{mul}\ 1\ (\mathtt{ListOf2}\ \mathtt{x})$$

For these reasons, besides the syntactic restriction avoiding intrinsically exponential recursive definitions, we impose a strict typing discipline inspired by the ideas of Light linear logic. In λ-calculus, Light linear logic allows to bound the number of beta-steps, using weak exponential connectives ! and $\S$ in order to control duplication of arguments and term stratification in order to control the size. The syntactic restriction of function definitions limits the number of recursive steps for *one* function call. But this is not enough since function calls appear at run time and the size of values can increase during the computation. Our type system addresses these issues, and a key point for that is the typing rule for recursive definition. In particular, a function of type $\mathbb{N} \multimap \mathbb{N}$ can increase the size of its input by at most a constant, while a function of type $\mathbb{N} \multimap \S\mathbb{N}$

can increase it by a (fixed) polynomial bound. For a recursive definition of the shape $\mathtt{f\ t} = \mathtt{M}\{\mathtt{f\ t'}\}$, the typing rule ensures that the context $\mathtt{M}$ does not *increase the size too much*, and it types the function $\mathtt{f}$ accordingly. In this way the type system allows to bound both the number of beta-steps and the size of values, and together with the syntactic restriction this results in a PTIME bound on execution.
The typing restrictions on higher order functions are not too severe. Indeed, we can write in a natural way some interesting programs using higher order functions without exponential blow up. For instance consider again the `foldr` function, we can type a term representing one of its classical uses as

$$\lambda \mathtt{x}.\mathtt{foldr\ add\ 0\ x}$$

About the methodology we use, we stress that we do not aim at proving the properties of LPL by encoding it into Light linear logic. Instead, we take inspiration from it and we adapt the *abstract* structure of its PTIME soundness proof to our new setting. Moreover, our guideline is to follow a gradual approach: we propose here a strict criterion, that has the advantage of handling naturally higher-order. We hope that once this step has been established we might study how the criterion can be relaxed in various ways. Indeed, the choice of using a *combined criterion*, i.e. a first condition ensuring termination, and a second one dealing with controlling the size, will be an advantage for future works. In particular, by relaxing either one of the two conditions one can explore generalizations, as well as different criteria to characterize other complexity classes. Finally we think that the ICC criterion we give can be effectively checked since this is the case for λ-calculus [10], but we leave the development of this point for future work.

Related works. *Higher-order calculi: linear logic and linear type systems.* Linear logic [11] was introduced to control in proof theory the operations of duplication and erasing, thanks to specific modalities !, ?. Through the proofs-as-programs correspondence it provided a way to analyze typed λ-calculus. The idea of designing alternative *weak versions* of the modalities implying a PTIME bound on normalization was proposed in [12] and led to *Light Linear Logic* (LLL) in [4] and *Soft Linear Logic* (SLL) in [13]. Starting from the principles underlying these logics different PTIME term languages have been proposed [14, 15]. In a second step [16] type systems as criteria for λ-calculus were designed out of these logics, like DLAL [17] and STA [18]. This approach completely fits in the proofs-as-programs paradigm, thereby offering some advantages from the programming language point of view (point 2 above): it encompasses higher-order and polymorphism. The drawback is that data are represented in λ-calculus and so one is handling encodings of data-types analogous to Church integers. Moreover the kinds of algorithms one can represent is very limited (point 1). However testing the criterion can be done efficiently (point 3) thanks to some polynomial time type inference algorithms [10, 19].
In [20] the authors propose a language for PTIME extending to higher-order the characterizations based on *ramification* [1, 2]. The language is a *ramified* variant of Gödel's system T where higher-order arguments cannot be duplicated, which is quite a strong restriction. Moreover, the system T style does not make it as easy to program in this system as one would like. Another characterization of PTIME by means of a restriction of System T in a linear setting has been studied in [21, 22].
In [5], Hofmann proposed a typed λ-calculus with recursor (essentially a variant of

system T), LFPL, overcoming the problems of ramification, which allows to represent *non-size-increasing* programs and enjoys a PTIME bound. This improves on point 1 by allowing to represent more algorithms and by featuring higher-order types. However, the restriction on higher-order arguments similar to the one in [20] and the system T programming style make it quite far from ordinary functional languages.
First-order calculi and interpretations. Starting from works on *ramification* [1, 2], Marion and collaborators have generalized them progressively by first replacing primitive recursion by termination orderings [23], and then ramification by notions of *quasi-interpretation* [6, 24, 25] and *sup-interpretation* [26, 27] on a first-order functional language with recursion and pattern-matching. These latter notions are semantic, inspired from polynomial interpretations, and essentially statically provide a bound on the size of the values computed during the evaluation. If a program admits both a termination ordering *and* a quasi-interpretation or sup-interpretation of a certain shape, then it admits a complexity bound. The main benefit of this method is that more algorithms are validated (point 1) than in the ramification-based frameworks. An advantage of our present contribution however is that it handles higher-order and that type checking is easier than checking of quasi-interpretations .

Outline. We introduce in Section 2 the language LPL, its type system and the syntactic criterion required on programs, and then provide some programming examples. In Section 3 we define an extended language, eLPL, where a stratification is explicit in the term syntax, and which is meant for translating and executing LPL typed programs. We then show that all PTIME functions can be computed in LPL (Section 4). Finally, Section 5 establishes our main result, that all LPL programs can be executed in polynomial time w.r.t. the size of the input.

2 LPL

We introduce the language LPL, an extension of λ-calculus with constructors for algebraic data types, pattern matching and recursive function definitions. In order to limit the computational complexity of programs we need to impose some restrictions. To achieve polynomial time properties two key ingredients are used: a syntactic criterion and a type system.
The syntactic criterion imposes restrictions to recursive schemes in order to avoid the ones which are intrinsically exponential. The type system allows through a stratification over terms to avoid the dangerous nesting of recursive definition.

2.1 The syntax

Let the denumerable sets Var, PVar, Cst and Fct be respectively a set of *variables*, a set of *pattern variables*, a set of *constructors* and a set of *function symbols*. Each constructor $\mathtt{c} \in \mathrm{Cst}$ and each function symbol $\mathtt{F} \in \mathrm{Fct}$ has an *arity* $n \geq 0$: the number of arguments that it expects. In particular a constructor $\mathtt{c}$ of arity 0 is a *base constant*.
The program syntax is given in Table 1 where $\mathtt{x} \in \mathrm{Var}$, $\mathtt{X} \in \mathrm{PVar}$, $\mathtt{c} \in \mathrm{Cst}$ and $\mathtt{F} \in \mathrm{Fct}$. Among function symbols we distinguish a subset of symbols which we will call *basic functions* and denote as $\underline{\mathtt{F}}, \underline{\mathtt{G}}, \ldots$ We use the symbol $\varkappa$ to denote either a variable or a pattern variable. Observe that values and patterns are subsets of terms.
The size $|\mathtt{M}|$ of a term $\mathtt{M}$ is the number of symbols occurring in it. The size of patterns and

$\mathtt{p} ::= \mathtt{M} \mid \mathtt{LetRec}\ \mathtt{d_F}\ \mathtt{in}\ \mathtt{p}$	program definition
$\mathtt{v} ::= \mathtt{c}\ \mathtt{v_1} \cdots \mathtt{v_n}$	value definition
$\mathtt{t} ::= \mathtt{X} \mid \mathtt{c}\ \mathtt{t_1} \ldots \mathtt{t_n}$	pattern definition
$\mathtt{d_F} ::= \mathtt{F}\ \mathtt{t_1} \ldots \mathtt{t_n} = \mathtt{N} \mid \mathtt{d_F}, \mathtt{d_F}$	function definition
$\mathtt{M}, \mathtt{N} ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{X} \mid \mathtt{F} \mid \lambda \mathtt{x}.\mathtt{M} \mid \mathtt{MM} \mid$	term definition

Table 1. LPL term language definition

programs are defined similarly. We denote by $n_o(\varkappa, \mathtt{M})$ the number of occurrences of $\varkappa$ in $\mathtt{M}$. Let $\mathtt{s} \in \mathrm{Cst} \cup \mathrm{Fct}$ be a symbol of arity n, then we will often write $\mathtt{s}(\mathtt{M}_1, \ldots, \mathtt{M}_n)$ or $\mathtt{s}(\overrightarrow{\mathtt{M}})$ instead of $\mathtt{s}\ \mathtt{M}_1 \ldots \mathtt{M}_n$.
The set Cst includes the usual constructors $\mathtt{s}$ of arity one and the base constant $\mathtt{0}$ for natural numbers, the constructor : of arity two and the base constant $\mathtt{nil}$ for lists of natural numbers, $\mathtt{node}$ of arity three and the base constant ϵ for binary trees with node natural numbers.
A *function definition* $\mathtt{d_F}$ for the function symbol $\mathtt{F}$ of arity n is a sequence of *definition cases* of the shape $\mathtt{F}(\mathtt{t_1}, \cdots, \mathtt{t_n}) = \mathtt{N}$ where $\mathtt{F}$ is applied to *patterns* $\mathtt{t_1}, \ldots, \mathtt{t_n}$, the free variables of $\mathtt{N}$ are a subset of the free variables of $\mathtt{t_1}, \ldots, \mathtt{t_n}$ (thus pattern variables), and $\mathtt{N}$ is normal for the reduction (which will be given in Def. 4). Besides in a definition case:

1. if $\mathtt{F}$ is a basic function $\underline{\mathtt{G}}$, then $\mathtt{N}$ does not contain any function symbol,
2. if $\mathtt{F}$ is not a basic function, then (i) $\mathtt{N}$ does not contain any basic function symbol, and (ii) every occurrence of $\mathtt{F}$ in $\mathtt{N}$ appears in subterms of the form $\mathtt{F}(\mathtt{t}^1_1, \ldots, \mathtt{t}^1_n), \ldots,$ $\mathtt{F}(\mathtt{t}^k_1, \ldots, \mathtt{t}^k_n)$; these subterms are called the *recursive calls* of $\mathtt{F}$ in $\mathtt{N}$.

Patterns are *linear* in the sense that a pattern variable $\mathtt{X}$ cannot appear several times in a given pattern. Moreover we assume that patterns $\mathtt{t_1}, \ldots, \mathtt{t_n}$ in the l.h.s. of a definition case have distinct sets of pattern variables. The notion of sub-term is adapted to patterns: we denote by $\mathtt{t}' \prec \mathtt{t}$ the fact that $\mathtt{t}'$ is a strict sub-pattern of $\mathtt{t}$. As usual $\preceq$ is the reflexive closure of $\prec$. Patterns $\mathtt{t}$ and $\mathtt{t}'$ such that $\mathtt{t} \npreceq \mathtt{t}'$ and $\mathtt{t}' \npreceq \mathtt{t}$ are *incomparable*.
A *program* is a term $\mathtt{M}$ without free pattern variables preceded by a sequence of function definitions $\mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}$ defining all the function symbols occurring in $\mathtt{M}$. We ask that every function definition $\mathtt{d_{F_{\mathit{i}}}}$ uses only the function symbols $\mathtt{F}_1, \ldots, \mathtt{F}_i$. We write a program of the shape $\mathtt{LetRec}\ \mathtt{d_{F_1}}\ \mathtt{in} \cdots \mathtt{LetRec}\ \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M}$ simply as $\mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}$ $\mathtt{in}\ \mathtt{M}$. As usual we consider programs up to renaming of bound variables.
A *substitution* σ is a mapping replacing variables by terms. This is used to define the notion of matching which is essential for the reduction mechanism of our language.
Let t be a pattern. We say the term $\mathtt{M}$ *matches* t if and only if there exists a substitution σ such that $\mathtt{M} = \sigma(t)$. Analogously, given a definition case $\mathtt{F}(\mathtt{t_1}, \ldots, \mathtt{t_n}) = \mathtt{N}$, the term $\mathtt{M}$ *matches* it if and only if there exists a substitution σ such that $\mathtt{M} = \mathtt{F}(\sigma(\overrightarrow{t}))$.
We say a sequence $\mathtt{d}_1, \ldots, \mathtt{d}_n$ of function definition cases for the function symbol $\mathtt{F}$ of arity n is *exhaustive* if for every sequence of values $V_1, \ldots, V_n$ such that $\mathtt{F}(\overrightarrow{V})$ is typable, there exists a unique $1 \leq i \leq n$ such that $\mathtt{F}(\overrightarrow{V})$ matches the l.h.s. of $\mathtt{d}_i$.
A program p is *well defined* if and only if all the function definitions in it are exhaustive.

2.2 Syntactic Criterion

As we have already stressed, the first ingredient to ensure the intended complexity properties for LPL programs is a *syntactic criterion*.
Consider a definition case $\mathtt{F}(\mathtt{t_1}, \ldots, \mathtt{t_n}) = \mathtt{M}$. We say t is *recursive* if it contains some

recursive calls $\mathtt{F}(\mathtt{t}_1^1,\ldots,\mathtt{t}_n^1)$, $\ldots,\mathtt{F}(\mathtt{t}_1^m,\ldots,\mathtt{t}_n^m)$ ($m \geq 1$) in M, and *base* otherwise. Note that basic functions by definition only have base definition cases. We now need the following notion of safe definition cases.

Definition 1 (Safe definition case). *Let* $\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) = \mathtt{M}$ *be a definition case. It is* safe *if for every recursive call* $\mathtt{F}(\mathtt{t}_1^1,\ldots,\mathtt{t}_n^1),\ldots,\mathtt{F}(\mathtt{t}_1^m,\ldots,\mathtt{t}_n^m)$ *of* F *in* M*, we have:*
(i) $\forall k, \forall i : \mathtt{t}_i^k \preceq \mathtt{t}_i$, *(ii)* $\forall k, \exists j : \mathtt{t}_j^k \prec \mathtt{t}_j$, *(iii)* $\forall j, \forall k \neq l, \mathtt{t}_j^k \not\preceq \mathtt{t}_j^l$ *and* $\mathtt{t}_j^l \not\preceq \mathtt{t}_j^k$.

Note that this condition is trivially satisfied by base definition cases, and thus by basic functions. The syntactic criterion for LPL program can now be defined:

Definition 2 (Syntactic Criterion). *An* LPL *program* M *satisfies the* syntactic criterion *if and only if every definition case in it is safe.*

We now state some definitions and properties that will be useful in the sequel.

Definition 3 (Matching argument). *Let* $\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) = \mathtt{M}$ *be a definition case. Every position of index* j *such that* $\mathtt{t}_j$ *is not a pattern variable* X *is a* matching position.
The set $\mathcal{R}(\mathtt{F})$ *is the set of all positions* j *for which there exists a definition case of* F *where* j *is in matching position. The* matching arguments *of a function symbol* F *are the arguments in a matching position of* $\mathcal{R}(\mathtt{F})$.

Note that in Definition 1 the condition (ii) asks that for every recursive call there exists at least one *recurrence argument*. Every such recurrence argument is a matching argument. Moreover conditions (iii) and (ii) imply that in safe definition cases making recursion over integer or list there is at most one recursive call. This to avoid exponential functions like the following: $\mathtt{exp}(\mathtt{s}\ \mathtt{X}) = (\mathtt{exp}\ \mathtt{X}) + (\mathtt{exp}\ \mathtt{X})$. Nevertheless, we have functions with more recursive calls over trees, for example:

$$\mathtt{Tadd}\ (\mathtt{node}\ \mathtt{X}\ \mathtt{Y}\ \mathtt{Z})\ (\mathtt{node}\ \mathtt{X}'\ \mathtt{Y}'\ \mathtt{Z}') = \mathtt{node}\ (\mathtt{X}+\mathtt{X}')\ (\mathtt{Tadd}\ \mathtt{Y}\ \mathtt{Y}')\ (\mathtt{Tadd}\ \mathtt{Z}\ \mathtt{Z}').$$

Safe definition cases have the following remarkable property.

Lemma 1. *Let* $\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) = \mathtt{M}$ *be a safe definition case and let the recursive calls of* F *in* M *be* $\mathtt{F}(\mathtt{t}_1^1,..,\mathtt{t}_n^1),\ldots,\mathtt{F}(\mathtt{t}_1^m,..,\mathtt{t}_n^m)$*. Then* $\sum_{i=1}^n |\mathtt{t}_i| > \sum_{k=1}^m (\sum_{i=1}^n |\mathtt{t}_i^k|)$.

2.3 Reduction

The computational mechanism of LPL will be the reduction relation obtained by extending the usual β-reduction with rewriting rules for the LetRec construct.
We denote by $\mathtt{M}\{\}$ a *context*, that is to say a term with a hole, and by $\mathtt{M}\{\mathtt{N}\}$ the result of substituting the term N for the hole.

Definition 4. *The reduction relation* $\to_{\mathtt{L}}$ *is the contextual closure of:*

- *the relation* $\to_\beta$ *defined as:* $(\lambda \mathtt{x}.\mathtt{M})\mathtt{N} \to_\beta \mathtt{M}[\mathtt{N}/\mathtt{x}]$,
- *the relation* $\to_\gamma$ *defined for basic functions* $\underline{\mathtt{F}}$ *by: if* $\exists i\ \sigma(\overrightarrow{\mathtt{t}_i}) = \overrightarrow{\mathtt{N}}$ *then*
$$\mathtt{LetRec}\ \underline{\mathtt{F}}(\mathtt{t}_1) = \mathtt{M}_1,\ldots,\underline{\mathtt{F}}(\mathtt{t}_n) = \mathtt{M}_n\ \mathtt{in}\ \mathtt{M}\{\underline{\mathtt{F}}(\overrightarrow{\mathtt{N}})\} \to_\gamma$$
$$\mathtt{LetRec}\ \underline{\mathtt{F}}(\mathtt{t}_1) = \mathtt{M}_1,\ldots,\underline{\mathtt{F}}(\mathtt{t}_n) = \mathtt{M}_n\ \mathtt{in}\ \mathtt{M}\{\sigma(\mathtt{M_i})\}$$
- *and of the relation* $\to_{\mathtt{Rec}}$ *defined as* $\to_\gamma$ *but for non-basic functions.*

We write $\to_{\gamma_{\underline{\mathtt{F_i}}}}$ *(resp.* $\to_{\mathtt{Rec}_{\mathtt{F_i}}}$*) instead of* $\to_\gamma$ *(resp.* $\to_{\mathtt{Rec}}$*) when we want to stress which function* $\underline{\mathtt{F}}_i$ *(resp.* $\mathtt{F}_i$*) has been triggered. As usual* $\to_{\mathtt{L}}^*$ *denotes the reflexive and transitive closure of* $\to_{\mathtt{L}}$.

We remark that the syntactic criterion alone implies that a program satisfying it cannot have an infinite $\to_{\mathtt{Rec}}$ reduction sequence.

$$\frac{}{\vdash \mathtt{c} : \mathcal{T}(\mathtt{c})} \qquad \frac{}{\vdash \mathtt{F} : \mathcal{T}(\mathtt{F})}$$

1: Constructors and functions

$$\frac{}{; \varkappa : A \vdash \varkappa : A}\ (Ax) \quad \frac{\Gamma_1; \Delta_1 \vdash \mathtt{M} : B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash \mathtt{M} : B}\ (W) \quad \frac{\Gamma, \varkappa_1 : A, \varkappa_2 : A; \Delta \vdash \mathtt{M} : B}{\Gamma, \varkappa : A; \Delta \vdash \mathtt{M}[\varkappa/\varkappa_1, \varkappa/\varkappa_2] : B}\ (C)$$

$$\frac{\Gamma; \Delta, \mathtt{x} : A \vdash \mathtt{M} : B}{\Gamma; \Delta \vdash \lambda\mathtt{x}.\mathtt{M} : A \multimap B}\ (\multimap I) \qquad \frac{\Gamma_1; \Delta_1 \vdash \mathtt{M} : A \multimap B \quad \Gamma_2; \Delta_2 \vdash \mathtt{N} : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash \mathtt{MN} : B}\ (\multimap E)$$

$$\frac{\Gamma, \mathtt{x} : A; \Delta \vdash \mathtt{M} : B}{\Gamma; \Delta \vdash \lambda\mathtt{x}.\mathtt{M} : !A \multimap B}\ (\Rightarrow I) \qquad \frac{\Gamma_1; \Delta \vdash \mathtt{M} : !A \multimap B \quad ; \Gamma_2 \vdash \mathtt{N} : A \quad \Gamma_2 \subseteq \{\varkappa : C\}}{\Gamma_1, \Gamma_2; \Delta \vdash \mathtt{MN} : B}\ (\Rightarrow E)$$

$$\frac{; \Gamma, \Delta \vdash \mathtt{M} : A}{\Gamma; \S\Delta \vdash \mathtt{M} : \S A}\ (\S I) \quad \frac{\Gamma_1; \Delta_1 \vdash \mathtt{N} : \S A \quad \Gamma_2; \mathtt{x} : \S A, \Delta_2 \vdash \mathtt{M} : B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, \vdash \mathtt{M}[\mathtt{N}/\mathtt{x}] : B}\ (\S E)$$

2: Terms

$$\frac{\Gamma; \Delta \vdash \mathtt{F}(\overrightarrow{\mathtt{t}_i}) : B \quad \Gamma; \Delta \vdash \mathtt{N}_i : B \quad \rhd \mathtt{d}_\mathtt{F} : B}{\rhd(\mathtt{F}(\overrightarrow{\mathtt{t}_i}) = \mathtt{N}_i), \mathtt{d}_\mathtt{F} : B}\ (D) \qquad \frac{\Gamma; \Delta \vdash \mathtt{p} : A \quad \rhd \mathtt{d}_\mathtt{F} : B}{\Gamma; \Delta \vdash \mathtt{LetRec}\ \mathtt{d}_\mathtt{F}\ \mathtt{in}\ \mathtt{p} : A}\ (R)$$

3: Recursive definitions and programs

Table 2. LPL Typing rules

2.4 Type system

The fundamental ingredient to ensure the complexity properties of LPL is the type system. It allows to derive different kinds of typing judgments. One assigns types to terms, another one assigns types to programs and the last one assigns types to function definitions. We start with a set of *ground* types containing $\mathbb{B}_n$, $\mathbb{N}$, $\mathbb{L}_n$, $\mathbb{L}$, $\mathbb{T}$ representing respectively finite types with $n \geq 1$ elements, unary integers (natural numbers), lists over $\mathbb{B}_n$, lists of unary integers and binary trees with unary integers at the nodes. Ground types can also be constructed using products $\mathbb{D}_1 \times \mathbb{D}_2$ whose elements are of the form $(\mathtt{p}\ \mathtt{d}_1\ \mathtt{d}_2)$ where $\mathtt{d}_\mathtt{i}$ is an element of $\mathbb{D}_\mathtt{i}$ ($i = \{1, 2\}$). The constructor p has type $\mathbb{D}_1 \multimap \mathbb{D}_2 \multimap \mathbb{D}_1 \times \mathbb{D}_2$. This set of ground types could easily be extended to all the usual standard data types. Types are defined by the following grammars:

$$\mathbb{D} ::= \mathbb{B}_n \mid \mathbb{N} \mid \mathbb{L}_n \mid \mathbb{L} \mid \mathbb{T} \mid \mathbb{D} \times \mathbb{D} \qquad \text{and} \qquad A ::= \mathbb{D} \mid A \multimap B \mid !A \multimap B \mid \S A$$

The type $!A \multimap B$ is the translation of the intuitionistic implication $A \Rightarrow B$ in linear logic. It uses the modality ! to manage typing of non-linear variable in a program. In particular, $!A \multimap B$ is a type for functions that can use their argument several times, while $A \multimap B$ (when A is not of the form $!A'$) is a type of functions that use their argument at most once. It is worth noting that the modality ! here cannot be nested, i.e. $!!A$ does not occur in types since ! is used only in combination with $\multimap$, on its left hand side. The other modality $\S A$ is used in Light linear logic [4] (and in DLAL) to guarantee a PTIME bound normalization. A possible intuition is that it marks different levels of computation, like in ramified type systems: a function defined by recursion over an argument of type D (a data type) will produce a result at higher level, so of type $\S A$, for some A. A formula A is *modal* if it is of the form $A = \S B$ or $!B$. We write $\dagger$ for modalities in $\{!, \S\}$, and $\dagger^n A = \dagger(\dagger^{n-1} A)$, $\dagger^0 A = A$.
Now we give types to constructors and functions:

Definition 5. *To each constructor or function symbol s, of arity n, a fixed type is associated, denoted $\mathcal{T}(s)$:*

- *If $s = \mathtt{c}$ or $\underline{\mathtt{F}}$ then $\mathcal{T}(s) = \mathbb{D}_1 \multimap \cdots \multimap \mathbb{D}_n \multimap \mathbb{D}_{n+1}$,*
- *If $s = \mathtt{F}$ a non-basic function then $\mathcal{T}(\mathtt{F}) = !^{i_1}\S^{j_1} A_1 \multimap \cdots \multimap !^{i_n}\S^{j_n} A_n \multimap \S^j A$ with:*

i) $j \geq 1$ *and* $0 \leq i_r \leq 1$ *for any* $1 \leq r \leq n$*,*

ii) *for* $1 \leq r \leq n$*, if* $r \in \mathcal{R}(\mathtt{F})$ *then* A_r *is a ground type* $\mathbb{D}$ *and* $i_r = j_r = 0$*; otherwise* $i_r + j_r \geq 1$*.*

Where for $1 \leq i \leq n+1$*, the* $\mathbb{D}_i$ *are ground types, and the* A_i *and* A *are non-modal.*

Example 1. For the ground type $\mathbb{N}$ of natural numbers we have: $\mathcal{T}(\mathtt{0}) = \mathbb{N}, \mathcal{T}(\mathtt{s}) = \mathbb{N} \multimap \mathbb{N}$. For the ground type $\mathbb{L}$ of finite lists of natural numbers we have: $\mathcal{T}(\mathtt{nil}) = \mathbb{L}$, $\mathcal{T}(:) = \mathbb{N} \multimap \mathbb{L} \multimap \mathbb{L}$. For the ground type $\mathbb{T}$ of finite binary trees with natural numbers as node we have: $\mathcal{T}(\epsilon) = \mathbb{T}$, $\mathcal{T}(\mathtt{node}) = \mathbb{N} \multimap \mathbb{T} \multimap \mathbb{T} \multimap \mathbb{T}$.

We design a declarative type assignment system (Table 2) to favor simplicity rather than to make type inference easy. So the typing rules will not be syntax-directed but they could be. Contexts, denoted $\Gamma, \Delta, \ldots$ are sets of assignments of the shape $\mathtt{x}:A$ or $\mathtt{X}:A$ where A is a type. Note that there are no symbols of function or constructor in contexts. The *type judgments for terms and programs* have the shape $\Gamma; \Delta \vdash \mathtt{M}:A$ and $\Gamma; \Delta \vdash \mathtt{p}:A$ respectively, where Γ and Δ are two distinct contexts, $\mathtt{M}$ is a term and $\mathtt{p}$ is a program, while A is a type. The context Γ is called *non-linear*, while Δ is *linear* (in fact *affine*): the type system will ensure that variables from Δ occur at most once in the term $\mathtt{M}$ or the program $\mathtt{p}$. If Δ is $\varkappa_1 : A_1, \ldots, \varkappa_n : A_n$ then $\S\Delta$ will stand for $\varkappa_1 : \S A_1, \ldots, \varkappa_n : \S A_n$. The *type judgments for function definitions* have the shape $\triangleright \mathtt{d}_\mathtt{F} : A$, where $\mathtt{d}_\mathtt{F}$ is a definition of $\mathtt{F}$ (possibly not completed yet) and A is a type.
We now explain some rules. In binary rules, like $(\Rightarrow\ E)$, the contexts of the two premises have disjoint sets of variables. The typing rules for terms in Table 2.2 are essentially those of the type system DLAL [17] for λ-calculus but extended to pattern variables. Note that the linear application $(\multimap E)$ is unrestricted, while in the non-linear one $(\Rightarrow E)$: the argument $\mathtt{N}$ should have at most one free variable $\varkappa$, which is linear; in the conclusion, $\varkappa$ then has a non-linear status. This is a key to bound β-reduction steps. The typing rules for definitions are presented in Table 2.3 and together with those for function symbols, are the main novelty of the present system. They need some comments. The rule (D) serves to add a definition case to a partial definition $\mathtt{d}_\mathtt{F}$ of $\mathtt{F}$. The new definition typed is then $\mathtt{d}'_\mathtt{F} = (\mathtt{F}(\overrightarrow{\mathtt{t}_i}) = \mathtt{N}_i), \mathtt{d}_\mathtt{F}$. Whereas the rule (R) then serves to form a new program from a program and a definition of a function.
By a straightforward adaptation of DLAL subject reduction we have:

Theorem 1 (Subject Reduction). *Let* $\mathtt{p}$ *be a LPL program such that* $\Gamma; \Delta \vdash \mathtt{p} : A$*. Then,* $\mathtt{p} \to_{\mathtt{L}}^* \mathtt{q}$ *implies* $\Gamma; \Delta \vdash \mathtt{q} : A$*.*

2.5 Some Examples

We give here some hints on how to program in LPL. More information about the typing can be found in Section 3.1. Addition can be defined by a standard definition $\mathtt{d}_\mathtt{A}$ as:

$$\mathtt{Add}\ (\mathtt{s}\ \mathtt{X})\ \mathtt{Y} = \mathtt{s}\ (\mathtt{Add}\ \mathtt{X}\ \mathtt{Y})\ ,\ \mathtt{Add}\ \mathtt{0}\ \mathtt{Y} = \mathtt{Y}$$

the first is a matching arguments, so $\mathtt{d}_\mathtt{A}$ is typable for example by taking $\mathtt{Add} : \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}$. Multiplication can be given by a function definition $\mathtt{d}_\mathtt{M}$ as:

$$\mathtt{Mul}\ (\mathtt{s}\ \mathtt{X})\ \mathtt{Y} = \mathtt{Add}\ \mathtt{Y}\ (\mathtt{Mul}\ \mathtt{X}\ \mathtt{Y})\ ,\ \mathtt{Mul}\ \mathtt{0}\ \mathtt{Y} = \mathtt{0}$$

the first is a matching argument and since $\mathtt{Add} : \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}$ we can type $\mathtt{d}_\mathtt{M}$ using

$\mathtt{Mul} : \mathbb{N} \multimap !\mathbb{N} \multimap \S\S\mathbb{N}$ and by means of rule $(\S I)$ and $(\S E)$. We have a type coercion program for every data type, e.g on numerals we have $\mathtt{d_C}$ as:

$$\mathtt{Coer}\ (\mathtt{s}\ \mathtt{X}) = \mathtt{s}\ (\mathtt{Coer}\ \mathtt{X})\ ,\ \mathtt{Coer}\ 0 = 0$$

typable with $\mathtt{Coer} : \mathbb{N} \multimap \S\mathbb{N}$. This can be used in $\mathtt{d_P}$ to define the usual Map program:

$$\mathtt{Map}\ \mathtt{Y}\ (\mathtt{X} : \mathtt{XS}) = (\mathtt{Y}\ (\mathtt{Coer}\ \mathtt{X})) : (\mathtt{Map}\ \mathtt{Y}\ \mathtt{XS})\ ,\ \mathtt{Map}\ \mathtt{Y}\ \mathtt{nil} = \mathtt{nil}$$

typable with $\mathtt{Map} :!(\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{L} \multimap \S\mathbb{L}$. Note that we have also a typing for non linear function argument of Map. Using this we can write a program Map $(\times 2)\ (1 : 2 : 3 : 4)$ that doubles all the elements of $(1 : 2 : 3 : 4)$ as

$$\mathtt{LetRec}\ \mathtt{d_A}, \mathtt{d_M}, \mathtt{d_C}, \mathtt{d_P}\ \mathtt{in}\ \mathtt{Map}\ (\lambda \mathtt{x}.\mathtt{Mul}\ \mathtt{x}\ 2)\ (1 : 2 : 3 : 4)$$

typable with type $\S\S\S\mathbb{L}$ using $\mathtt{Map} :!(\mathbb{N} \multimap \S\S\mathbb{N}) \multimap \mathbb{L} \multimap \S\S\S\mathbb{L}$. Using again the coercions we have a definition $\mathtt{d_R}$ for the Foldr program:

$$\mathtt{Foldr}\ \mathtt{Y}\ \mathtt{Z}\ (\mathtt{X} : \mathtt{XS}) = \mathtt{Y}\ (\mathtt{Coer}\ \mathtt{X})\ (\mathtt{Foldr}\ \mathtt{Y}\ \mathtt{Z}\ \mathtt{XS})\ ,\ \mathtt{Foldr}\ \mathtt{Y}\ \mathtt{Z}\ \mathtt{nil} = \mathtt{Z}$$

typable with $\mathtt{Foldr} :!(\mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}) \multimap \S\mathbb{N} \multimap \mathbb{L} \multimap \S\mathbb{N}$. Note that we have also a typing $\mathtt{Foldr} :!(\mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}) \multimap \S\S\mathbb{N} \multimap \mathbb{L} \multimap \S\S\mathbb{N}$, we can use it to write the program Foldr $(+)\ 0\ (1 : 2 : 3)$ that sums the values in the list $(1 : 2 : 3)$. We have

$$\mathtt{LetRec}\ \mathtt{d_A}, \mathtt{d_C}, \mathtt{d_R}\ \ \mathtt{in}\ \mathtt{Foldr}\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{Add}\ \mathtt{x}\ \mathtt{y})\ 0\ (1 : 2 : 3)$$

typable with type $\S\S\mathbb{N}$. Finally have also some interesting programs over trees. For example we have $\mathtt{d_T}$ defining addition $\mathtt{Tadd} : \mathbb{T} \multimap \mathbb{T} \multimap \S\mathbb{T}$ as:

$$\mathtt{Tadd}\ (\mathtt{node}\ \mathtt{X}\ \mathtt{Y}\ \mathtt{Z})\ (\mathtt{node}\ \mathtt{X}'\ \mathtt{Y}'\ \mathtt{Z}') = \mathtt{node}\ (\mathtt{Add}\ \mathtt{X}\ \mathtt{X}')\ (\mathtt{Tadd}\ \mathtt{Y}\ \mathtt{Y}')\ (\mathtt{Tadd}\ \mathtt{Z}\ \mathtt{Z}')\ ,$$
$$\mathtt{Tadd}\ \epsilon\ \mathtt{X} = \mathtt{Coer}_{\mathbb{T}}\ \mathtt{X}\ ,\ \mathtt{Tadd}\ \mathtt{X}\ \epsilon = \mathtt{Coer}_{\mathbb{T}}\ \mathtt{X}$$

where $\mathtt{Coer}_{\mathbb{T}}$ is a coercion for the $\mathbb{T}$ data type (defined analogously to the one for natural numbers). It should be stressed that even though in LLL one can define a type for binary trees (as in system F) there is no simple way in this system to program Tadd. Moreover, we here can program some more examples that would be awkward to program in LLL. For example division by 2 on unary integers.

$$\mathtt{Div}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{X})) = \mathtt{s}\ (\mathtt{Div}\ \mathtt{X}),\ \mathtt{Div}\ (\mathtt{s}\ 0) = 0\ ,\ \mathtt{Div}\ 0 = 0\ ,$$

that gets type $\mathbb{N} \multimap \S\mathbb{N}$. The problem for typing DIV in LLL is that this kind of recursion scheme (using the pattern $\mathtt{s}(\mathtt{s}\ \mathtt{X})$) cannot be implemented directly on Church integers, by using their iteration scheme. Similarly functions that are defined by pattern matching over two arguments, like for example Tadd above and the minimum function:

$$\mathtt{Min}\ (\mathtt{s}\ \mathtt{X})\ (\mathtt{s}\ \mathtt{Y}) = \mathtt{s}\ (\mathtt{Min}\ \mathtt{X}\ \mathtt{Y}),\ \mathtt{Min}\ (\mathtt{s}\ \mathtt{X})\ 0 = 0\ ,\ \mathtt{Min}\ 0\ (\mathtt{s}\ \mathtt{Y}) = 0\ ,$$

that is typable as $\mathtt{Min} : \mathbb{N} \multimap \mathbb{N} \multimap \S\mathbb{N}$ cannot be programmed naturally in LLL.

3 Translating LPL in eLPL

The proof of the polynomial time complexity bound for light linear logic [4] and light λ-calculus [14] uses a notion of *stratification* of the proofs or λ-terms by *depths*. To adapt this methodology to LPL we need to make the stratification explicit in the programs. For that we introduce an intermediate language called eLPL, adapted from light λ-calculus [14], and where the stratification is managed by new constructions (corresponding to the modality rules). Note that the user is not expected to program directly in eLPL, but instead he will write typed LPL programs, which will then be *compiled* in eLPL. The polynomial bound on execution will then be proved for a certain strategy of evaluation

of eLPL programs.
The syntax of eLPL is given in Table 3. An eLPL term $\lambda\mathtt{x}.\mathtt{let\ x\ be\ !y\ in\ M[y/x]}$, where $\mathtt{y}$ is fresh, is abbreviated by $\lambda^{!}\mathtt{x.M}$. Moreover, we write $\mathtt{let\ M\ be\ }\dagger^{\mathtt{n}}\mathtt{x\ in\ N}$ to denote terms as $\mathtt{let\ M\ be\ }\dagger\mathtt{x_1\ in\ (let\ x_1\ be\ }\dagger\mathtt{x_2\ in\ }(\dots(\mathtt{let\ x_{n-1}\ be\ }\dagger\mathtt{x_n\ in\ N})\cdots)$.
We will give a translation of type derivations of LPL programs to type derivations of eLPL programs, which will leave almost unchanged the typing part, and act only on the term part of LPL programs.
The contexts of typing judgments for eLPL terms and programs can contain a new kind of type declaration, denoted $\mathtt{x} : [A]_{\S}$, where A is a type, which corresponds to a kind of intermediary status for variables with type $\S A$. In particular, $[A]_{\S}$ does not belong to the type grammar and these variables cannot be λ-abstracted, the only possibility is to bind them by means of a $\mathtt{let}$. This kind of declarations is made necessary by the fact that eLPL is handling explicitly stratification. If $\Delta = \mathtt{x}_1 : A_1, \dots, \mathtt{x}_n : A_n$ then $[\Delta]_{\S}$ is $\mathtt{x}_1 : [A_1]_{\S}, \dots, \mathtt{x}_n : [A_n]_{\S}$. The typing rules are given in Table 4. Note that declarations $\mathtt{x} : [A]_{\S}$ are introduced by (§ I) rules, and eliminated by (§ E) rules. Intuitively, a variable $\mathtt{x} : [A]_{\S}$ is a kind of special pattern for $\S\mathtt{x}$, and only a term of the shape $\S\mathtt{M}$ will be able to trigger the reduction of the $\mathtt{let}$. Observe that if $\lambda\mathtt{x.M}$ is a well typed eLPL term, then $n_o(\mathtt{x}, \mathtt{M}) \leq 1$.
Note that all the rules in Table 4, but the rule $(l!)$, are the same rules as in Table 2 but for

program definition	$\mathtt{p} ::= \mathtt{M} \mid \mathtt{LetRec\ d_F\ in\ p}$
value definition	$\mathtt{v} ::= \mathtt{c}(\mathtt{v_1}, \cdots, \mathtt{v_n})$
pattern definition	$\mathtt{t} ::= \mathtt{X} \mid \mathtt{c}(\mathtt{t_1}, \dots, \mathtt{t_n})$
function definition	$\mathtt{d_F} ::= \mathtt{F}(\mathtt{t_1}, \dots, \mathtt{t_n}) = \mathtt{N} \mid \mathtt{d_F}, \mathtt{d_F}$
term definition	$\mathtt{M}, \mathtt{N} ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{X} \mid \mathtt{F} \mid \lambda\mathtt{x.M} \mid \mathtt{MM} \mid \mathtt{!M} \mid \S\mathtt{M} \mid \mathtt{let\ M\ be\ !x\ in\ M} \mid \mathtt{let\ M\ be\ }\S\mathtt{x\ in\ M}$

Table 3. eLPL term language definition

the terms being the subjects of each rule and for the distinction between $\S A$ and $[A]_{\S}$. This suggests that we can give a translation on type derivation inducing a translation on typable terms. From this observation we have the following:

Definition 6. *Let* $\mathtt{M}$ *be an* LPL *term and* Π *be a type derivation proving* $\Gamma; \Delta \vdash \mathtt{M} : B$. *Then,* Π^* *is the type derivation in* eLPL *proving* $\Gamma; \Delta \vdash \mathtt{M}^* : B$ *obtained by:*

- *substituting to each rule* (X) *of* LPL *in* Π *the corresponding rule* (X) *in* eLPL *and changing accordingly the subject,*
- *adding at the end: for each variable* $\varkappa \in \Gamma$ *(resp.* $\mathtt{x} : [A]_{\S} \in \Delta$*) an occurrence of the rule* $(l!)$ *(resp. (§ E) with a l.h.s. premise of the form* $; \mathtt{y} : \S A \vdash \mathtt{y} : \S A$*).*

constructors and functions rules, $(Ax), (W), (C), (\multimap\ I), (\multimap\ E), (D), (R)$: as in Table 2

$$\frac{; \Gamma, \Delta \vdash \mathtt{M} : A}{\Gamma; [\Delta]_{\S} \vdash \S\mathtt{M} : \S A}\ (\S I) \qquad \frac{\Gamma, \mathtt{x} : A; \Delta \vdash \mathtt{M} : B}{\Gamma; \Delta \vdash \lambda^{!}\mathtt{x.M} : !A \multimap B}\ (\Rightarrow I) \qquad \frac{\Gamma, \varkappa : A; \Delta \vdash \mathtt{M} : B \quad \mathtt{x}\ \text{fresh}}{\Gamma, \varkappa : A; \Delta \vdash \mathtt{let}\ \varkappa\ \mathtt{be\ !x\ in\ M[x/}\varkappa\mathtt{]} : B}\ (l!)$$

$$\frac{\Gamma_1; \Delta_1 \vdash \mathtt{N} : \S A \quad \Gamma_2; \mathtt{x} : [A]_{\S}, \Delta_2 \vdash \mathtt{M} : B}{\Gamma_1, \Gamma_2, \Delta_1; \Delta_2, \vdash \mathtt{let\ N\ be\ }\S\mathtt{x\ in\ M} : B}\ (\S E) \qquad \frac{\Gamma_1; \Delta \vdash \mathtt{M} : (!A) \multimap B \ ; \Gamma_2 \vdash \mathtt{N} : A \ \ \Gamma_2 \subseteq \{\varkappa : C\}}{\Gamma_1, \Gamma_2; \Delta \vdash \mathtt{M!N} : B}\ (\Rightarrow E)$$

Table 4. eLPL type system

The above translation leaves the contexts Γ and Δ and the type B, the same as in the source derivation. The translation can be easily extended to function definitions:

Definition 7. *Let* $\mathtt{F}(\overrightarrow{\mathtt{t}_i}) = \mathtt{N_i}, \mathtt{d_F}$ *be an* LPL *function definition and* Π *be its type derivation in* LPL *ending as:*

$$\frac{\Sigma_1 : \Gamma; \Delta \vdash \mathtt{F}(\overrightarrow{\mathtt{t}_i}) : B \quad \Sigma_2 : \Gamma; \Delta \vdash \mathtt{N}_i : B \quad \Sigma_3 : \rhd \mathtt{d_F} : B}{\rhd \mathtt{F}(\overrightarrow{\mathtt{t}_i}) = \mathtt{N}_i, \mathtt{d_F} : B} \ (D)$$

Then, Π^* *is the type derivation in* eLPL *ending as:*

$$\frac{\Sigma_1 : \Gamma; \Delta \vdash \mathtt{F}(\overrightarrow{\mathtt{t}_i}) : B \quad \Sigma_2^* : \Gamma; \Delta \vdash \mathtt{N}_i^* : B \quad \Sigma_3^* : \rhd \mathtt{d_F}^* : B}{\rhd \mathtt{F}(\overrightarrow{\mathtt{t}_i}) = \mathtt{N}_i^*, \mathtt{d_F}^* : B} \ (D)$$

Note that in the translation we do not translate the left hand-side of a definition case, we keep it to be exactly the same as in LPL. We can now extend the translation to programs.

Definition 8. *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M}$ *be an* LPL *program and let* Π *be a type derivation in* LPL *proving* $\Gamma; \Delta \vdash \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M} : B$*. Then,* Π^* *is the derivation in* eLPL *proving* $\Gamma; \Delta \vdash \mathtt{LetRec}\ \mathtt{d_{F_1}}^*, \ldots, \mathtt{d_{F_n}}^*\ \mathtt{in}\ \mathtt{M}^* : B$ *obtained by replacing every derivation* $\Sigma_i : \rhd \mathtt{d_{F_i}} : \S B$ *in* Π *by the derivation* $\Sigma_i^* \ \rhd \mathtt{d_{F_i}}^* : \S B$ *and by replacing the derivation* $\Sigma : \Gamma; \Delta \vdash \mathtt{M} : B$ *by the derivation* $\Sigma^* : \Gamma; \Delta \vdash \mathtt{M}^* : B$*.*

The above translation is not exactly syntax directed; the reason is that we want the following remarkable property:

Lemma 2. *Let* $\mathtt{M}$ *be an* LPL *term and* Π *be a type derivation for it. Then the term* $\mathtt{M}^*$ *obtained by the derivation* Π^* *is such that* $n_o(\varkappa, \mathtt{M}^*) \leq 1$ *for each* $\varkappa \in \mathrm{FV}(\mathtt{M}^*)$*.*

Because of the new `let` constructions, the reduction rules are extended as follows:

Definition 9. *The reduction relation* $\rightarrow_{\mathtt{I}}$ *is the contextual closure of the relations* $\rightarrow_{\mathtt{Rec}}$*,* $\rightarrow_\gamma$ *(as in Def. 4) and of the reductions* $\rightarrow_\beta$*,* $\rightarrow_!$*,* $\rightarrow_\S$*,* $\rightarrow_{\mathtt{com_1}}$*,* $\rightarrow_{\mathtt{com_2}}$ *and* $\rightarrow_{\mathtt{com_3}}$ *for* $\dagger \in \{!, \S\}$ *defined as:*

$$(\lambda \mathtt{x.M})\mathtt{N} \rightarrow_\beta \mathtt{M}[\mathtt{N}/\mathtt{x}], \quad \mathtt{let}\ !\mathtt{N}\ \mathtt{be}\ !\mathtt{x}\ \mathtt{in}\ \mathtt{M} \rightarrow_! \mathtt{M}[\mathtt{N}/\mathtt{x}], \quad \mathtt{let}\ \S\mathtt{N}\ \mathtt{be}\ \S\mathtt{x}\ \mathtt{in}\ \mathtt{M} \rightarrow_\S \mathtt{M}[\mathtt{N}/\mathtt{x}],$$

$$\mathtt{M}(\mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{V}) \rightarrow_{\mathtt{com_1}} \mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ (\mathtt{MV}),$$

$$(\mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{V})\mathtt{M} \rightarrow_{\mathtt{com_2}} \mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ (\mathtt{VM}),$$

$$\mathtt{let}\ (\mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{V})\ \mathtt{be}\ \dagger\mathtt{y}\ \mathtt{in}\ \mathtt{W} \rightarrow_{\mathtt{com_3}} \mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ (\mathtt{let}\ \mathtt{V}\ \mathtt{be}\ \dagger\mathtt{y}\ \mathtt{in}\ \mathtt{W}).$$

As usual $\rightarrow_{\mathtt{I}}^*$ *denotes the reflexive and transitive closure of* $\rightarrow_{\mathtt{I}}$*.*

We write $\rightarrow_{com}$ for any one of the three *commutation reductions* $\rightarrow_{com_i}$. Note that:

- in $\rightarrow_\beta$ and $\rightarrow_\S$ *at most one occurrence* of $\mathtt{x}$ is substituted in $\mathtt{M}$ (linear substitution),
- the reduction $\rightarrow_!$, $\rightarrow_{\mathtt{Rec}}$, $\rightarrow_\gamma$ are the only ones inducing non-linear substitutions.

In fact, a β-step in LPL corresponds in eLPL to a (linear) β step followed by a ! step. Now, to reason about the stratification we define the notion of *depth*.

Definition 10. *Let* $\mathtt{M}$ *be an eLPL term and* $\mathtt{N}$ *be an occurrence of a subterm in it. The depth of* $\mathtt{N}$ *in* $\mathtt{M}$*, denoted* $\mathtt{d}(\mathtt{N},\mathtt{M})$ *is the number of* $\S$ *or* ! *symbols encountered in the syntax tree of* $\mathtt{M}$ *when going from the root of* $\mathtt{M}$ *to the root of* N*. The* degree *of an eLPL term* $\mathtt{M}$*, denoted by* $\mathtt{d}(\mathtt{M})$*, is the maximal depth of any subterm in it.*

E.g. Take $\mathtt{M}$ as $\mathtt{N}!(\mathtt{let\ y\ be\ !x\ in}\S(\mathtt{F\ x}))$. Then $\mathtt{d}(\mathtt{N},\mathtt{M}) = 0, \mathtt{d}(\mathtt{y},\mathtt{M}) = 1$ and $\mathtt{d}(\mathtt{x},\mathtt{M}) = 2$. In what follows we write $\mathtt{N} \in_i \mathtt{M}$ to denote the fact that $\mathtt{N}$ is a subterm of $\mathtt{M}$ at depth i, i.e. $\mathtt{d}(\mathtt{N},\mathtt{M}) = i$. We write $n_o^i(\varkappa,\mathtt{M})$ (respectively $|\mathtt{M}|_i$, $\mathrm{FV}(\mathtt{M})^i$ and $\mathrm{FO}(\mathtt{M})^i$) to denote the restriction of $n_o(\varkappa,\mathtt{M})$ (respectively $|\mathtt{M}|$, $\mathrm{FV}(\mathtt{M})$ and $\mathrm{FO}(\mathtt{M})$) at depth i.
Now we can state some important properties of typing on eLPL terms.

Lemma 3 (Variable occurrences). *Let $\Gamma;\Delta \vdash \mathtt{M} : A$. Then:*

i) if $\varkappa \in \mathrm{dom}(\Delta)$ then $n_o(\varkappa,\mathtt{M}) \leq 1$.
ii) if $n_o(\varkappa,\mathtt{M}) > 1$ then $\varkappa \in \mathrm{dom}(\Gamma)$ and $\mathtt{d}(\varkappa,\mathtt{M}) = 1$.
iii) if $\varkappa \in \mathrm{dom}(\Gamma \cup \Delta)$ we have $n_o^0(\varkappa,\mathtt{M}) \leq 1$.

Lemma 4. *Let $\mathtt{F}(\mathtt{t_1},\ldots,\mathtt{t_n}) = \mathtt{N}$ and let $\mathtt{F}(\mathtt{t}_1^1,\ldots,\mathtt{t}_n^1),\ldots,\mathtt{F}(\mathtt{t}_1^m,\ldots,\mathtt{t}_n^m)$ be the recursive calls of $\mathtt{F}$ in $\mathtt{N}$. Then, $\mathtt{d}(\mathtt{F}(\mathtt{t}_1^i,\ldots,\mathtt{t}_n^i),\mathtt{N}) = 0$.*

These properties will be useful when studying the bounds on the reductions in eLPL.

3.1 Revisiting the examples

We now come back to the examples of Section 2.5 in order to clarify the way such programs can be typed by giving the translations in eLPL. The function definitions $\mathtt{d_C},\mathtt{d_A}$ and $\mathtt{d_M}$ for the programs $\mathtt{Coer} : \mathbb{N} \multimap \S\mathbb{N}$, $\mathtt{Add} : \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}$ and $\mathtt{Mul} : \mathbb{N} \multimap !\mathbb{N} \multimap \S\S\mathbb{N}$ respectively, can be translated in eLPL as

$\mathtt{Coer\ (s\ X) = let\ (Coer\ X)\ be}\ \S\mathtt{z\ in}\ \S(\mathtt{sz})\ ,\ \mathtt{Coer\ 0} = \S\mathtt{0}$
$\mathtt{Add\ (s\ X)\ Y = let\ (Add\ X\ Y)\ be}\ \S\mathtt{z\ in}\ \S(\mathtt{sz})\ ,\ \mathtt{Add\ 0\ Y = Y}$
$\mathtt{Mul\ (s\ X)\ Y = let\ Y\ be\ !r\ in\ let\ (Mul\ X\ !r)\ be}\ \S\mathtt{z\ in}\ \S(\mathtt{Add\ r\ z})\ ,\ \mathtt{Mul\ 0\ Y} = \S\S\mathtt{0}$

Similarly, the definition $\mathtt{d_P}$ for $\mathtt{Map} :!(\mathbb{N} \multimap \S\S\mathbb{N}) \multimap \mathbb{L} \multimap \S\S\S\mathbb{L}$ can be translated as:

$\mathtt{Map\ Y\ (X : XS) = let\ Y\ be\ !y\ in\ let\ Map\ !y\ XS\ be}\ \S\S\S\mathtt{z\ in\ let\ Coer\ X\ be}\ \S\mathtt{x\ in}$
$\S(\mathtt{let\ y\ x\ be}\ \S\S\mathtt{r\ in\ r : z})\ ,\ \mathtt{Map\ Y\ nil} = \S\S\S\mathtt{nil}$

Then the program $\mathtt{LetRec}\ \mathtt{d_A},\mathtt{d_M},\mathtt{d_C},\mathtt{d_P}\ \mathtt{in\ Map}\ (\lambda \mathtt{x.Mul\ x\ 2})\ (\mathtt{1 : 2 : 3 : 4})$ can be translated in eLPL as $\mathtt{LetRec}\ \mathtt{d_A^*},\mathtt{d_M^*},\mathtt{d_C^*},\mathtt{d_P^*}\ \mathtt{in\ Map}\ !(\lambda \mathtt{x.Mul\ x\ !2})\ (\mathtt{1 : 2 : 3 : 4})$. Analogously, the definition $\mathtt{d_R}$ for $\mathtt{Foldr} :!(\mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}) \multimap \S\S\mathbb{N} \multimap \mathbb{L} \multimap \S\S\mathbb{N}$ can be translated as:

$\mathtt{Foldr\ Y\ Z\ (X : XS) = let\ Y\ be\ !y\ in\ let\ Coer\ X\ be}\ \S\mathtt{x\ in\ let\ Foldr\ !y\ Z\ XS}$
$\mathtt{be}\ \S\mathtt{r\ in}\ \S(\mathtt{yxr})\ ,\ \mathtt{Foldr\ Y\ Z\ nil = Z}$

Then the program $\mathtt{LetRec}\ \mathtt{d_A},\mathtt{d_C},\mathtt{d_R}\ \ \mathtt{in\ Foldr}\ (\lambda \mathtt{x}.\lambda \mathtt{y.Add\ x\ y})\ \mathtt{0}\ (\mathtt{1 : 2 : 3})$ can be translated as $\mathtt{LetRec}\ \mathtt{d_A^*},\mathtt{d_C^*},\mathtt{d_R^*}\ \ \mathtt{in\ Foldr}\ !(\lambda \mathtt{x}.\lambda \mathtt{y.Add\ x\ y})\ \S\S\mathtt{0}\ (\mathtt{1 : 2 : 3})$.
Finally the definition $\mathtt{d_T}$ for $\mathtt{Tadd} : \mathbb{T} \multimap \mathbb{T} \multimap \S\mathbb{T}$ can be translated using a coercion $\mathtt{Coer}_{\mathbb{T}}$ for the $\mathbb{T}$ data type (defined analogously to the one for natural numbers) as:

$\mathtt{Tadd\ (node\ X\ Y\ Z)\ (node\ X'\ Y'\ Z') = let\ Add\ X\ X'\ be}\ \S\mathtt{x\ in\ let\ Tadd\ Y\ Y'\ be}\ \S\mathtt{y\ in}$
$\mathtt{let\ Tadd\ Z\ Z'\ be}\ \S\mathtt{z\ in}\ \S(\mathtt{node\ x\ y\ z})\ ,\ \mathtt{Tadd\ X}\ \epsilon = \mathtt{Coer}_{\mathbb{T}}\ \mathtt{X}\ ,\ \mathtt{Tadd}\ \epsilon\ \mathtt{X} = \mathtt{Coer}_{\mathbb{T}}\ \mathtt{X}$

4 PTIME completeness

The proof that LPL is complete for polynomial time functions is rather standard: we can simulate any polynomial time (one tape) Turing machine in the language. In LPL we can represent all the polynomials in $\mathbb{N}[X]$, but here it is sufficient to use:

Lemma 5. *For any $K, k \in \mathbf{N}$, there exists an integer l and an LPL program of type $\mathbb{N} \multimap \S^l\mathbb{N}$ representing the polynomial $K \times x^{2^k}$.*

We consider a polytime Turing machine $\mathcal{M}$ with witness time polynomial P, n states, and a 3 symbol alphabet (0,1 and blank). We encode the configurations with the type $\mathtt{config} = (\mathbb{L}_3 \times \mathbb{L}_3) \times \mathbb{B}_n$ where the first $\mathbb{L}_3$ type corresponds to the left part of the tape, the second one corresponds to the right part, starting from the scanned symbol.

Lemma 6. *For any transition function δ, there exists an* LPL *basic function* $\underline{\mathtt{conf2conf}}$: $\mathtt{config} \multimap \mathtt{config}$ *representing the corresponding action on configurations.*

We easily check that $\underline{\mathtt{conf2conf}}$ can be defined by a case analysis, using the pattern-matching, and does not need recursion; so it is a basic function, hence its type. We will also use an iterator of type $\mathbb{N} \multimap !(A \multimap A) \multimap \S A \multimap \S A$ with $A = \mathtt{config}$ defined by:

$\mathtt{Iter}\ (\mathtt{s}\ \mathtt{X})\ \mathtt{f}\ \mathtt{base} = \mathtt{f}\ \mathtt{Iter}\ \mathtt{X}\ \mathtt{f}\ \mathtt{base}\ ,\ \mathtt{Iter}\ 0\ \mathtt{f}\ \mathtt{base} = \mathtt{base}$

Theorem 2 (Ptime Completeness). *For any polynomial time function f on $\{0,1\}^\star$, there exists an integer j and an* LPL *program of type $\mathbb{L}_2 \multimap \S^j\mathbb{L}_2$, representing f.*

Proof. We simulate the machine $\mathcal{M}$ computing f. By Lemma 5 we represent in LPL the polynomial P, with a term t of type $\mathbb{N} \multimap \S^m\mathbb{N}$, for some m. It is also easy to define a $\mathtt{Length} : \mathbb{L}_2 \multimap \S\mathbb{N}$ and a $\mathtt{Init} : \mathbb{L}_2 \multimap \mathtt{config}$ which maps a word to the corresponding initial configuration. The simulation is then obtained by iterating $\underline{\mathtt{conf2conf}}$ for $(\mathtt{t}\ (\mathtt{Length}\ \mathtt{w}))$ steps, starting from the base $(\mathtt{Init}\ \mathtt{w})$, and then extracting the result by using projection maps. This can be suitably typed, using some coercions on $\mathbb{L}_2$. □

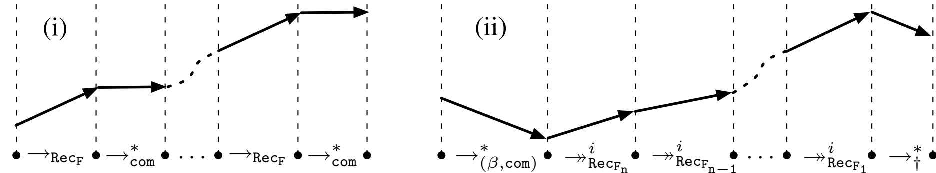

Fig. 1. Term size variations at fixed depth i by a standard reduction step and round at same depth.

5 Polynomial Time soundness

We here show that well-typed LPL programs satisfying the syntactic criterion can be evaluated in polynomial time in the size of the input (with the degree of the polynomial given by the type derivation). For that we work on the eLPL translated programs. For simplicity we do not consider basic functions, but the proof can be easily extended to the whole LPL. From now on we only consider eLPL programs obtained by translation from well-typed LPL programs satisfying the syntactic criterion.
Similarly to the polynomial soundness proof for LLL, we prove that the evaluation of eLPL programs can be done in polynomial time using a specific depth-by-depth stratified strategy. The polynomial bound for this strategy in LLL relies on:

1. reducing a redex at depth i does not affect the size at depth $j < i$
2. a reduction at depth i strictly decreases the size at depth i
3. a reduction at depth i increases the size at depth $j > i$ at most quadratically

4. the reduction does not increase the degree of a term

Unfortunately for eLPL facts 2, 3 and 4 above do not hold due to the presence of LetRec, hence some adaptations are needed.

In order to adapt these facts we need to impose a rigid structure on the reductions at a fixed depth. We consider a notion of *standard reduction round* at a fixed depth i, denoted $\Rightarrow^i$ and a notion of *standard reduction step* at a fixed depth i denoted $\twoheadrightarrow^i_{\mathtt{Rec_F}}$ for each function symbol F of the program. A *standard reduction step* $\twoheadrightarrow^i_{\mathtt{Rec_F}}$ is an alternating maximal sequence of $\rightarrow_{\mathtt{Rec_F}}$ and $\rightarrow^*_{\mathtt{com}}$ steps at depth i as represented in Figure 1.(i). It is maximal in the sense that in the step $\rightarrow^*_{\mathtt{com}}$ all the possible commutations are done. Note that, during a standard reduction step the size of the term at depth i might grow as depicted in Figure 1.(i), i.e. $\rightarrow^*_{\mathtt{com}}$ steps leave the size unchanged while $\rightarrow_{\mathtt{Rec_F}}$ steps can grow the size at depth i. We introduce some new measures on matching arguments to show that this growth is polynomial in the size of the initial term i.e. Lemma 20.

A *standard reduction round* $\Rightarrow^i$ is a sequence of maximal reduction steps as represented in Figure 1.(ii). Every reduction step is maximal in the sense that it reduces all the possible redexes of the intended kind. Note that, also during a standard reduction round the size of the term at depth i might grow as depicted in Figure 1.(ii), i.e. $\rightarrow^*_{\beta,\mathtt{com}}$ and $\rightarrow^*_{\dagger}$ steps make the size decrease while $\twoheadrightarrow^i_{\mathtt{Rec_{F_j}}}$ steps can make the size grow as discussed above. So, by using the bound on a standard reduction step and by the fact that the number of standard reduction steps depends on the shape of the program, we adapt fact 2 above by showing that this growth is polynomial in the size of the initial term, i.e. Theorem 3. Moreover, by similar arguments we adapt fact 3 above by showing that a standard reduction round at depth i can increase the size at depth $j > i$ at most polynomially, Lemma 21.

Finally, in order to adapt fact 4 to our framework, we introduce the notion of *potential degree*. This is the maximal degree a term can have during the reduction and it can be statically determinated. We show that a *standard reduction*, i.e. a sequence of standard reduction rounds of increasing depth, does not increase the potential degree, Lemma 23. Summarizing, what we obtain can be reformulated for eLPL as:

1. reducing a redex at depth i does not affect depth $j < i$
2. a *standard reduction round* at depth i strictly decreases *some measures on matching arguments* and increases the size at depth i at most *polynomially*
3. a *standard reduction round* at depth i increases the size at depth $j > i$ at most *polynomially*
4. the *standard reduction* does not increase the *potential degree* of a term

Now, from these new key facts, the polynomial soundness, Theorem 4, will follow.

5.1 Preliminary properties

For a given a program $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M}$ it is convenient to introduce the following static constants:

$$K_{\mathtt{F_i}} = \max\{|\mathtt{N}|_j \mid \mathtt{F_i}(\mathtt{t_1}, \ldots, \mathtt{t_n}) = \mathtt{N} \in \mathtt{d_{F_i}}\} \quad \text{and} \quad K = \max\{K_{\mathtt{F_i}} \mid 1 \leq i \leq n\}$$

We now show some simple properties about eLPL term depths. Recall that eLPL has been designed in such a way to preserve the good LLL properties. Indeed, the following lemmas can be directly adapted from the arguments in [14].

Lemma 7. *1. Let $\lambda\mathtt{x}.\mathtt{M}$ be a well typed term. Then $n_o(\mathtt{x},\mathtt{M}) \leq 1$ and $\mathtt{d}(\mathtt{x},\mathtt{M}) = 0$.*

2. Let $\mathtt{let}\ \mathtt{M}\ \mathtt{be}\dagger\mathtt{x}\ \mathtt{in}\ \mathtt{N}$ be a well typed term. Then $\mathtt{x} \in \mathrm{FV}(\mathtt{N})$ implies that for each occurrence $\mathtt{x}_i$ of $\mathtt{x}$ in $\mathtt{N}$, $\mathtt{d}(\mathtt{x}_i,\mathtt{N}) = 1$.

3. Let $\mathtt{F}(\mathtt{t_1},\dots,\mathtt{t_n}) = \mathtt{N}$ be a typed definition case: if $\mathtt{X} \in \mathrm{FV}(\mathtt{t}_i)$, then $\mathtt{d}(\mathtt{X},\mathtt{N}) = 0$.

Lemma 8. *1. Let $\mathtt{M}$ be $\mathtt{com}$-normal at depth $i \geq 0$. If $\mathtt{M} \to^*_\dagger \mathtt{M}'$ at depth i then for $j > i$ we have $|\mathtt{M}'|_j \leq |\mathtt{M}|_j \times |\mathtt{M}|_i$ and it does not create redexes at depth $k \leq i$.*

2. At depth $i \geq 0$, $\to_\beta$ and $\to_\dagger$ reductions strictly decrease the term size at depth i. The number of $\mathtt{com}$-reductions in $\mathtt{M} \to^*_{\mathtt{com}} \mathtt{M}'$ is bounded by $(|\mathtt{M}|_i)^2$. The number of $(\beta,\dagger)$-reductions in $\mathtt{M} \to^*_{\beta,\dagger} \mathtt{M}'$ is bounded by $|\mathtt{M}|_i$.

5.2 Bound the number of steps at a fixed depth

We need to define a measure, denoted $\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M})$, that will be used to bound the number of $\mathtt{rec}$-reduction steps at depth j. For that we will first introduce an intermediary notion:

Definition 11 (External constructor size). *The* external constructor size *of a term $\mathtt{M}$ at depth j, denoted $\|\mathtt{M}\|_j$, is the number of constructors of $\mathtt{M}$ at depth j which are not in an argument of a function, of a $\mathtt{let}$ or of a variable at depth j.*

The external constructor size measure enjoys the following remarkable property.

Lemma 9. *Let $\mathtt{F}(\mathtt{t_1},\dots,\mathtt{t_n}) = \mathtt{N}$ be safe and let the recursive calls of $\mathtt{F}$ in $\mathtt{N}$ be $\mathtt{F}(\mathtt{t}^1_1,..,\mathtt{t}^1_n),\dots,\mathtt{F}(\mathtt{t}^m_1,..,\mathtt{t}^m_n)$. Then $\sum_{r\in\mathcal{R}(\mathtt{F})} \|\mathtt{t}_r\|_0 > \sum_{k=1}^m \sum_{r\in\mathcal{R}(\mathtt{F})} \|\mathtt{t}^k_r\|_0$.*

Proof. By induction on m by using Lemma 4 and Definition 1. □

Lemma 10. *1. If $\Gamma;\Delta \vdash \mathtt{M} : \dagger A$ and $\mathtt{M}$ is $(\beta,\mathtt{com})$-normal at depth 0, then $\|\mathtt{M}\|_0 = 0$.*

2. If $\Gamma;\Delta \vdash \mathtt{M} : A$ and $\mathtt{M}$ is $(\beta,\mathtt{com})$-normal at depth 0 and $\mathtt{M} \to_{\mathtt{Rec_F}} \mathtt{M}'$ at depth 0, then $\|\mathtt{M}'\|_0 = \|\mathtt{M}\|_0$.

Proof. 1. By induction on $\mathtt{M}$. 2. By induction on $\mathtt{M}$ using point 1. □

Note that the above lemma applies on each typable term. This means that for each $(\beta,\mathtt{com})$-normal term $\mathtt{M}$ the measure $\|\mathtt{M}\|_0$ is invariant under $\mathtt{Rec_F}$ function calls.

Definition 12. *We call $\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M})$ the sum of the external constructor sizes of the matching arguments at depth j of the function $\mathtt{F}$ in $\mathtt{M}$. It is inductively defined as:*

$$\mathrm{SA}^{\mathtt{F}}_0(\dagger\mathtt{M}') = 0 \quad \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}+1}(\dagger\mathtt{M}') = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}') \qquad \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}+1}(\mathtt{F}(\mathtt{M_1},\dots,\mathtt{M_n})) = \textstyle\sum_{i=1}^n \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}+1}(\mathtt{M_i})$$

$$\mathrm{SA}^{\mathtt{F}}_0(\mathtt{F}(\mathtt{M_1},\dots,\mathtt{M_n})) = \textstyle\sum_{i=1}^n \mathrm{SA}^{\mathtt{F}}_0(\mathtt{M_i}) + \sum_{r\in\mathcal{R}(\mathtt{F})} \|\mathtt{M}_r\|_0$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{s}\mathtt{M_1}\cdots\mathtt{M_n}) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{G}(\mathtt{M_1},\dots,\mathtt{M_n})) = \textstyle\sum_{i=1}^n \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M_i}) \qquad \textit{if } \mathtt{s} \in \{\mathtt{y},\mathtt{c}\}$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}((\lambda\mathtt{x}.\mathtt{M}')\mathtt{M_1}\cdots\mathtt{M_n}) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}') + \textstyle\sum_{i=1}^n \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M_i})$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}((\mathtt{let}\ \mathtt{N_1}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{N_2})\mathtt{M_1}\cdots\mathtt{M_n}) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{N_1}) + \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{N_2}) + \textstyle\sum_{i=1}^n \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M_i})$$

The following Lemma follows by the above definition.

Lemma 11. *We have* $\|\mathtt{M}\|_0 + \sum\{\mathrm{SA}_0^{\mathtt{G}}(\mathtt{M}) \mid \mathtt{G} \in \mathtt{M}\} \leqslant |\mathtt{M}|_0$. *Moreover for* $i \geq 0$ *we have* $\sum\{\mathrm{SA}_{\mathtt{i}}^{\mathtt{G}}(\mathtt{M}) \mid \mathtt{G} \in \mathtt{M}\} \leq |\mathtt{M}|_i$.

We remark that the above lemma gives a bound for all the function symbols of the program, but often we use it to give a bound only for one function symbol : $\mathrm{SA}_{\mathtt{i}}^{\mathtt{F}}(\mathtt{M}) \leqslant |\mathtt{M}|_i$. The following key lemma is the reason for which we have introduced $\mathrm{SA}_{\mathtt{i}}^{\mathtt{F}}(\mathtt{M})$:

Lemma 12. *If* $\mathtt{M}$ *is* (β, com)*-normal and* $\mathtt{M} \to_{\mathtt{Rec}_\mathtt{F}} \mathtt{M}'$ *at depth* i *then* $\mathrm{SA}_{\mathtt{i}}^{\mathtt{F}}(\mathtt{M}') < \mathrm{SA}_{\mathtt{i}}^{\mathtt{F}}(\mathtt{M})$.

Proof. Let $\mathtt{M} = \mathtt{Q}\{\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))\} \to_{\mathtt{Rec}_\mathtt{F}} \mathtt{Q}\{\sigma(\mathtt{N})\} = \mathtt{M}'$. It follows by induction on the shape of $\mathtt{Q}\{\}$ and on the depth i by using Definition 12 and Lemma 10.2. □

The above lemma will be useful to show that the number of Rec-reductions is bounded. Before, we need some properties on Rec-redexes w.r.t. other redexes.

Lemma 13.
1. *Reducing a* Rec*-redex at depth* i *cannot introduce a* β*-redex at depth* i.
2. *Reducing a* com*-redex at depth* i *cannot introduce a* Rec*-redex at depth* i.

Proof.
1. By typing constraints and since the r.h.s. of a definition case is normal.
2. Easy, by the shape of the reduct in a com-reduction. □

Note that from the above lemma follows that if $\mathtt{M}$ is β-normal at depth i and $\mathtt{M} \to_{\mathtt{Rec}} \mathtt{M}'$ at depth i then $\mathtt{M}'$ is β-normal at depth i and analogously, if $\mathtt{M}$ is $\mathtt{Rec}_\mathtt{F}$-normal at depth i and $\mathtt{M} \to_{\mathtt{com}} \mathtt{M}'$ at depth i then $\mathtt{M}'$ is $\mathtt{Rec}_\mathtt{F}$-normal at depth i.
In order to show that the number of Rec-reductions is bounded, we now need to consider the behaviour of Rec-reductions on Rec-redexes of other function symbols.

Lemma 14. *Consider* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$.

1. *A* $\mathtt{Rec}_{\mathtt{F}_\mathtt{i}}$*-reduction in* $\mathtt{M}$ *at depth* d *can introduce only* $\mathtt{F}_\mathtt{j}$ *for* $j \leq i$ *function symbols at a depth less or equal to* $d + \max\{\mathtt{d}(\mathtt{N}_\mathtt{j}) \mid \mathtt{N}_\mathtt{j}$ *body in a definition case of* $\mathtt{F}_\mathtt{i}\}$.
2. *If* $\mathtt{M}$ *is* (β, com)*-normal, a* $\mathtt{Rec}_{\mathtt{F}_\mathtt{i}}$*-reduction in* $\mathtt{M}$ *at depth* d *cannot introduce a* $\mathtt{rec}_{\mathtt{F}_\mathtt{j}}$*-redex for* $n \geq j > i$ *at depth* d.

Proof.
1. Substitutions are done at depth d of terms with degree at most $\max \mathtt{d}(\mathtt{N}_\mathtt{j})$.
2. Easy, blocked symbols remain blocked by point 1 and Lemma 10.2. □

From the above lemmas and Lemma 12 we have the following.

Corollary 15 ($\mathtt{Rec}_\mathtt{F}$-reductions bound) *Let* $\mathtt{M}$ *be* $(\beta, \mathtt{com})$*-normal at depth* i. *If* $\mathtt{M} \to^k_{\mathtt{Rec}_\mathtt{F}} \mathtt{M}'$ *at depth* i *then* $k \leq \mathrm{SA}_{\mathtt{i}}^{\mathtt{F}}(\mathtt{M})$.

Now we also need to control the term's size increase during a Rec-reduction step.

Lemma 16 (Size lemma). *If* $\mathtt{M} \to_{\mathtt{Rec}_\mathtt{F}} \mathtt{M}'$ *at depth* i *then for all* $j \geq i$ *we have* $|\mathtt{M}'|_j \leq |\mathtt{M}|_j + K_\mathtt{F}$.

Proof. Let $\mathtt{M} = \mathtt{M}_1\{\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))\} \to_{\mathtt{Rec}_\mathtt{F}} \mathtt{M}_1\{\sigma(\mathtt{N})\} = \mathtt{M}'$. By definition we have $|\mathtt{M}'|_j = |\mathtt{M}_1\{\}|_j + |\sigma(\mathtt{N})|_{j-i}$ and $|\mathtt{M}|_j + K_\mathtt{F} = |\mathtt{M}_1\{\}|_j + K_\mathtt{F} + |\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))|_{j-i}$. What we need to show is that $|\sigma(\mathtt{N})|_{j-i} \leq K_\mathtt{F} + |\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))|_{j-i}$. We consider the following two cases: $j - i = 0$ or $j - i > 0$. In the case $j - i = 0$

we have $|\sigma(\mathtt{N})|_0 = |\mathtt{N}|_0 + \sum_{\mathtt{X}\in_0 \mathrm{FO}(\mathtt{N})}(|\sigma(\mathtt{X})|_0 - 1)$ and $|\mathtt{F}(\sigma(\mathtt{t}_1),\ldots,\sigma(\mathtt{t}_n))|_0 = 1 + \sum_{k=1}^{n} |\mathtt{t}_k|_0 + \sum_{\mathtt{X}\in_0 \mathrm{FO}(\overrightarrow{\mathtt{t}})}(|\sigma(\mathtt{X})|_0 - 1)$. By definition $|\mathtt{N}|_0 \leq K_{\mathtt{F}}$, moreover by definition $\mathrm{FV}(\mathtt{N}) \subseteq \mathrm{FV}(\overrightarrow{\mathtt{t}})$ and by Lemma 2 every $\mathtt{X} \in \mathrm{FV}(\overrightarrow{\mathtt{t}})$ occurs at most once in N at depth 0. So we have $\sum_{\mathtt{X}\in_0 \mathrm{FO}(\mathtt{N})}(|\sigma(\mathtt{X})|_0 - 1) \leq \sum_{\mathtt{X}\in_0 \mathrm{FO}(\overrightarrow{\mathtt{t}})}(|\sigma(\mathtt{X})|_0 - 1)$. So the conclusion follows for this case. In the case $j - i = h > 0$ we have the same by Lemma 2: the pattern variables of $\overrightarrow{\mathtt{t}}$ occur linearly in N at depth 0. □

It remains to observe that com-reductions preserve our term measures.

Lemma 17. *Let* $\mathtt{M} \rightarrow_{com} \mathtt{M}'$ *then we have: (i)* $\mathtt{d}(\mathtt{M}) = \mathtt{d}(\mathtt{M}')$*, (ii)* $|\mathtt{M}'|_i = |\mathtt{M}|_i$ *for each* $i \leq \mathtt{d}(\mathtt{M})$*, and (iii)* $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M}) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M}')$ *for every* F *and every* $i \leq \mathtt{d}(\mathtt{M})$.

The above properties justify the next definition. We describe the reduction strategy *at a fixed depth* that we will use to bound the number of reduction steps of eLPL programs.

Definition 13 (standard reduction round). *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1},\ldots,\mathtt{d}_{\mathtt{F}_n}\ in\ \mathtt{M}$ *be a program. Then:*

- *a* standard reduction step *at depth* i*, denoted* $\mathtt{R} \twoheadrightarrow^{i}_{\mathtt{Rec}_\mathtt{F}} \mathtt{R}'$*, is a sequence of reductions at depth* i *of the shape:*
$$\mathtt{R} \rightarrow_{\mathtt{Rec}_\mathtt{F}} \mathtt{T} \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_1 \ \rightarrow_{\mathtt{Rec}_\mathtt{F}} \mathtt{T}_1 \rightarrow^{*}_{\mathtt{com}} \cdots \ \rightarrow_{\mathtt{Rec}_\mathtt{F}} \mathtt{T}_k \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_k \equiv \mathtt{R}'$$
such that every $\mathtt{R}_j$ *is* com*-normal and* $\mathtt{R}_k$ *is* $\mathtt{Rec}_\mathtt{F}$*-normal at depth* i*.*
- *a* standard reduction round *at depth* i*, denoted* $\mathtt{M} \Rightarrow^{i} \mathtt{M}'$*, is the following sequence of reductions at depth* i*:*
$$\mathtt{M} \rightarrow^{*}_{(\beta,\mathtt{com})} \mathtt{M}_0 \twoheadrightarrow^{i}_{\mathtt{Rec}_{\mathtt{F}_n}} \mathtt{M}_1 \twoheadrightarrow^{i}_{\mathtt{Rec}_{\mathtt{F}_{n-1}}} \cdots \twoheadrightarrow^{i}_{\mathtt{Rec}_{\mathtt{F}_1}} \mathtt{M}_n \rightarrow^{*}_{\dagger} \mathtt{M}'$$
such that $\mathtt{M}_0$ *is* $(\beta, \mathtt{com})$*-normal and* $\mathtt{M}'$ *is normal at depth* i*.*

When we need to stress the number k *of reduction steps in a standard reduction round we simply write it as* $\mathtt{M} \Rightarrow^{i}_{k} \mathtt{M}'$*.*

In order to show that the relation $\twoheadrightarrow^{i}$ is well defined for every term we need to prove that all the reductions are finite. First we need the following in order to have its direct corollary.

Lemma 18. *A sequence of reductions* $\rightarrow_{\mathtt{Rec}_\mathtt{F}}\rightarrow^{*}_{\mathtt{com}}$ *at depth* i *cannot introduce a* β*-redex at depth* i*.*

Proof. By typing constraints and by cases on Definition 9. □

Corollary 19 *If* M *is* β*-normal at depth* i *and* $\mathtt{M} \rightarrow_{\mathtt{Rec}_\mathtt{F}}\rightarrow^{*}_{\mathtt{com}} \mathtt{M}'$ *at depth* i *then* $\mathtt{M}'$ *is* β*-normal at depth* i*.*

Now we can prove that the relation $\twoheadrightarrow^{i}$ is well defined.

Lemma 20 (Bound on standard reduction step at depth i). *Let* M *be* $(\beta, \mathtt{com})$*-normal at depth* i*. If* $\mathtt{M} \twoheadrightarrow^{i}_{\mathtt{Rec}_\mathtt{F}} \mathtt{M}'$ *then* $\mathtt{M}'$ *is* $(\beta, \mathtt{com}, \mathtt{Rec}_\mathtt{F})$*-normal at depth* i*, the number of reductions is bounded by*
$$2 \times (|\mathtt{M}|_i)^3 \times (K_\mathtt{F} + 1)^2 \quad \textit{and for } j \geq i, \quad |\mathtt{M}'|_j \leq |\mathtt{M}|_j + |\mathtt{M}|_i \times K_\mathtt{F}.$$

Proof. By a detailed analysis of the standard reduction step $\mathtt{M} \twoheadrightarrow^{i}_{\mathtt{Rec_F}} \mathtt{M}'$ and by using Lemma 12, Lemma 8.2, Lemma 17.ii-iii, Corollary 19, Lemma 16 and Lemma 11. □

With this bound on standard reduction steps at fixed depth, we now state what we obtain whenever a standard round is done at fixed depth.

Theorem 3 (Bound on standard round at depth i). *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M}$ *be a program. Let* $\mathtt{M} \Rightarrow^{i}_{k} \mathtt{M}'$ *be a standard reduction round at depth* $i \geq 0$*. Then* $\mathtt{M}'$ *is normal at depth* i *and we have*

$$|\mathtt{M}'|_i \leq |\mathtt{M}|_i \times (K+1)^n \quad \textit{and} \quad k \leq 3 \times (|\mathtt{M}|_i)^3 \times (K+1)^{3n+2}$$

Proof. By analyzing the standard reduction round $\mathtt{M} \Rightarrow^{i}_{k} \mathtt{M}'$ and by using Lemma 8.1-2, Lemma 20, Lemma 14.1-2, Lemma 13.2, Lemma 12 and Lemma 17.ii □

Now we have a bound on a term size at fixed depth when we apply our strategy at the same depth. In order to bound the whole program execution we need next to examine what happens to the sizes at higher depth during the standard reduction round.

Lemma 21 (Size bound at depth greater than i, for a standard reduction round). *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M}$ *be a program. Let* $\mathtt{M} \Rightarrow^{i}_{k} \mathtt{M}'$ *be a standard reduction round at depth* $i \geq 0$*. Then we have*

$$|\mathtt{M}'|_{i+1} \leq |\mathtt{M}|_{i+1}|\mathtt{M}|_i \times (K+1)^n + (|\mathtt{M}|_i)^2 \times (K+1)^{2n+1}$$

Proof. By an analysis of the shape of the standard reduction round $\mathtt{M} \Rightarrow^{i}_{k} \mathtt{M}'$ and by using Lemma 8.1-2, Lemma 20 and Lemma 17. □

Corollary 22 *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ \mathtt{in}\ \mathtt{M}$ *be a program. Let* $\mathtt{M} \Rightarrow^{i} \mathtt{M}'$ *be a standard reduction round at depth* $i \geq 0$*. Then we have* $|\mathtt{M}'| \leq 2(|\mathtt{M}|)^2 \times (K+1)^{2n+1}$.

5.3 Bound on a program normalization

We apply our reduction strategy by standard rounds progressively at depths $0, 1, 2 \ldots$

Definition 14 (standard reduction). *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ in\ \mathtt{M}$ *be a program. A* standard reduction*, denoted* $\mathtt{M} \Rrightarrow \mathtt{M}'$*, is a sequence of standard reduction rounds of increasing depths of the shape:*

$$\mathtt{M} \Rightarrow^{0} \mathtt{M}_0 \Rightarrow^{1} \cdots \Rightarrow^{d-1} \mathtt{M}_{d-1} \Rightarrow^{d} \mathtt{M}'$$

To stress the number k *of total reduction steps we simply write it as* $\mathtt{M} \Rrightarrow_k \mathtt{M}'$.

Every standard reduction can be summarized as follows

$$\begin{array}{ll}
\mathtt{M} & \rightarrow^{*}_{\beta,com} \mathtt{M}^0_{k_0} \twoheadrightarrow^{0}_{\mathtt{Rec_{F_n}}} \mathtt{M}^0_{k_1} \twoheadrightarrow^{0}_{\mathtt{Rec_{F_{n-1}}}} \cdots \twoheadrightarrow^{0}_{\mathtt{Rec_{F_1}}} \mathtt{M}^0_{k_n} \rightarrow^{*}_{\dagger} \mathtt{M}_0 \\
\mathtt{M}_0 & \rightarrow^{*}_{\beta,com} \mathtt{M}^1_{0} \twoheadrightarrow^{1}_{\mathtt{Rec_{F_n}}} \mathtt{M}^1_{1} \twoheadrightarrow^{1}_{\mathtt{Rec_{F_{n-1}}}} \cdots \twoheadrightarrow^{1}_{\mathtt{Rec_{F_1}}} \mathtt{M}^1_{n} \rightarrow^{*}_{\dagger} \mathtt{M}_1 \\
\vdots & \\
\mathtt{M}_{m-1} & \rightarrow^{*}_{\beta,com} \mathtt{M}^m_{0} \twoheadrightarrow^{m}_{\mathtt{Rec_{F_n}}} \mathtt{M}^m_{1} \twoheadrightarrow^{m}_{\mathtt{Rec_{F_{n-1}}}} \cdots \twoheadrightarrow^{m}_{\mathtt{Rec_{F_1}}} \mathtt{M}^m_{n} \rightarrow^{*}_{\dagger} \mathtt{M}_m
\end{array}$$

To give an upper bound on standard reductions we need the notion of *potential depth*.

Definition 15 (Potential Depth). *Consider* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ *and an occurrence* $\mathtt{N}$ *of a subterm in* $\mathtt{M}$*. The* potential depth*,* $\mathtt{pt}_{\mathtt{d}}(\mathtt{N},\mathtt{p})$*, of* $\mathtt{N}$ *in* $\mathtt{p}$*, is defined as*

$$\mathtt{pt}_{\mathtt{d}}(\mathtt{N},\mathtt{p}) = \mathtt{d}(\mathtt{N},\mathtt{M}) + \sum_{i=1}^{n} \max_j \{\mathtt{d}(\mathtt{N}_i^j) \mid \mathtt{F}_i(\mathtt{t}_1^{\mathtt{j}}, .., \mathtt{t}_{\mathtt{n}}^{\mathtt{j}}) = \mathtt{N}_i^j \in \mathtt{d}_{\mathtt{F}_i}\}$$

The potential degree*,* $\mathtt{pt}_{\mathtt{d}}(\mathtt{p})$*, of* $\mathtt{p}$ *is the maximal potential depth of any subterm in* $\mathtt{M}$*.*

Even if standard reductions can increase the depth of a term, we have the following:

Lemma 23. *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, .., \mathtt{d}_{\mathtt{F}_n}\ in\ \mathtt{M}$ *be a program and* $\mathtt{M} \Rightarrow^0 \mathtt{M}_0 \Rightarrow^1 \cdots \Rightarrow^m \mathtt{M}_m$ *be a standard reduction. Then* $m < \mathtt{pt}_{\mathtt{d}}(\mathtt{p})$*.*

Proof. By double induction using Lemma 14. □

In the previous subsection we gave a bound on the number of program reduction steps at fixed depth when we apply a standard reduction round. In the previous lemma we stated that the potential depth is a bound on the possible depths to apply such standard reduction rounds. So our standard reduction normalizes a given program as follows:

Theorem 4. *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ *be an eLPL translated program satisfying the syntactic criterion and* $d = \mathtt{pt}_{\mathtt{d}}(\mathtt{p})$ *be its potential degree. Let* $\mathtt{M} \Rrightarrow_k \mathtt{M}'$ *be a standard reduction. Then,* $\mathtt{M}'$ *is normal, and* $|\mathtt{M}'| \in \mathcal{O}(|\mathtt{M}|^{2^{d+1}})$ *and* $k \in \mathcal{O}(|\mathtt{M}|^{3\times 2^d})$*.*

Proof. Looking at the shape of the standard reduction $\mathtt{M} \Rrightarrow_k \mathtt{M}'$ and by using Theorem 3, Lemma 23, Corollary 22. □

Corollary 24 *If* $\mathtt{p}$ *is is a closed LPL program which satisfies the syntactic criterion and with type* $\mathbb{D}_1 \multimap \S^i\mathbb{D}_2$*, where* i *is an integer and* $\mathbb{D}_1$*,* $\mathbb{D}_2$ *are ground types, then* $\mathtt{p}$ *represents a polynomial time function.*

Proof. If $\mathtt{v}$ is value of type $\mathbb{D}_1$ we consider the translation of $(\mathtt{p}\ \mathtt{v})$ in eLPL, and use the fact that its potential degree only depends on the type derivation of $\mathtt{p}$. Therefore using Theorem 4 the evaluation can be done in eLPL a polynomial number of steps, hence in polynomial time since the cost of each step can be polynomially bounded.

6 Conclusion and future developments

In this work we have introduced Light linear Programming Language (LPL), a typed functional programming language with pattern-matching, recursive definitions and higher-order types. The main feature of LPL is to give an implicit complexity characterization of PTIME where programming is more natural than in previous proposals. In order to ensure the PTIME soundness we have given a combined criterion composed of a syntactic restriction and a type system inspired by the one of Dual Light Affine Logic.
As future developments we consider the following directions:

- Verifying the effectiveness of our criterion and study the exact complexity of its checking. This study should lead to an efficient type inference procedure.
- Studying different ways of relaxing our criterion in order to improve the intensional expressiveness of LPL. One interesting direction is to include, in analogy with [5], recursive definitions of non-size increasing functions with a special status.
- Analyzing the relation between the strategy proposed here to prove the PTIME soundness and some standard evaluation strategies, e.g. lazy evaluation.

References

1. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2**(2) (1992) 97–110
2. Leivant, D.: A foundational delineation of computational feasibility. In Meyer, A.R., ed.: LICS '91, IEEE Computer Society (1991) 2–11
3. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. In: (TLCA '93). Volume 664 of LNCS., Springer (1993) 274–288
4. Girard, J.Y.: Light linear logic. Information and Computation **143(2)** (1998) 175–204
5. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. In: LICS 99. (1999) 464–473
6. Marion, J.Y., Moyen, J.Y.: Efficient first order functional program interpreter with time bound certifications. In: LPAR '00. Volume 1955 of LNCS., Springer (2000) 25–42
7. Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In: ACM ICFP'99. (1999) 70–81
8. Hofmann, M., Jost, S.: Static prediction of heap usage for first-order functional programs. In: ACM POPL'03, New Orleans, LA, USA (2003)
9. Crary, K., Weirich, S.: Resource bound certification. In: ACM POPL'00. (2000) 184–198
10. Atassi, V., Baillot, P., Terui, K.: Verification of ptime reducibility for system F terms via dual light affine logic. In: CSL '06. Volume 4207 of LNCS., Springer (2006)
11. Girard, J.Y.: Linear logic. TCS **50** (1987) 1–102
12. Girard, J.Y., Scedrov, A., Scott, P.J.: Bounded linear logic: a modular approach to polynomial-time computability. TCS **97(1)** (1992) 1–66
13. Lafont, Y.: Soft linear logic and polynomial time. TCS **318**(1-2) (2004) 163–180
14. Terui, K.: Light affine lambda calculus and polytime strong normalization. In: (LICS '01), IEEE Computer Society (2001) 209–220
15. Baillot, P., Mogbil, V.: Soft lambda-calculus: a language for polynomial time computation. In: FoSSaCS'04. Volume 2987 of LNCS., Springer (2004) 27–41
16. Baillot, P.: Checking polynomial time complexity with types. In: Proceedings of the 2nd IFIP International Conference on TCS. (2002) 370–382
17. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. Information and Computation **207**(1) (2009) 41–62
18. Gaboardi, M., Ronchi Della Rocca, S.: A soft type assignment system for λ-calculus. In: CSL '07. Volume 4646 of LNCS., Springer (2007) 253–267
19. Gaboardi, M., Ronchi Della Rocca, S.: Type inference for a polynomial lambda calculus. In: TYPES'08. Volume 5497 of LNCS., Springer (2008) 136–152
20. Bellantoni, S., Niggl, K.H., Schwichtenberg, H.: Higher type recursion, ramification and polynomial time. APAL **104** (2000) 17–30
21. Dal Lago, U., Martini, S., Roversi, L.: Higher order linear ramified recurrence. In: TYPES'03. Volume 3085 of LNCS., Springer (2003) 178–193
22. Dal Lago, U.: The geometry of linear higher-order recursion. ACM TOCL **10**(2) (2009)
23. Marion, J.Y.: Analysing the implicit complexity of programs. Information and Computation **183**(1) (2003) 2–18
24. Bonfante, G., Marion, J.Y., Moyen, J.Y.: On lexicographic termination ordering with space bound certifications. In: PSI 2001. Volume 2244 of LNCS., Springer (2001) 482–493
25. Amadio, R.M.: Synthesis of max-plus quasi-interpretations. Fundamenta Informaticae **65**(1-2) (2005) 29–60
26. Marion, J.Y., Péchoux, R.: Characterizations of polynomial complexity classes with a better intensionality. In: ACM-PPDP'08. (2008) 79–88
27. Marion, J.Y., Péchoux, R.: Sup-interpretations, a semantic method for static analysis of program resources. ACM TOCL **10**(4) (2009)