A PolyTime Functional Language from Light Linear Logic

Patrick Baillot, Marco Gaboardi, Virgile Mogbil

A PolyTime Functional Language from Light Linear Logic

(rapport interne LIPN - Janvier 2010)

Patrick Baillot
LIP - UMR 5668 CNRS ENS-Lyon UCBL INRIA - Université de Lyon

Marco Gaboardi
Dipartimento di Informatica, Università degli studi di Torino

Virgile Mogbil
LIPN - UMR7030, CNRS - Université Paris 13

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 [BC92, Lei91, LM93, Gir98]. In a second step, e.g. [Hof99, MM00], 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 following other approaches, e.g. [HP99, HJ03, CW00]. 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 fors a Turing-complete language the class of 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 ...
3. how effective is the criterion: what is the complexity of the corresponding decision problem.

Result 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 aspect 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` constructor,
- 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 term to obtain exponential time programs like the following

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

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 programs return 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 § 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, we mean a function of type $\mathbb{N} \multimap \mathbb{N}$ to be such that it can increase the size of its input by at most a constant, while a function of type $\mathbb{N} \multimap \S\mathbb{N}$ to be such that it can increase it at most quadratically. For a recursive definition of the shape $\mathtt{f\ t} = \mathtt{M}\{\mathtt{f\ t'}\}$, the typing rule ensures that the context `M` does not increase the size too much, and it types the function `f` accordingly. Therefore 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. E.g. consider again the `foldr` function, we can type a term representing one of its classical use as

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

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 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 [ABT06], but we leave the development of this point for future work.

Related works.

Higher-order calculi: linear logic and linear type systems. Linear logic [Gir87] 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 [GSS92] and led to *Light Linear Logic* (LLL) in [Gir98] and *Soft Linear Logic* (SLL) in [Laf04]. Starting from the principles underlying these logics different PTIME term languages have been proposed in [Ter01] for LLL and in [BM04] for SLL. In a second step [Bai02] type systems as criteria for λ-calculus were designed out of this logic, like the system DLAL [BT09] and STA [GRDR07]. 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): one can decide in polynomial time if a (system F typed) λ-term is typable in DLAL [ABT06] and if a pure λ-term is typable in (propositional) STA [GR08].
In [BNS00] the authors propose a language for PTIME extending to higher-order the characterizations based on *ramification* [BC92, Lei91]. The language is a *ramified* vari-

ant 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 [DLMR03, DL09].
In [Hof99], Hofmann proposed a typed λ-calculus with recursor (essentially a variant of Gödel's system T), LFPL, overcoming the problems of ramification and 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 argument is similar to the one in [BNS00] and the system T programming style make it far from ordinary functional languages.
First-order calculi and interpretations. Starting from the ICC characterizations based on *ramification* such as [BC92, Lei91], a progressive work of generalization was carried out by Marion and collaborators, trading first primitive recursion for termination orderings on constructor rewrite systems [Mar03], and then ramification for notions of *quasi-interpretation* [MM00, BMM01] and *sup-interpretation* [MP08, MP09] over a first-order functional language with recursion and pattern-matching. Quasi-interpretations and sup-interpretations are a semantic notion inspired from polynomial interpretations, and which essentially statically provides 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 advantage of this method is its intensional expressivity, since more algorithms are validated (point 1) than in the ramification-based frameworks.

Outline.

In Section 2 we introduce LPL. We present its syntax, its type system and the syntactic criterion that its program should meet. In Section 3 we show some programming examples in LPL. In Section 4 we introduce an extended language, named eLPL, where the stratification measures inherited by the type system are explicit. Moreover, we give a translation of each LPL programs in it. Then, in Section 5 we give the translations in eLPL of the examples of Section 3.
In Section 6 we prove the polynomial time soundness of LPL programs by means of their translation in eLPL. The proof shows that a depth-by-depth strategy normalize LPL programs in polynomial time in the size of their input. Finally, in Section 7 we briefly outline how to extend the language in order to achieve PTIME completeness.

2 LPL

We introduce the language LPL, an extension of λ-calculus by constants, 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 ingredient 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.

$\mathtt{p}$	$::=$	$\mathtt{M} \mid \mathtt{LetRec}\ \mathtt{d_F}\ \mathtt{in}\ \mathtt{p}$	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: LPLterm language definition

2.1 The syntax

Let the set Var be a denumerable set of *variables*, the set PVar be a denumerable set of *pattern variables*, the set Cst be a denumerable set of *constructors* and the set Fct be a denumerable 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 programs 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 a subset of terms.
The size $|\mathtt{M}|$ of a term $\mathtt{M}$ is the number of symbols occurring in it. The size of patterns and 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 $\mathtt{s}\ \mathtt{M}_1 \ldots \mathtt{M}_n$.

Example. *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. 7). Moreover, 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 $\mathtt{F}(\mathtt{t}_1^1, \ldots, \mathtt{t}_n^1), \ldots, \mathtt{F}(\mathtt{t}_1^k, \ldots, \mathtt{t}_n^k)$; 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.

Remark. *Note that the condition 1 above implies that definition cases of basic functions do not contain any recursive calls. The condition 2(i) is merely a technical assumption which will make it easier to obtain the complexity bound, but we expect that it could be dropped.*

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 $\mathtt{t} \not\preceq \mathtt{t}'$ and $\mathtt{t}' \not\preceq \mathtt{t}$ are *incomparable*.
A *program* is a term M without free pattern variables preceded by a sequence of function definitions $\mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}$ defining all the function symbols occurring in M. Moreover we ask that every function definition $\mathtt{d}_{\mathtt{F}_i}$ uses only the function symbols $\mathtt{F}_1, \ldots, \mathtt{F}_{i-1}$. We usually write a program of the shape $\mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}\ \mathtt{in}\ \cdots\ \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ simply as $\mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$. As usual programs are considered up to renaming of bound variables.
A *substitution* σ is a mapping replacing variables by terms. The notion of substitution can be used to define the notion of matching which is essential in defining the reduction mechanism of our language.

Definition 1. *Given a term* M *and a pattern* t *we say that* M *matches* t *if and only if there exists a substitution* σ *such that* $\mathtt{M} = \sigma(t)$. *Analogously, given a term* M *and a definition case of the shape* $\mathtt{F}(\mathtt{t}_1, \ldots, \mathtt{t}_n) = \mathtt{N}$ *we say that* M *matches it if and only if there exists a substitution* σ *such that* $\mathtt{M} = \mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))$.

Definition 2. *A sequence* $\mathtt{d}_1, \ldots, \mathtt{d}_n$ *of function definition cases for the function symbol* F *of arity* n *is* exhaustive *if for every sequence of values* $V_1, \ldots, V_n$ *such that* $\mathtt{F}(\mathtt{V}_1, \ldots, \mathtt{V}_n)$ *is typable, there exists a unique* $\mathtt{d}_i$ *in* $\mathtt{d}_1, .., \mathtt{d}_n$ *such that* $\mathtt{F}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$ *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*.

Definition 3. *Let* $\mathtt{F}(\mathtt{t}_1, \ldots, \mathtt{t}_n) = \mathtt{M}$ *be a definition case for the function symbol* F. *It is* recursive *if it contains some recursive calls* $\mathtt{F}(\mathtt{t}_1^1, \ldots, \mathtt{t}_n^1), \ldots, \mathtt{F}(\mathtt{t}_1^m, \ldots, \mathtt{t}_n^m)$ *of* F *in* M. *It is* base *otherwise.*

Note that basic funtions by definition only have base definition cases. We now need the following notion of safe definition cases.

Definition 4 (Safe definition case)**.** *Let* $\mathtt{F}(\mathtt{t}_1, \ldots, \mathtt{t}_n) = \mathtt{M}$ *be a definition case for the function symbol* F. *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

(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 using the notion of safe definition case.

Definition 5 (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 6 (Matching argument)**.** *Let* $\mathtt{F}(\mathtt{t_1},\ldots,\mathtt{t_n}) = \mathtt{M}$ *be a definition case for the function symbol* F*. 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 matching positions in some definition case of* F*. The* matching arguments *of a function symbol* F *are the arguments in a matching position of* $\mathcal{R}(\mathtt{F})$*.*

Note that in Definition 4 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

$$\mathtt{exp(s\ X)} = (\mathtt{exp\ X}) + (\mathtt{exp\ X})$$

Nevertheless, we have functions with more recursive calls over trees, for example:

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

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* $\mathtt{F}(\mathtt{t}^1_1,..,\mathtt{t}^1_n),\ldots,\mathtt{F}(\mathtt{t}^m_1,..,\mathtt{t}^m_n)$*. Then*

$$\sum_{i=1}^{n} |\mathtt{t}_i| > \sum_{k=1}^{m}(\sum_{i=1}^{n} |\mathtt{t}^k_i|)$$

Proof. The case $m = 1$ follows directly by conditions (i) and (ii) of Definition 4. In the case $m > 1$ conditions (i) and (iii) of Definition 4 imply that for every i, the $\mathtt{t}^k_i$ for $1 \leq k \leq m$ are pairwise disjoint subterms of $\mathtt{t}_i$, and thus $|\mathtt{t}_i| \geq \sum_{k=1}^{m} |\mathtt{t}^k_i|$. Consider $k = 1$, by (ii) there exists i_1 such that $\mathtt{t}^1_{i_1} \prec \mathtt{t}_{i_1}$. It follows that $|\mathtt{t}_{i_1}| > \sum_{k=1}^{m} |\mathtt{t}^k_{i_1}|$. Therefore $\sum_{i=1}^{n} |\mathtt{t}_i| > \sum_{i=1}^{n}(\sum_{k=1}^{m} |\mathtt{t}^k_i|)$. □

2.3 Reduction

The computational mechanism of LPLwill 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 7. *The reduction relation* $\rightarrow_{\mathtt{L}}$ *is the contextual closure of:*

- *the relation* $\rightarrow_\beta$ *defined as:* $(\lambda \mathtt{x.M})\mathtt{N} \rightarrow_\beta \mathtt{M}[\mathtt{N}/\mathtt{x}]$*,*
- *the relation* $\rightarrow_\gamma$ *defined for basic functions* $\underline{\mathtt{F}}$ *by:*

$$\mathtt{LetRec}\ \underline{\mathtt{F}}(\vec{\mathtt{t}}_1) = \mathtt{M}_1,\ldots,\underline{\mathtt{F}}(\vec{\mathtt{t}}_n) = \mathtt{M}_n\ \mathtt{in}\ \mathtt{M}\{\underline{\mathtt{F}}(\overrightarrow{\mathtt{N}})\} \rightarrow_\gamma$$
$$\mathtt{LetRec}\ \underline{\mathtt{F}}(\vec{\mathtt{t}}_1) = \mathtt{M}_1,\ldots,\underline{\mathtt{F}}(\vec{\mathtt{t}}_n) = \mathtt{M}_n\ \mathtt{in}\ \mathtt{M}\{\sigma(\mathtt{M_i})\}$$

if there exists i *s.t.* $\sigma(\overrightarrow{\mathtt{t}_i}) = \overrightarrow{N}$*.*

- *and of the relation* $\rightarrow_{\mathtt{Rec}}$ *defined as* $\rightarrow_\gamma$ *but for non-basic functions.*

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

1: Constructors and functions

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

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

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

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

2: Terms

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

3: Recursive definitions and programs

Table 2: LPL Typing rules

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

Remark that the syntactic criterion alone implies that a program M satisfying it cannot have an infinite $\rightarrow_{\mathtt{Rec}}$ reduction sequence.

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.
The types of LPL are defined starting from a set of *ground* types containing $\mathbb{B}_n$, $\mathbb{N}$, $\mathbb{L}_n$, $\mathbb{L}$, $\mathbb{T}$ representing respectively finite types with n elements (for any $n \leq 1$, 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: $\mathbb{D}_1 \times \mathbb{D}_2$. The elements of such a type 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:

$$\begin{array}{lll} \mathbb{D} & ::= & \mathbb{B}_n \mid \mathbb{N} \mid \mathbb{L}_n \mid \mathbb{L} \mid \mathbb{T} \mid \mathbb{D} \times \mathbb{D} \\ A & ::= & \mathbb{D} \mid A \multimap B \mid !A \multimap B \mid \S A \ . \end{array}$$

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. The other modality $\S A$ is used in Light linear logic [Gir98] (and in DLAL) to guarantee a PTIME bound normalization. A formula A is *modal* if it is of the form $A = \S B$ or $!B$.

The intuitions behind modalities will be clarified by the type assignment system of LPL (Table 2). Just keep in mind for the moment that in our types the ! modality is always used in the left position of a linear implication. We write † for modalities in $\{!, \S\}$, and $\dagger^n A = \dagger(\dagger^{n-1} A)$, $\dagger^0 A = A$.
The following definition gives types to constructors and functions:

Definition 8. *To each constructor or (basic or non-basic) function symbol s, of arity n, a fixed type is associated, denoted $\mathcal{T}(s)$:*

- *If $s = \mathtt{c}$ or $\underline{\mathtt{F}}$ then: $\mathcal{T}(s) = \mathbb{D}_1 \multimap \cdots \multimap \mathbb{D}_n \multimap \mathbb{D}_{n+1}$, where the $\mathbb{D}_i$ for $1 \leq i \leq n+1$ are ground types.*
- *If $s = \mathtt{F}$ a non-basic function, then: $\mathcal{T}(\mathtt{F}) = !^{i_1}\S^{j_1} A_1 \multimap \cdots \multimap !^{i_n}\S^{j_n} A_n \multimap \S^j A$ where:*
  - *i) every A_i, for $1 \leq i \leq n$, and A are non-modal formulas,*
  - *ii) $j \geq 1$ and $0 \leq i_r \leq 1$ for any $1 \leq r \leq n$,*
  - *iii) 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 $i_r + j_r \geq 1$.*

Example. *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: $\mathcal{T}(\epsilon) = \mathbb{T}$, $\mathcal{T}(\mathtt{node}) = \mathbb{N} \multimap \mathbb{T} \multimap \mathbb{T} \multimap \mathbb{T}$.*

The type assignment system we present here is a declarative one, that is to say it is designed to favor simplicity, rather than to make type inference easy. In particular the typing rules will not be syntax-directed. We could design an equivalent algorithmic type system, meant to be used for type inference, but this is not our goal here.
Contexts are sets of assignments of either one of the following shapes $\mathtt{x} : A$, $\mathtt{X} : A$ where A is a type. Note in particular that function symbols and constructor symbols will not occur in contexts. Contexts will be usually denoted as $\Gamma, \Delta, \Gamma_1, \ldots$
As already stressed we need different kinds of type judgments. In particular we have *type judgments for terms and programs* that 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*: 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 a context $\varkappa_1 : A_1, \ldots, \varkappa_n : A_n$ then $\S\Delta$ will stand for the context $\varkappa_1 : \S A_1, \ldots, \varkappa_n : \S A_n$.
Moreover we have *type judgments for function definitions* of the shape $\triangleright \mathtt{d_F} : A$, where $\mathtt{d_F}$ is a definition of $\mathtt{F}$ (possibly not completed yet) and A is a type.
The typing rules for LPL are depicted in Table 2. We will now discuss the three classes of rules (2.1, 2.2 and 2.3). In binary rules, like $(\Rightarrow E)$, the contexts of the two premises are assumed to have disjoint sets of variables.
In Table 2.1, the rule for function symbols might seem a bit surprising at first sight because the type of the function is assigned before the function is concretely defined, but recall that this presentation is not tailored for type inference. The typing rules for terms in Table 2.2 are not new and are essentially those of the type system DLAL [BT09] for λ-calculus. A minor difference is that here we have both variables and pattern variables.
The typing rules dealing with definitions are presented in Table 2.3 and together with

the typing rule for function symbols are the main novelty of the present language. They need some comments. The rule (D) serves to add a definition case to a partial definition $d_{\mathtt{F}}$ of $\mathtt{F}$. The new definition typed is then $\mathtt{d}'_{\mathtt{F}} = (\mathtt{F}(\overrightarrow{\mathtt{t}_i}) = \mathtt{N}_i), \mathtt{d}_{\mathtt{F}}$.
The rule (R) then serves to form a new program from a program and a definition of a function.
As usual the set Val of *values* corresponds to the typable terms of the algebra freely generated (respecting the arity) by constructors.
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* $\Gamma; \Delta \vdash \mathtt{q} : A$.

3 Some Examples

Before giving the details of LPL complexity properties, we give here some hints on how to program in LPL. In particular more information about the typing can be found in Section 5.
The standard recursive definition of addition turns in a function definition $\mathtt{d}_{\mathtt{A}}$ as:

$$\begin{array}{lcl} \mathtt{Add}(\mathtt{s}(\mathtt{X}),\ \mathtt{Y}) & = & \mathtt{s}(\mathtt{Add}(\mathtt{X},\mathtt{Y}))\ , \\ \mathtt{Add}(\mathtt{0},\mathtt{Y}) & = & \mathtt{Y} \end{array}$$

the first argument is a recurrence arguments, so $\mathtt{d}_{\mathtt{A}}$ is typable for example by taking $\mathtt{Add} : \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}$. Analogously, the standard recursive definition of multiplication turns in a function definition $\mathtt{d}_{\mathtt{M}}$ as:

$$\begin{array}{lcl} \mathtt{Mul}(\mathtt{s}(\mathtt{X}),\mathtt{Y}) & = & \mathtt{Add}(\mathtt{Y},\mathtt{Mul}(\mathtt{X},\mathtt{Y}))\ , \\ \mathtt{Mul}(\mathtt{0},\mathtt{Y}) & = & \mathtt{0} \end{array}$$

since the first is a recurrence argument and since $\mathtt{Add} : \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}$ we can type $\mathtt{d}_{\mathtt{M}}$ by taking $\mathtt{Mul} : \mathbb{N} \multimap !\mathbb{N} \multimap \S\S\mathbb{N}$ and by typing accordingly the right-hand side of the definition cases by means of rule $(\S I)$ and $(\S E)$.
We can also program a type coercion function for every data type. On numerals this can be obtained by a function definition $\mathtt{d}_{\mathtt{C}}$ as:

$$\begin{array}{lcl} \mathtt{Coer}(\mathtt{s}(\mathtt{X})) & = & \mathtt{s}(\mathtt{Coer}(\mathtt{X}))\ , \\ \mathtt{Coer}(\mathtt{0}) & = & \mathtt{0} \end{array}$$

typable by taking $\mathtt{Coer} : \mathbb{N} \multimap \S\mathbb{N}$ and by typing accordingly the right-hand side of the definition cases by means of rule $(\S I)$ and $(\S E)$. The above coercion can be used to define the usual Map function on lists of numerals. We have a function definition $\mathtt{d}_{\mathtt{P}}$ as:

$$\begin{array}{lcl} \mathtt{Map}(\mathtt{Y},\mathtt{X}:\mathtt{XS}) & = & (\mathtt{Y}(\mathtt{Coer}(\mathtt{X}))) : \mathtt{Map}(\mathtt{Y},\mathtt{XS})\ , \\ \mathtt{Map}(\mathtt{Y},\mathtt{nil}) & = & \mathtt{nil} \end{array}$$

that is typable by taking $\mathtt{Map} :!(\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{L} \multimap \S\mathbb{L}$. Note that we have also a typing for non linear function argument of Map. Using the function definitions we have just defined we can write a program

$$\mathsf{Map}\ (\times 2)\ (1 : 2 : 3 : 4)$$

that doubles all the elements of the given list $(1 : 2 : 3 : 4)$ as follows

$$\mathtt{LetRec}\ \mathtt{d}_{\mathtt{A}}, \mathtt{d}_{\mathtt{M}}, \mathtt{d}_{\mathtt{C}}, \mathtt{d}_{\mathtt{P}}\ \mathtt{in}\ \mathtt{Map}\ (\lambda \mathtt{x}.\mathtt{Mul}(\mathtt{x},\mathtt{2}), (\mathtt{1} : \mathtt{2} : \mathtt{3} : \mathtt{4}))$$

such a program is typable with type $\S\S\S\mathbb{L}$ using $\mathtt{Map} :!(\mathbb{N} \multimap \S\S\mathbb{N}) \multimap \mathbb{L} \multimap \S\S\S\mathbb{L}$.
Finally, by using again the coercions we can write a function definition $\mathtt{d_R}$ for the standard recursive definition of Foldr as:

$$\begin{array}{lcl} \mathtt{Foldr}\ (\mathtt{Y}, \mathtt{Z}, \mathtt{X} : \mathtt{XS}) & = & \mathtt{Y}\ (\mathtt{Coer}(\mathtt{X}))\ (\mathtt{Foldr}(\mathtt{Y}, \mathtt{Z}, \mathtt{XS}))\ , \\ \mathtt{Foldr}(\mathtt{Y}, \mathtt{Z}, \mathtt{nil}) & = & \mathtt{Z} \end{array}$$

such a function definition is typable by $\mathtt{Foldr} :!(\mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}) \multimap \S\mathbb{N} \multimap \mathbb{L} \multimap \S\mathbb{N}$ and again by typing accordingly the right-hand side of the definition cases by means of rule $(\S I)$ and $(\S E)$. Note that we have also a typing $\mathtt{Foldr} :!(\mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}) \multimap \S\S\mathbb{N} \multimap \mathbb{L} \multimap \S\S\mathbb{N}$. So we can use foldr to write the program

$$\mathsf{Foldr}\ (+)\ 0\ (1 : 2 : 3)$$

that sums the values in the list $(1 : 2 : 3)$ as

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

and this is typable with type $\S\S\mathbb{N}$.
Finally we can also give some interesting programs over trees. For example we have the following addition of trees. We have a function definition $\mathtt{d_T}$ as:

$$\begin{array}{lcl} \mathtt{Tadd}(\mathtt{node}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}), \mathtt{node}(\mathtt{X'}, \mathtt{Y'}, \mathtt{Z'})) & = & \mathtt{node}(\mathtt{Add}(\mathtt{X}, \mathtt{X'}), \mathtt{Tadd}(\mathtt{Y}, \mathtt{Y'}), \mathtt{Tadd}(\mathtt{Z}, \mathtt{Z'}))\ , \\ \mathtt{Tadd}(\epsilon, \mathtt{X}) & = & \mathtt{X}\ , \\ \mathtt{Tadd}(\mathtt{X}, \epsilon) & = & \mathtt{X}\ , \end{array}$$

we can type $\mathtt{d_T}$ by taking $\mathtt{Tadd} : \mathbb{T} \multimap \mathbb{T} \multimap \S\mathbb{T}$ and by typing accordingly the right-hand side of the definition cases by means of rule $(\S I)$ and $(\S E)$.

4 Translating LPL in eLPL

The proof of the polynomial time complexity bound for light linear logic [Gir98] and light λ-calculus [Ter01] 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 [Ter01], 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 programs is depicted in Table 3. An eLPL term $\lambda \mathtt{x}.\mathtt{let}\ \mathtt{x}\ \mathtt{be}\ !\mathtt{y}\ \mathtt{in}\ \mathtt{M}[\mathtt{y}/\mathtt{x}]$, where y is fresh, is abbreviated by $\lambda^{!}\mathtt{x}.\mathtt{M}$. 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 terms part of LPL programs.
The contexts of typing judgements for eLPL terms and programs can contain a new kind of type declaration, denoted $x : [A]_{\S}$, 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 abstracted. This kind of declarations is made necessary by the fact that eLPL is handling explicitly stratification. If $\Delta = x_1 : A_1, \ldots, x_n : A_n$ then $[\Delta]_{\S}$ is $x_1 : [A_1]_{\S}, \ldots, x_n : [A_n]_{\S}$. The typing rules are are given in Table 4. Note that declarations $x : [A]_{\S}$ are introduced by (§ I) rules,

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

Table 3: eLPL term language definition

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

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

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

Table 4: eLPL type system

and eliminated by (§ E) rules. Observe that if $\lambda x.\mathtt{M}$ is a well typed eLPL term, then $n_o(x, \mathtt{M}) \leq 1$.

Note that all the rules in Table 4, but the rule $(l!)$, are the same rules as in Table 2 but for 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 9. *Let* $\mathtt{M}$ *be an* LPL *term and* Π *be a type derivation in* LPL *proving the type judgement* $\Gamma; \Delta \vdash \mathtt{M} : B$*. 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. each* $x : [A]_{\S}$ *in* Δ*) an occurrence of the rule* $(l!)$ *(resp. of the rule (*§ *E) with a l.h.s. premise of the form* $; y : \S A \vdash y : \S A$*).*

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

Definition 10. *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:*

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

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

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

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

Definition 11. *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ *be an* LPL *program and let* Π *be a type derivation in* LPL *proving the judgment* $\Gamma; \Delta \vdash \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M} : B$. *Then,* Π^* *is the type derivation in* eLPL *proving* $\Gamma; \Delta \vdash \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}{}^*, \ldots, \mathtt{d}_{\mathtt{F}_n}{}^*\ \mathtt{in}\ \mathtt{M}^* : B$ *obtained by replacing every derivation* $\Sigma_i :: \rhd \mathtt{d}_{\mathtt{F}_i} : \S B$ *in* Π *by the derivation* $\Sigma_i^* :: \rhd \mathtt{d}_{\mathtt{F}_i}{}^* : \S B$ *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* $\varkappa \in \mathrm{FV}(\mathtt{M}^*)$.

Proof. Easy. □

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

Definition 12. *The reduction relation* $\to_{\mathtt{I}}$ *is the contextual closure of the relations* $\to_{\mathtt{Rec}}$, $\to_\gamma$ *(as in Def. 7) and of the reductions* $\to_\beta$, $\to_!$, $\to_\S$, $\to_{\mathtt{com}_1}, \to_{\mathtt{com}_2}$ *and* $\to_{\mathtt{com}_3}$ *for* $\dagger \in \{!, \S\}$ *defined as:*

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

$$\mathtt{M}(\mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{V}) \to_{\mathtt{com}_1} \mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ (\mathtt{MV}),$$

$$(\mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{V})\mathtt{M} \to_{\mathtt{com}_2} \mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ (\mathtt{VM}),$$

$$\mathtt{let}\ (\mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{V})\ \mathtt{be}\ \dagger\mathtt{y}\ \mathtt{in}\ \mathtt{W} \to_{\mathtt{com}_3} \mathtt{let}\ \mathtt{U}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ (\mathtt{let}\ \mathtt{V}\ \mathtt{be}\ \dagger\mathtt{y}\ \mathtt{in}\ \mathtt{W}).$$

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

We write $\to_{com}$ for anyone of the three *commutation reductions* $\to_{com_i}$. Note that:

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

In fact, a $\to_\beta$ step in LPL is decomposed in eLPL into a (linear) $\to_\beta$ step followed by a $\to_!$ step.
We write $\mathtt{let}\ \mathtt{M}\ \mathtt{be}\ \dagger^{\mathtt{n}}\mathtt{x}\ \mathtt{in}\ \mathtt{N}$ to denote $\mathtt{let}\ \mathtt{M}\ \mathtt{be}\ \dagger\mathtt{x}_1\ \mathtt{in}\ (\mathtt{let}\ \mathtt{x}_1\ \mathtt{be}\ \dagger\mathtt{x}_2\ \mathtt{in}\ (\ldots(\mathtt{let}\ \mathtt{x}_{n-1}\ \mathtt{be}\ \dagger\mathtt{x}_n\ \mathtt{in}\ \mathtt{N})\cdots)$.
Now, to reason about the stratification we define the notion of *depth*.

Definition 13. *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. *It is inductively defined as:*

$$\mathtt{d}(\mathtt{N}, \mathtt{N}) = 0 \qquad \mathtt{d}(\mathtt{N}, \lambda\mathtt{x}.\mathtt{P}) = \mathtt{d}(\mathtt{N}, \mathtt{P}) \qquad \mathtt{d}(\mathtt{N}, \dagger\mathtt{P}) = \mathtt{d}(\mathtt{N}, \mathtt{P}) + 1$$

$$\mathtt{d}(\mathtt{N}, \mathtt{let}\ \mathtt{P}\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{Q}) \;=\; \mathtt{d}(\mathtt{N}, \mathtt{PQ}) \;=\; \begin{cases} \mathtt{d}(\mathtt{N}, \mathtt{P}) & \textit{if } \mathtt{N} \textit{ occurs in } \mathtt{P} \\ \mathtt{d}(\mathtt{N}, \mathtt{Q}) & \textit{if } \mathtt{N} \textit{ occurs in } \mathtt{Q} \end{cases}$$

The degree *of an eLPL term* $\mathtt{M}$*, denoted by* $\mathtt{d}(\mathtt{M})$*, is the maximal depth of any subterm in it.*

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.
In the sequel we will use the following characterization of eLPL terms shape in order to reason about them.

Lemma 3 (eLPL term shape)**.** *Let* $\Gamma;\Delta \vdash \mathtt{M} : A$ *then it is of the following shape:* $\mathtt{M} = \lambda \mathtt{x}_1 \ldots \mathtt{x}_n \xi \mathtt{M}_1 \cdots \mathtt{M}_m$ *where* ξ *could be:*

- *a symbol* $\mathtt{c}$ *or a symbol* $\mathtt{F}$*, in these cases* m *is less than or equal to the arity of* ξ*.*
- *a term of the shape* $\dagger\mathtt{N}$*, in this case* $m = 0$*.*
- *a variable* $\varkappa$*, or a term of either the shape* $(\lambda \mathtt{x}.\mathtt{P})\mathtt{Q}$ *or* $\mathtt{let\ P\ be\ }\dagger\mathtt{x\ in\ Q}$*.*

Now we can state some important properties of typing on eLPL terms.

Lemma 4 (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* $n_o^0(\varkappa,\mathtt{M}) \leq 1$*.*

Proof. By induction on the derivation proving $\Gamma;\Delta \vdash \mathtt{M} : A$. □

We can now state a

Lemma 5 (Generation Lemma)**.**

1. *Let* $\Gamma;\Delta \vdash \lambda\mathtt{x}.\mathtt{M} : C$*, then either* $C = A \multimap B$ *or* $C = A \Rightarrow B$*.*
2. *Let* $\Gamma;\Delta \vdash \mathtt{c}\mathtt{t}_1 \ldots \mathtt{t}_n : \mathbb{D}$*, then each* $\mathtt{X} \in \mathrm{FV}(\mathtt{t}_i)$ *is such that* $\mathtt{X} : \mathbb{D}'$ *for some* $\mathbb{D}'$ *and* $\mathtt{d}(\mathtt{X},\mathtt{t}_i) = 0$*.*
3. *Let* $\triangleright\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) = \mathtt{N}$ *be a definition case for the function symbol* $\mathtt{F}$ *such that* $\Gamma;\Delta \vdash \mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) : \S B$ *and* $\Gamma;\Delta \vdash \mathtt{N} : \S B$*. If* $\mathtt{X} \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta)$*, then* $\mathtt{d}(\mathtt{X},\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n)) = \mathtt{d}(\mathtt{X},\mathtt{N})$*. If* $\mathtt{X} \in \mathrm{FV}(\mathtt{t}_i)$ *and* $\mathtt{t}_i$ *is a matching argument, then* $\mathtt{d}(\mathtt{X},\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n)) = 0$*.*
4. *Let* $\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_n)$ *be a subterm in* $\mathtt{M}$*. Then* $\mathtt{d}(\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_n),\mathtt{M}) = \mathtt{d}(\mathtt{M}_1,\mathtt{M}) = \cdots = \mathtt{d}(\mathtt{M}_n,\mathtt{M})$*.*

Lemma 6. *Let* $\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) = \mathtt{N}$ *be a recursive definition case for the function symbol* $\mathtt{F}$ *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,* $\mathtt{d}(\mathtt{F}(\mathtt{t}_1^i,\ldots,\mathtt{t}_n^i),\mathtt{N}) = 0$*.*

Proof. By definition, $\mathtt{N}$ should be in normal form, so in particular for every $\mathtt{F}(\mathtt{t}_1^j,\ldots,\mathtt{t}_n^j)$ there is a $\mathtt{t}_k^j$ corresponding to a matching argument such that $\vdash \mathtt{t}_k^j : \mathbb{D}$, and $\mathtt{t}_k^j$ should have at least one free variable $\mathtt{X}$. By Lemma 5.4 and Lemma 5.2 $\mathtt{d}(\mathtt{F}(\mathtt{t}_1^j,\ldots,\mathtt{t}_n^j),\mathtt{N}) = \mathtt{d}(\mathtt{t}_k^j,\mathtt{N}) = \mathtt{d}(\mathtt{X},\mathtt{N})$. Moreover, since $\mathtt{t}_k^j \prec \mathtt{t}_k$, we have $\mathtt{X} \in \mathrm{FV}(\mathtt{t}_k)$, by Lemma 5.4 and Lemma 5.3 the conclusion follows □

5 Revisiting the examples

We now come back to the examples of Section 3. We give some possible translations in eLPL. Besides showing how the translations works, this should clarify the way such programs can be typed.
Consider the function definition $\mathtt{d_C}$ for the program `Coer` typable as $\mathtt{Coer} : \mathbb{N} \multimap \S\mathbb{N}$. The expected type derivation can be translated in eLPL as

$$\begin{array}{lcl} \mathtt{Coer(s(X))} & = & \mathtt{let\ Coer(X)\ be\ \S z\ in\ \S s(z)}\ , \\ \mathtt{Coer(0)} & = & \mathtt{\S 0} \end{array}$$

Consider the function definition $\mathtt{d_A}$ for the program `Add` typable as $\mathtt{Add} : \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}$. The expected type derivation can be translated in eLPL as

$$\begin{array}{lcl} \mathtt{Add(s(X),Y)} & = & \mathtt{let\ Add(X,Y)\ be\ \S z\ in\ \S s(z)}\ , \\ \mathtt{Add(0,Y)} & = & \mathtt{Y} \end{array}$$

Consider the function definition $\mathtt{d_M}$ for the program `Mul` typable as $\mathtt{Mul} : \mathbb{N} \multimap !\mathbb{N} \multimap \S\S\mathbb{N}$. The expected type derivation can be translated in eLPL as

$$\begin{array}{l} \mathtt{Mul(s(X),Y)} = \mathtt{let\ Y\ be\ !r\ in\ let\ Mul(X,!r)\ be\ \S z\ in\ \S Add(r,z)}\ , \\ \mathtt{Mul\ (0,Y)} = \mathtt{\S\S 0} \end{array}$$

Consider the function definition $\mathtt{d_P}$ for the program `Map` typable as $\mathtt{Map} : !(\mathbb{N} \multimap \S\S\mathbb{N}) \multimap \mathbb{L} \multimap \S\S\S\mathbb{L}$. The expected type derivation can be translated in eLPL as

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

Then the program

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

can be translated in eLPL as

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

Analogously, consider the function definition $\mathtt{d_R}$ for the program `Foldr` typable as $\mathtt{Foldr} : !(\mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}) \multimap \S\S\mathbb{N} \multimap \mathbb{L} \multimap \S\S\mathbb{N}$. The expected type derivation can be translated in eLPL as

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

Then the program

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

can be translated in eLPL as

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

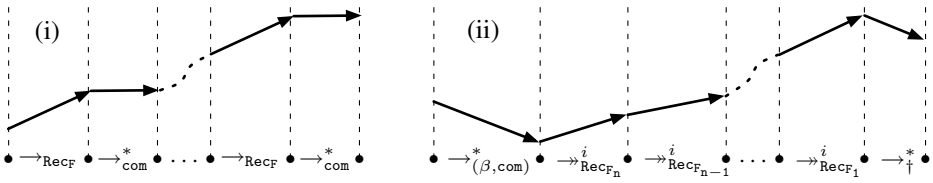

Figure 1: Term size variations at fixed depth i by a standard reduction step and round at same depth.

Finally consider the program Tadd summing trees, typable as $\mathtt{Tadd} : \mathbb{T} \multimap \mathbb{T} \multimap \S\mathbb{T}$. With a type coercion function for $\mathbb{T}$ data type, the expected type derivation can be translated in eLPL as:

$$\begin{array}{lcl} \mathtt{Tadd}(\mathtt{node}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\mathtt{node}(\mathtt{X}',\mathtt{Y}',\mathtt{Z}')) & = & \mathtt{let}\ \mathtt{Add}(\mathtt{X},\mathtt{X}')\ \mathtt{be}\ \S\mathtt{x}\ \mathtt{in} \\ & & \mathtt{let}\ \mathtt{Tadd}(\mathtt{Y},\mathtt{Y}')\ \mathtt{be}\ \S\mathtt{y}\ \mathtt{in}\ \mathtt{let}\ \mathtt{Tadd}(\mathtt{Z},\mathtt{Z}')\mathtt{be}\ \S\mathtt{z}\ \mathtt{in}\ \S\mathtt{node}(\mathtt{x},\mathtt{y},\mathtt{z}) \\ \mathtt{Tadd}(\mathtt{X},\epsilon) & = & \mathtt{Coer}(\mathtt{X}) \\ \mathtt{Tadd}(\epsilon,\mathtt{X}) & = & \mathtt{Coer}(\mathtt{X}) \end{array}$$

6 Polynomial Time soundness

We here show that LPL programs can be effectively evaluated in polynomial time in the size of the input with the degree of the polynomial, as usual, given by the structure of program. We show this by working on the translation of programs in eLPL. For simplicity we consider LPL without basic functions, but the proof can be easily extended to the whole LPL. We consider now only well typed and translated eLPL programs.
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 soundness proof for the depth-by-depth strategy in LLL relies on the following key facts:

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 make the size at depth i grow. So, by introducing some new measures on

matching arguments, we show that this growth is polynomial in the size of the initial term, i.e. Lemma 29.
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 $\twoheadrightarrow^i_{\mathtt{Rec}_{\mathtt{F_j}}}$ 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 2. 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 30.
Finally, in order to adapt fact 4 to our framework, we introduce the notion of *potential degree*. This correspond to 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 32.
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 3, will follow.

6.1 Preliminary properties

For a given a program $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}\ in\ \mathtt{M}$ we need 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 show simple properties about eLPL term depths.

We have built eLPL in such a way to keep depth properties similar to those of light logics. One of the most important ones, directly adapted from [Ter01], is:

Lemma 7. *Let* $\mathtt{M}$ *be a* com*-normal at depth* $i \geq 0$*. If* $\mathtt{M} \rightarrow^*_\dagger \mathtt{M}'$ *at depth* i *then for* $j > i$ *we have* $|\mathtt{M}'|_j \leq |\mathtt{M}|_j \times |\mathtt{M}|_i$*. Moreover it does not create redexes at depth* $j \leq i$*.*

The following properties hold for eLPL well-typed terms.

Lemma 8. *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* $\mathtt{N}$*,* $\mathtt{d}(\mathtt{x}_i, \mathtt{N}) = 1$*.*

Proof. Easy by depth definition and by typing rules. □ □

The last point above does not hold in general for non matching arguments, but it holds for translated function definition cases. Furthermore every redex is at the same depth of its arguments:

Lemma 9. *1. Let* $(\lambda \mathtt{x.M})\mathtt{N}$ *be a* β*-redex in* $\mathtt{p}$*. Then* $\mathtt{d}((\lambda \mathtt{x.M})\mathtt{N},\mathtt{p}) = \mathtt{d}(\lambda \mathtt{x.M},\mathtt{p}) = \mathtt{d}(\mathtt{N},\mathtt{p})$.

2. Let $\mathtt{let\ M} = \dagger\mathtt{x\ in\ N}$ *be a* $\dagger$*-redex in* $\mathtt{p}$*. Then* $\mathtt{d}(\mathtt{let\ M} = \dagger\mathtt{x\ in\ N},\mathtt{p}) = \mathtt{d}(\mathtt{M},\mathtt{p}) = \mathtt{d}(\mathtt{N},\mathtt{p})$.

3. Let $\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_n)$ *be a* `rec`*-redex in* $\mathtt{p}$*. Then* $\mathtt{d}(\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_n),\mathtt{p}) = \mathtt{d}(\mathtt{M}_1,\mathtt{p}) = \cdots = \mathtt{d}(\mathtt{M}_n,\mathtt{p})$.

Proof. Trivial. □ □

We now give some interesting properties of the reduction. Like in [Ter01], we have:

Lemma 10. *At depth* $i \geq 0$*, a* $\rightarrow_\dagger$ *reduction for* $\dagger \in \{!,\S\}$ *or a* $\rightarrow_\beta$ *reduction strictly decreases the term size at depth* i*, and the number of* `com`*-reductions in* $\mathtt{M} \rightarrow^*_{\mathtt{com}} \mathtt{M}'$ *is bounded by* $(|\mathtt{M}|_i)^2$.

Proof. Easy, by using Lemma 8.2 and Lemma 5.3. □

Corollary 11. *At depth* $i \geq 0$ *the number of* $(\beta,\dagger)$*-reductions in* $\mathtt{M} \rightarrow^*_{\beta,\dagger} \mathtt{M}'$ *is bounded by* $|\mathtt{M}|_i$.

6.2 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 `rec`-reduction steps at depth j. For that we will first introduce an intermediary notion:.

Definition 14 (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* `let` *or of a variable at depth* j*. This is inductively defined as:*

$$\|\dagger\mathtt{M}'\|_{\mathtt{0}} = \mathtt{0} \qquad \|\dagger\mathtt{M}'\|_{\mathtt{j+1}} = \|\mathtt{M}'\|_{\mathtt{j}}$$

$$\|\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_\mathtt{n})\|_{\mathtt{0}} = \|\mathtt{y}\mathtt{M}_1\cdots\mathtt{M}_\mathtt{n}\|_{\mathtt{0}} = \mathtt{0} \qquad \|\mathtt{c}\mathtt{M}_1\cdots\mathtt{M}_\mathtt{n}\|_{\mathtt{0}} = \sum_{\mathtt{i=1}}^{\mathtt{n}} \|\mathtt{M}_\mathtt{i}\|_{\mathtt{0}} + \mathtt{1}$$

$$\|\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_\mathtt{n})\|_{\mathtt{j+1}} = \|\mathtt{y}\mathtt{M}_1\cdots\mathtt{M}_\mathtt{n}\|_{\mathtt{j+1}} = \|\mathtt{c}\mathtt{M}_1\cdots\mathtt{M}_\mathtt{n}\|_{\mathtt{j+1}} = \sum_{\mathtt{i=1}}^{\mathtt{n}} \|\mathtt{M}_\mathtt{i}\|_{\mathtt{j+1}}$$

$$\begin{aligned}
\|(\mathtt{let\ N}_1\ \mathtt{be}\ \dagger\mathtt{x\ in\ N}_2)\mathtt{M}_1\cdots\mathtt{M}_n\|_0 &= \|\mathtt{N}_2\|_0 + \sum_{i=1}^{n} \|\mathtt{M}_i\|_0 \\
\|(\mathtt{let\ N}_1\ \mathtt{be}\ \dagger\mathtt{x\ in\ N}_2)\mathtt{M}_1\cdots\mathtt{M}_n\|_{j+1} &= \|\mathtt{N}_1\|_{j+1} + \|\mathtt{N}_2\|_{j+1} + \sum_{i=1}^{n} \|\mathtt{M}_i\|_{j+1} \\
\|(\lambda\mathtt{x.M}')\mathtt{M}_1\cdots\mathtt{M}_n\|_j &= \|\mathtt{M}'\|_j + \sum_{i=1}^{n} \|\mathtt{M}_i\|_j
\end{aligned}$$

The external constructor size measure enjoys the following remarkable property.

Lemma 12. *Let* $\mathtt{F}(\mathtt{t}_1,\ldots,\mathtt{t}_n) = \mathtt{N}$ *be a safe definition case and let the recursive calls of* F *in* N *be* $\mathtt{F}(\mathtt{t}^1_1,..,\mathtt{t}^1_n),\ldots,\mathtt{F}(\mathtt{t}^m_1,..,\mathtt{t}^m_n)$*. 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 6 and Definition 4. □ □

Lemma 13. *If* $\Gamma;\Delta\vdash\mathtt{M}:\dagger A$ *and* M *is* $(\beta,\mathtt{com})$*-normal at depth* 0*, then* $\|\mathtt{M}\|_0 = 0$*.*

Proof. By induction on the structure of M. □ □

Lemma 14. *If* $\Gamma;\Delta\vdash\mathtt{M}:A$ *and* 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. By hypothesis we have $\mathtt{M} = \mathtt{M}_1\{\mathtt{F}(\sigma(\mathtt{t}_1),\ldots,\sigma(\mathtt{t}_n))\}\to_{\mathtt{Rec_F}}\mathtt{M}_1\{\sigma(\mathtt{N})\} = \mathtt{M}'$. By Definition 14 we have $\|\mathtt{M}\|_0 = \|\mathtt{M}_1\{\mathtt{x}\}\|_0+\|\mathtt{F}(\sigma(\mathtt{t}_1),\ldots,\sigma(\mathtt{t}_n))\|_0 = \|\mathtt{M}_1\{\mathtt{x}\}\|_0$ for a fresh variable x. Analogously it is easy to verify that $\|\mathtt{M}'\|_0 = \|\mathtt{M}_1\{\mathtt{x}\}\|_0+\|\sigma(\mathtt{N})\|_0$ for a fresh variable x. By typing constraints we have $\Gamma_1;\Delta_1\vdash\sigma(\mathtt{N}):\S B$ for some Γ_1,Δ_1 and B. So by Lemma 13 we have $\|\sigma(\mathtt{N})\|_0 = 0$ and so the conclusion follows. □ □

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

Definition 15. *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* F *in* M*. It is inductively defined as:*

$$\mathrm{SA}^{\mathtt{F}}_0(\dagger\mathtt{M}') = 0 \quad \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}+1}(\dagger\mathtt{M}') = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}')$$

$$\mathrm{SA}^{\mathtt{F}}_0(\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_n)) = \sum_{i=1}^{n}\mathrm{SA}^{\mathtt{F}}_0(\mathtt{M}_i) + \sum_{r\in\mathcal{R}(\mathtt{F})}\|\mathtt{M}_r\|_0$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}+1}(\mathtt{F}(\mathtt{M}_1,\ldots,\mathtt{M}_n)) = \sum_{i=1}^{n}\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}+1}(\mathtt{M}_i)$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{s}\mathtt{M}_1\cdots\mathtt{M}_n) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{G}(\mathtt{M}_1,\ldots,\mathtt{M}_n)) = \sum_{i=1}^{n}\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}_i) \qquad \textit{if } \mathtt{s}\in\{\mathtt{y},\mathtt{c}\}$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}((\lambda\mathtt{x}.\mathtt{M}')\mathtt{M}_1\cdots\mathtt{M}_n) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}') + \sum_{i=1}^{n}\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}_i)$$

$$\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}((\mathtt{let}\ \mathtt{N}_1\ \mathtt{be}\ \dagger\mathtt{x}\ \mathtt{in}\ \mathtt{N}_2)\mathtt{M}_1\cdots\mathtt{M}_n) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{N}_1) + \mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{N}_2) + \sum_{i=1}^{n}\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}_i)$$

Lemma 15. *We have* $\|\mathtt{M}\|_0 + \sum\{\mathrm{SA}^{\mathtt{G}}_0(\mathtt{M}) \mid \mathtt{G}\in\mathtt{M}\} \leqslant |\mathtt{M}|_0$*. Moreover for* $i\geq 0$ *we have* $\sum\{\mathrm{SA}^{\mathtt{G}}_{\mathtt{i}}(\mathtt{M}) \mid \mathtt{G}\in\mathtt{M}\} \leq |\mathtt{M}|_i$*.*

Proof. By induction on $i\geq 0$ and on the structure of M. □ □

Remark that this lemma gives a bound for all the recursive functions defined in a program, but often we use it to give a bound only for one function symbol : $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M}) \leqslant |\mathtt{M}|_i$.

Lemma 16. *We have for all $j \geq 0$:* $\mathrm{SA}^{\mathtt{F}}_{\mathtt{j}}(\mathtt{M}) = \sum_{r \in \mathcal{R}(\mathtt{F})} \{\|\mathtt{M}_r\|_0 \mid \mathtt{F}(\mathtt{M}_1, \ldots, \mathtt{M}_k) \in_j \mathtt{M}\}$.

Proof. By induction on the structure of $\mathtt{M}$ and on j. □ □

Now comes the key Lemma for which we have introduced $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M})$:

Lemma 17. *If $\mathtt{M}$ is a well typed (β, com)-normal term and $\mathtt{M} \to_{\mathtt{Rec_F}} \mathtt{M}'$ at depth i then* $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M}') < \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M})$.

Proof. Let $\mathtt{M}\{\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))\} \to_{\mathtt{Rec_F}} \mathtt{M}\{\sigma(\mathtt{N})\} = \mathtt{M}'$. We proceed by induction on the shape of the context $\mathtt{M}\{\}$. The most interesting case is when $\mathtt{M}\{\} = \mathtt{F}(\mathtt{M}_1, \ldots, \mathtt{C}\{\}, \ldots, \mathtt{M}_n)$ where $\mathtt{C}\{\}$ is the k-th argument of $\mathtt{F}$. We proceed by induction on depth i, by using Definition 15 and Lemma 14. □ □

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

Lemma 18. *Reducing a* Rec*-redex at depth i cannot introduce a β-redex at depth i.*

Proof. Let $\mathtt{M}\{\mathtt{F}(\sigma(\overrightarrow{\mathtt{t}}))\} \to_{\mathtt{Rec_F}} \mathtt{M}\{\sigma(\mathtt{N})\}$ by applying a definition case of the shape $\mathtt{F}(\overrightarrow{\mathtt{t}}) = \mathtt{N}$. By typing constraints we have both $\Gamma, \Delta \vdash \mathtt{F}(\overrightarrow{\mathtt{t}}) : \S B$ and $\Gamma, \Delta \vdash \sigma(\mathtt{N}) : \S B$, so in particular by Lemma 5.1, $\mathtt{N}$ cannot be a term of the shape $\lambda \mathtt{x}.\mathtt{P}$, so it cannot create itself a β-redex. Since $\mathtt{N}$ by definition is normal, the only way to create a β-redex at depth i is by the substitution σ to a variable $\mathtt{X}$ in a subterm of $\mathtt{N}$ of the shape $\mathtt{XQ}$ at depth i. But in particular $\mathtt{X} \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta)$, so by typing constraints $\mathtt{X}$ should have type $\S B$ and so it cannot be the case. So, the conclusion follows. □ □

Corollary 19. *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.*

Lemma 20. *Reducing a* com*-redex at depth i cannot introduce a* Rec*-redex at depth i.*

Proof. Easy, by the shape of the reduct in a com-reduction. □ □

From the above lemma we have directly the following:

Corollary 21. *If $\mathtt{M}$ is $\mathtt{Rec_F}$-normal at depth i and $\mathtt{M} \to_{\mathtt{com}} \mathtt{M}'$ at depth i then $\mathtt{M}'$ is $\mathtt{Rec_F}$-normal at depth i.*

In order to show that the number of Rec-reductions is bound, we now need to consider the behaviour of Rec-reductions on Rec-redexes of other function symbols.

Lemma 22. *Let $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}$ in $\mathtt{M}$ be a program. Then, reducing $\mathtt{Rec}_{\mathtt{F}_i}$ redexes in $\mathtt{M}$ at depth d can introduce only $\mathtt{F}_\mathtt{j}$ for $j \leq i$ function symbols and the occurrences introduced are at a depth inferior or equal to $d + \max\{\mathtt{d}(\mathtt{N}_\mathtt{j}) \mid \mathtt{N}_\mathtt{j}$ body in a definition case of $\mathtt{F}_\mathtt{i}\}$.*

Proof. Easy as substitution are done at depth d of terms with depth bounded by $\max\{\mathtt{d}(\mathtt{N}_\mathtt{j}) \mid \mathtt{N}_\mathtt{j}$ body in a definition case of $\mathtt{F}_\mathtt{i}\}$. □ □

Lemma 23. *Let $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}$ in $\mathtt{M}$ be a program with $\mathtt{M}$ being (β, com)-normal. Then, reducing a $\mathtt{Rec}_{\mathtt{F}_i}$ redex in $\mathtt{M}$ at depth d cannot introduce a $\mathtt{rec}_{\mathtt{F}_j}$-redex for $n \geq j > i$ at depth d.*

Proof. Easy: blocked function symbols remain blocked by Lemma 22 and Lemma 14. □ □

From Lemma 17 we directly have the following:

Corollary 24 ($\mathtt{Rec_F}$-reductions bound)**.** *Let* $\mathtt{M}$ *be* $(\beta, \mathtt{com})$*-normal at depth* i*. If* $\mathtt{M} \to^{k}_{\mathtt{Rec_F}} \mathtt{M}'$ *at depth* i *then* $k \leq \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M})$.

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

Lemma 25 (Size lemma)**.** *If* $\mathtt{M} \to_{\mathtt{Rec_F}} \mathtt{M}'$ *at depth* i *then for all* $j \geq i$ *we have* $|\mathtt{M}'|_j \leq |\mathtt{M}|_j + K_{\mathtt{F}}$.

Proof. Let $\mathtt{M} = \mathtt{M}_1\{\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))\} \to_{\mathtt{Rec_F}} \mathtt{M}_1\{\sigma(\mathtt{N})\} = \mathtt{M}'$. 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

$$\begin{aligned} |\sigma(\mathtt{N})|_0 &= |\mathtt{N}|_0 + \sum_{\mathtt{X} \in_0 \mathrm{FO}(\mathtt{N})} (|\sigma(\mathtt{X})|_0 - 1) \\ |\mathtt{F}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_n))|_0 &= 1 + \sum_{k=1}^{n} |\mathtt{t}_k|_0 + \sum_{\mathtt{X} \in_0 \mathrm{FO}(\overrightarrow{\mathtt{t}})} (|\sigma(\mathtt{X})|_0 - 1) \end{aligned}$$

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 $\mathtt{N}$ 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 $\mathtt{N}$ at depth 0. □ □

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

Lemma 26. *Let* $\mathtt{M} \to_{com} \mathtt{M}'$ *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* $\mathtt{F}$ *and every* $i \leq \mathtt{d}(\mathtt{M})$.

Proof. Easy by induction on the shape of $\mathtt{M}$. □ □

We precisely describe the reduction strategy *at a fixed depth* used to bound the number of reduction steps of well typed eLPL programs, in the following definitions:

Definition 16 (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* $\mathtt{R} \twoheadrightarrow^{i}_{\mathtt{Rec_F}} \mathtt{R}'$*, is a sequence of reductions at depth* i *of the shape:*

$$\mathtt{R} \to_{\mathtt{Rec_F}} \mathtt{T} \to^{*}_{\mathtt{com}} \mathtt{R}_1 \to_{\mathtt{Rec_F}} \mathtt{T}_1 \to^{*}_{\mathtt{com}} \cdots \to_{\mathtt{Rec_F}} \mathtt{T}_k \to^{*}_{\mathtt{com}} \mathtt{R}_k \equiv \mathtt{R}'$$

  such that every $\mathtt{R}_i$ *is* com*-normal, and* $\mathtt{R}_k$ *does not contain any* $\mathtt{Rec_F}$*-redex..*

- *a* standard reduction round *at depth* i*, denoted* $\mathtt{M} \Rightarrow^{i} \mathtt{M}'$*, is the following sequence of reductions at depth* i*:*

$$\mathtt{M} \to^{*}_{(\beta,\mathtt{com})} \mathtt{M}_0 \twoheadrightarrow^{i}_{\mathtt{Rec_{F_n}}} \mathtt{M}_1 \twoheadrightarrow^{i}_{\mathtt{Rec_{F_{n-1}}}} \cdots \twoheadrightarrow^{i}_{\mathtt{Rec_{F_1}}} \mathtt{M}_n \to^{*}_{\dagger} \mathtt{M}'$$

  such that $\mathtt{M}_0$ *is* $(\beta, \mathtt{com})$*-normal and* $\mathtt{M}'$ *is* $\dagger$*-normal.*

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 we need to prove that all the reductions are finite. First we need the following.

Lemma 27. *A sequence of reductions $\rightarrow_{\mathtt{Rec_F}}\rightarrow^{*}_{\mathtt{com}}$ at depth i cannot introduce a β-redex at depth i.*

Proof. As in the proof of Lemma 18 we know that the substitution σ of the $\mathtt{Rec_F}$ step is applied to variables $\mathtt{X}$ which have as type either a data-type $\mathbb{D}$, or a type $\S B$. By Definition 12 of the $\mathtt{com}$-reduction rules, only the rule $(\mathtt{com}_2)$ can introduce a beta-redex in the situation where $\mathtt{V}$ is of the form $\mathtt{V} = \lambda y.\mathtt{W}$. But a subterm of the form $(\mathtt{let\ U\ be}\ \dagger\mathtt{x\ in}\ \lambda \mathtt{y}.\mathtt{W})$ cannot be created by a $(\mathtt{com}_1)$, $(\mathtt{com}_2)$ or $(\mathtt{com}_3)$ reduction step and cannot also be created by the $(\mathtt{Rec_F})$ step either, because its type is not $\mathbb{D}$ or $\S B$. □ □

From the above lemma we have directly the following:

Corollary 28. *If $\mathtt{M}$ is β-normal at depth i and $\mathtt{M} \rightarrow_{\mathtt{Rec_F}}\rightarrow^{*}_{\mathtt{com}} \mathtt{M}'$ 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 29 (Bound on standard reduction step at depth i)**.** *Let $\mathtt{M}$ be $(\beta, \mathtt{com})$-normal at depth i. If $\mathtt{M} \twoheadrightarrow^{i}_{\mathtt{Rec_F}} \mathtt{M}'$ then $\mathtt{M}'$ is $(\beta, \mathtt{com}, \mathtt{Rec_F})$-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. Let $\mathtt{M}$ be $(\beta, \mathtt{com})$-normal at depth i such that $\mathtt{M} \twoheadrightarrow^{i}_{\mathtt{Rec_F}} \mathtt{M}'$. By definition we have

$$\mathtt{M} \equiv \mathtt{R} \rightarrow_{\mathtt{Rec_F}} \mathtt{T}_1 \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_1 \ \rightarrow_{\mathtt{Rec_F}} \mathtt{T}_2 \rightarrow^{*}_{\mathtt{com}} \cdots \ \rightarrow_{\mathtt{Rec_F}} \mathtt{T}_k \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_k \equiv \mathtt{M}'$$

By Lemma 17 since $\mathtt{R}$ is $(\beta, \mathtt{com})$-normal and $\mathtt{R} \rightarrow_{\mathtt{Rec_F}} \mathtt{T}_1$ at depth i we have $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{R}) > \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{T}_1)$. By Lemma 10 we have $\mathtt{T}_1 \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_1$ normalizes at depth i in a number of steps bounded by $(|\mathtt{T}_1|_{\mathtt{i}})^2$. Moreover, by Lemma 26.iii we have $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{T}_1) = \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{R}_1)$. So if $\mathtt{R}$ is $(\beta, \mathtt{com})$-normal and $\mathtt{R} \rightarrow_{\mathtt{Rec_F}} \mathtt{T}_1 \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_1$ at depth i then $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{R}) > \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{R}_1)$.
Moreover by Corollary 28 and by the definition we have $\mathtt{R}_1$ is $(\beta, \mathtt{com})$-normal. So we can repeatedly apply the hypothesis. This shows that $k \leq \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M})$ and that $\mathtt{M}'$ is $(\beta, \mathtt{com}, \mathtt{Rec_F})$-normal at depth i.
Now we prove the size bound on such a standard reduction step at depth i. By lemma 26.ii the size of a term at depth i is unchanged by $\mathtt{com}$-reductions, so we just need to consider $\mathtt{Rec_F}$-reductions. By Lemma 25 if $\mathtt{N} \rightarrow_{\mathtt{Rec_F}} \mathtt{N}'$ at depth i then $|\mathtt{N}'|_i \leq |\mathtt{N}|_i + K_{\mathtt{F}}$ so since we have at most $\mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M})$ steps of $\mathtt{Rec_F}$-reduction, by Lemma 15 we can conclude

$$|\mathtt{M}'|_i \leq |\mathtt{M}|_i + K_{\mathtt{F}} \times \mathrm{SA}^{\mathtt{F}}_{\mathtt{i}}(\mathtt{M}) \leq |\mathtt{M}|_i \times (K_{\mathtt{F}} + 1)$$

Now we can complete the proof for the bound on the total number of steps. As stressed above, the number of $\mathtt{Rec_F}$-reductions is bounded by $|\mathtt{M}|_i$, so we just need to count the number of $\mathtt{com}$-reductions. By Corollary 10 we have that for each reduction $\mathtt{T}_m \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_m$ for $1 \leq m \leq k$ the number of step is bounded by $(|\mathtt{T}_{\mathtt{m}}|_{\mathtt{i}})^2$. Since by reasoning as above we have $|\mathtt{T}_{\mathtt{m}}| \leq |\mathtt{M}|_{\mathtt{i}} + \mathtt{K_F} \times \mathtt{m} \leq |\mathtt{M}|_{\mathtt{i}} \times (\mathtt{K_F} + \mathtt{1})$, we can conclude that for each m the number of $\mathtt{com}$-reduction step in $\mathtt{T}_m \rightarrow^{*}_{\mathtt{com}} \mathtt{R}_m$ for $1 \leq m \leq k$ is bounded by $(|\mathtt{M}|_i)^2 \times (K_{\mathtt{F}} + 1)^2$. So, we globally have at most $k + k \times ((|\mathtt{M}|_i)^2 \times (K_{\mathtt{F}} + 1)^2) \leq 2 \times (|\mathtt{M}|_i)^3 \times (K_{\mathtt{F}} + 1)^2$ steps. □ □

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 2 (Bound on standard round at depth i)**.** *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ *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. Let $\mathtt{M} \Rightarrow^i_k \mathtt{M}'$ be a standard reduction round at depth $i \geq 0$. First we prove that it terminates in normal form. By definition we have

$$\mathtt{M} \to^*_{(\beta,\mathtt{com})} \mathtt{M}_0 \twoheadrightarrow^i_{\mathtt{Rec}_{\mathtt{F}_n}} \mathtt{M}_1 \twoheadrightarrow^i_{\mathtt{Rec}_{\mathtt{F}_{n-1}}} \cdots \twoheadrightarrow^i_{\mathtt{Rec}_{\mathtt{F}_1}} \mathtt{M}_n \to^*_{\dagger} \mathtt{M}'$$

By Corollary 11 and Lemma 10 we have $\mathtt{M}_0$ is $(\beta, \mathtt{com})$-normal at depth i and the number of steps is bounded by $(|\mathtt{M}|_i)^2$. So by lemma 29 all the $\mathtt{M}_j$ for $1 \leq j \leq n$ are also $(\beta, \mathtt{com})$-normal at depth i.
Note that a standard reduction round is composed by a fixed number of standard reduction steps given by the number of function definitions. Consider $\mathtt{M}_j \twoheadrightarrow^i_{\mathtt{Rec}_{\mathtt{F}_{n-j}}} \mathtt{M}_{j+1}$ for $0 \leq j \leq n-1$.
By lemma 29 we have $\mathtt{M}_{j+1}$ is $\mathtt{Rec}_{\mathtt{F}_{n-j}}$-normal at depth i. Moreover by Lemma 22 such a standard reduction step can introduce only $\mathtt{F}_{n-u}$ function symbols for $n-j \geq n-u \geq 1$ at depth i and by Lemma 23 and Lemma 20 no new $\mathtt{Rec}_{\mathtt{F}_{n-u}}$ redex for $n \geq n-u > n-j$ can be created at depth i by Rec-reduction or com-reduction. Hence, in particular we have that if $\mathtt{M}_j$ is $\mathtt{Rec}_{\mathtt{F}_{n-u}}$-normal at depth i for all $n \geq n-u > n-j$ then $\mathtt{M}_{j+1}$ is $\mathtt{Rec}_{\mathtt{F}_{n-u}}$-normal at depth i for all $n \geq n-u \geq n-j$
So, we have $\mathtt{M}_n$ is normal for all recursive function symbols. Finally by Lemma 7 we have $\mathtt{M}'$ is normal for all reductions at depth i.
Now we prove the bound on $|\mathtt{M}'|_i$ and on the entire number k of reduction steps in a standard reduction round at depth i. Clearly, by Corollary 11 and Lemma 10 we have $k \leq (|\mathtt{M}|_i)^2 + R + C + |\mathtt{M}_n|_i$ where C is the number of com-reduction steps, and R is the sum of $\mathtt{Rec}_{\mathtt{F}_j}$-reduction steps for $n \geq j \geq 1$.
By lemma 17 and lemma 29 we have

$$R = \sum_{j=0}^{n-1} \mathrm{SA}_{\mathtt{i}}^{\mathtt{F}_{n-j}}(\mathtt{M}_j) \leq \sum_{j=0}^{n-1} |\mathtt{M}_j|_i \leq \sum_{j=0}^{n-1} |\mathtt{M}_0|_i \times (K+1)^j \leq |\mathtt{M}_0|_i \times (K+1)^n$$

By Lemma 10 we have $C = \sum_{j=0}^{n-1} 2 \times (|\mathtt{M}_j|_i)^3 \times (K+1)^2$

$$\leq 2 \times (K+1)^2 \times (|\mathtt{M}_0|_i)^3 \times (K+1)^{3n} \leq 2 \times (|\mathtt{M}_0|_i)^3 \times (K+1)^{3n+2}$$

By lemma 10 and lemma 7 we obtain the bound on the number of reduction steps in a standard reduction round at depth i:

$$k \leq (|\mathtt{M}|_i)^2 + R + C + |\mathtt{M}_n|_i \leq (|\mathtt{M}|_i)^2 + |\mathtt{M}_0|_i \times (K+1)^n + 2 \times (|\mathtt{M}_0|_i)^3 \times (K+1)^{3n+2} + |\mathtt{M}_0|_i \times (K+1)^n \leq 3 \times (|\mathtt{M}|_i)^3 \times (K+1)^{3n+2}$$

There remains to bound the size of $\mathtt{M}'$ at depth i. By lemma 10, lemma 26.ii and lemma 29 we have

$$|\mathtt{M}_0|_i \leq |\mathtt{M}|_i \text{ and } |\mathtt{M}_n|_i \leq |\mathtt{M}_0|_i \times (K+1)^n \text{ and } |\mathtt{M}'|_i \leq |\mathtt{M}_n|_i$$

So at depth i we have the bound $|\mathtt{M}'|_i \leq |\mathtt{M}|_i \times (K+1)^n$. □ □

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 30 (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* $\mathtt{M} \Rightarrow_k^i \mathtt{M'}$ *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. Let $\mathtt{M} \Rightarrow_k^i \mathtt{M'}$ be a standard reduction round at depth i as

$$\mathtt{M} \rightarrow^*_{(\beta,\mathtt{com})} \mathtt{M}_0 \twoheadrightarrow^i_{\mathtt{Rec_{F_n}}} \mathtt{M}_1 \twoheadrightarrow^i_{\mathtt{Rec_{F_{n-1}}}} \cdots \twoheadrightarrow^i_{\mathtt{Rec_{F_1}}} \mathtt{M}_n \rightarrow^*_{\dagger} \mathtt{M'}$$

Following the theorem 2 proof, by lemma 10 we have $|\mathtt{M}_0|_i \leq |\mathtt{M}|_i$ and $|\mathtt{M}_0|_{i+1} \leq |\mathtt{M}|_{i+1}$. By lemma 29 for $n-1 \geq u \geq 1$ we have $|\mathtt{M}_{u+1}|_{i+1} \leq |\mathtt{M}_u|_{i+1} + |\mathtt{M}_u|_i \times K$. Then

$$|\mathtt{M}_n|_{i+1} \leq |\mathtt{M}_0|_{i+1} + \sum_{v=0}^{n-1} |\mathtt{M}_0|_i \times K \times (K+1)^v \leq |\mathtt{M}_0|_{i+1} + |\mathtt{M}_0|_i \times (K+1)^n$$

By lemma 26 and lemma 7 we have $|\mathtt{M'}|_{i+1} \leq |\mathtt{M}_n|_{i+1} \times |\mathtt{M}_n|_i$. Then

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

□ □

It then implies:

Corollary 31. *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* $|\mathtt{M'}| \leq 2(|\mathtt{M}|)^2 \times (K+1)^{2n+1}$.

6.3 Bound on a program normalization

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

Definition 17 (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:* $\mathtt{M} \Rightarrow^0 \Rightarrow^1 \cdots \Rightarrow^d \mathtt{M'}$
When we need to stress the number k *of reduction rounds in a standard reduction we simply write it as* $\mathtt{M} \Rrightarrow_k \mathtt{M'}$.

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

Definition 18 (Potential Depth)**.** *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d_{F_1}}, \ldots, \mathtt{d_{F_n}}$ *in* $\mathtt{M}$ *be a program,* $\mathtt{N}$ *be an occurrence of a subterm in* $\mathtt{M}$*. The* potential depth *of* $\mathtt{N}$ *in* $\mathtt{p}$*, denoted* $\mathtt{pt_d}(\mathtt{N}, \mathtt{p})$ *is defined as*

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

The potential degree *of* $\mathtt{p}$*, denoted* $\mathtt{pt_d}(\mathtt{p})$*, 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 32. *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* $m < \mathtt{pt_d(p)}$.

Proof. Every standard reduction can be summarized as follows

$$\begin{array}{lllllllllll}
\mathtt{M} & \to^*_{\beta,com} & \mathtt{M}^0_{k_0} & \twoheadrightarrow^0_{\mathtt{Rec}_{\mathtt{F_n}}} & \mathtt{M}^0_{k_1} & \twoheadrightarrow^0_{\mathtt{Rec}_{\mathtt{F_{n-1}}}} & \cdots & \twoheadrightarrow^0_{\mathtt{Rec}_{\mathtt{F}_1}} & \mathtt{M}^0_{k_n} & \to^*_{\dagger} & \mathtt{M}_0 \\
\mathtt{M}_0 & \to^*_{\beta,com} & \mathtt{M}^1_0 & \twoheadrightarrow^1_{\mathtt{Rec}_{\mathtt{F_n}}} & \mathtt{M}^1_1 & \twoheadrightarrow^1_{\mathtt{Rec}_{\mathtt{F_{n-1}}}} & \cdots & \twoheadrightarrow^1_{\mathtt{Rec}_{\mathtt{F}_1}} & \mathtt{M}^1_n & \to^*_{\dagger} & \mathtt{M}_1 \\
\vdots & & & & & & & & & & \\
\mathtt{M}_{m-1} & \to^*_{\beta,com} & \mathtt{M}^m_0 & \twoheadrightarrow^m_{\mathtt{Rec}_{\mathtt{F_n}}} & \mathtt{M}^m_1 & \twoheadrightarrow^m_{\mathtt{Rec}_{\mathtt{F_{n-1}}}} & \cdots & \twoheadrightarrow^m_{\mathtt{Rec}_{\mathtt{F}_1}} & \mathtt{M}^m_n & \to^*_{\dagger} & \mathtt{M}_m
\end{array}$$

We prove by induction on i: $\forall j \leq n, \forall k \leq m$:

$$\mathtt{d}(\mathtt{F}_{n-i}, \mathtt{M}^j_k) \leq \mathtt{d(p)} + \sum_{m=0}^{i} \max_o \{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m}(\overrightarrow{\mathtt{t}}) = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F_{n-m}}}\}$$

Then the conclusion follows easily since β, com and † reductions cannot increase the depth. Base case $i = 0$ follows from the fact that β, com and † reductions preserve the depth of terms, so in particular of function symbols, and from the fact that $\mathtt{F_n}$ can introduce only other $\mathtt{F_n}$ redexes at the same depth while it cannot be introduced by other `rec`-redexes (Lemma 22). So in particular for each j, k we have $\mathtt{d}(\mathtt{F_n}, \mathtt{M}^{\mathtt{j}}_{\mathtt{k}}) \leq \mathtt{d(p)}$. Now consider the case $i = l + 1 < n$. By induction hypothesis for every $l_1 \leq l$ we have $\forall j \leq n, \forall k \leq m$:

$$\mathtt{d}(\mathtt{F}_{n-l_1}, \mathtt{M}^j_k) \leq \mathtt{d(p)} + \sum_{m=0}^{l_1} \max_o \{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m}(\overrightarrow{\mathtt{t}}) = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F_{n-m}}}\}$$

Moreover, we now also need to proceed by induction on the number of reduction steps (the number of $\mathtt{M}^{\mathtt{j}}_{\mathtt{k}}$). Base case is trivial, so consider the case the reduction is of length $s+1$, in fact we proceed by simultaneous induction on j, k. Suppose we are performing a reduction $\mathtt{M}^{\mathtt{j_1}}_{\mathtt{k_1}} \to \mathtt{M}^{\mathtt{j'_1}}_{\mathtt{k'_1}}$ for some j_1 and k_1. The case of a β or † reduction follows by induction hypothesis and from the fact that such reductions preserve the depth of subterms.

So we can consider the case of a reduction of the shape $\mathtt{M}^{\mathtt{j_1}}_{\mathtt{k_1}} \twoheadrightarrow^{j_1}_{\mathtt{Rec}_{\mathtt{F_h}}} \mathtt{M}^{\mathtt{j_1}}_{\mathtt{k_1+1}}$. Let $\mathtt{F}'_h$ be the occurrence of the function symbols $\mathtt{F}_h$ involved in the reduction. Then, by Lemma **??** all occurrences of every functions symbols $\mathtt{F}_r$ but $\mathtt{F}'_h$ are such that $\mathtt{d}(\mathtt{F}_r, \mathtt{M}^{j_1}_{k_1}) = \mathtt{d}(\mathtt{F}_r, \mathtt{M}^{\mathtt{j_1}}_{\mathtt{k_1+1}})$. So we just need to consider the new occurrences of function symbols introduced by the $\mathtt{rec}_{\mathtt{F'_h}}$ reduction. By induction hypothesis we also have

$$\mathtt{d}(\mathtt{F_{n-(1+1)}}, \mathtt{M}^{\mathtt{j_1}}_{\mathtt{k_1}}) \leq \mathtt{d(p)} + \sum_{m=0}^{l+1} \max_o \{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m} = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F_{n-m}}}\}$$

We distinguish three cases. If $h \leq n - l$ then by Lemma 22 the reduction cannot introduce new occurrences of the function symbol $\mathtt{F}_{n-(l+1)}$, so the conclusion follows by induction hypothesis.

If $h = n-(l+1)$ then the reduction can introduce occurrences of the $\mathtt{F}_{n-(l+1)}$ function symbol just at depth equal to j_1, so we can conclude by induction hypothesis:

$$j_1 \leq \mathtt{d(p)} + \sum_{m=0}^{l+1} \max_o \{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m} = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F_{n-m}}}\}$$

If $n \geq h > n-(l+1)$ then by Lemma 22 the reduction can introduce occurrences of the $\mathtt{F}_{n-(l+1)}$ function symbol just at depth less than $j_1 + \max_o\{\mathtt{d}(\mathtt{N}^o_h)\}$. Note that $h = n - l'$ with $l' \leq l$, so by induction hypothesis we have

$$\mathtt{d}(\mathtt{F}_h, \mathtt{M}^{j_1}_{k_1}) \leq \mathtt{d}(\mathtt{p}) + \sum_{m=0}^{h} \max_o\{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m} = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F}_{n-m}}\}$$

So we can conclude by: $\mathtt{d}(\mathtt{F}_{n-(l+1)}, \mathtt{M}^j_{k+1})$

$$\begin{array}{ll} \leq & \mathtt{d}(\mathtt{p}) + \sum_{m=0}^{h} \max_o\{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m} = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F}_{n-m}}\} + \max_o\{\mathtt{d}(\mathtt{N}^o_h)\} \\ \leq & \mathtt{d}(\mathtt{p}) + \sum_{m=0}^{l+1} \max_o\{\mathtt{d}(\mathtt{N}^o_{n-m}) \mid \mathtt{F}_{n-m} = \mathtt{N}^o_{n-m} \in \mathtt{d}_{\mathtt{F}_{n-m}}\} \quad \square \end{array}$$

$\square$

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 3. *Let* $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ *be a translated program and* $d = \mathtt{pt}_{\mathtt{d}}(\mathtt{p})$ *be its potential degree. Let* $\mathtt{M} \Rrightarrow_k \mathtt{M}'$ *be a standard reduction of the shape*

$$\mathtt{M} \equiv \mathtt{M}^0 \Rightarrow^0 \mathtt{M}^1 \Rightarrow^1 \mathtt{M}^2 \cdots \Rightarrow^{d-1} \mathtt{M}^d \Rightarrow^d \mathtt{M}^{d+1} \equiv \mathtt{M}'$$

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. Let $\mathtt{p} = \mathtt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots, \mathtt{d}_{\mathtt{F}_n}\ \mathtt{in}\ \mathtt{M}$ be a translated program of potential degree $d = \mathtt{pt}_{\mathtt{d}}(\mathtt{p})$. Let $\mathtt{M} \Rrightarrow_k \mathtt{M}'$ be a standard reduction of the shape as in the theorem hypothesis. By Theorem 2 we have that each standard reduction round at depth i reaches a normal form at depth i, moreover this does not affect terms at depth lower than i. So, since by Lemma 32 the maximal number of standard reduction rounds is bound by d then we have that $\mathtt{M}'$ is in normal form. Now we prove the bound on the size of $\mathtt{M}'$. We proceed by induction on $0 \leq i \leq d$ and we prove

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

By Corollary 31 we have $|\mathtt{M}^{i+1}| \leq 2 \times (|\mathtt{M}^i|)^2 \times (K+1)^{2n+1}$. So the conclusion follows for $i = 0$. By induction hypothesis we have:

$$\begin{array}{l} |\mathtt{M}^{i+1}| \leq 2 \times [2^{(2^i-1)} \times (|\mathtt{M}|)^{2^i} \times (K+1)^{(2^i-1)(2n+1)}]^2 \times (K+1)^{2n+1} \\ \qquad\qquad \leq 2^{(2^{i+1}-1)} \times (|\mathtt{M}|)^{2^{i+1}} \times (K+1)^{(2^{i+1}-1)(2n+1)} \end{array}$$

and so the conclusion. Now the bound is obtained by considering $i = d$ and obviously that $2^{d+1} - 1 < 2^{d+1}$. Now we prove the bound on the number k of reduction steps in a standard reduction. Consider $\mathtt{M}^i \Rightarrow^i_{k_i} \mathtt{M}^{i+1}$, by Theorem 2 we have

$$k_i \leq 3.(|\mathtt{M}^i|_i)^3 \times (K+1)^{3n+2} \leq 3.(|\mathtt{M}^i|)^3 \times (K+1)^{3n+2}$$

so, by Corollary 31 and observation above we have

$$k_i \leq 3.[2^{(2^i)-1} \times (|\mathtt{M}|)^{2^i} \times (K+1)^{(2^i-1)(2n+1)}]^3 \times (K+1)^{3n+2}$$
$$\leq 3.\Big(2^3 \times |\mathtt{M}|^3 \times (K+1)^{3(2n+1)}\Big)^{2^i}$$

For all $0 \leq i \leq d$, $k_i \leq k_d$ so we can conclude

$$k = \sum_{i=0}^{d} k_i \leq 3.(d+1) \times \Big(2^3 \times |\mathtt{M}|^3 \times (K+1)^{3(2n+1)}\Big)^{2^d}$$

□

7 PTIME completeness

The proof that LPL is complete for polynomial time functions is rather standard: we show that we can simulate any polynomial time (one tape) Turing machine in the language.
In LPL we can represent all the polynomials in $\mathbb{N}[X]$ For simplicity we just show the following.

Lemma 33. *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 $K \times x^{2^k}$.*

Proof. As shown in Section 3 we have a term for multiplication typable as $\mathtt{Mul} : \mathbb{N} \multimap !\mathbb{N} \multimap \S\S\mathbb{N}$. By functoriality of $\S$ we have also the typing $\mathtt{Mul} : \S\mathbb{N} \multimap \S(!\mathbb{N} \multimap \S\S\mathbb{N})$. Similarly to what happens in DLAL [BT09] there is for any type A a coercion: $\mathtt{C} : \mathbb{N} \multimap \S((!\mathbb{N} \multimap A) \multimap \S A)$ we have also $\mathtt{Mul} : \S\mathbb{N} \multimap \mathbb{N} \multimap \S\S\S\mathbb{N}$. So, by using the $\mathtt{Coer} : \mathbb{N} \multimap \S\mathbb{N}$ program we get: $\mathbb{N} \multimap \mathbb{N} \multimap \S^3\mathbb{N}$. By the obtained term we can build a term for squaring: $\mathtt{square} : !\mathbb{N} \multimap \S^4\mathbb{N}$. So by using $\mathtt{C}$ again we get $\mathbb{N} \multimap \S^5\mathbb{N}$. By composing it k times and using another multiplication with the constant K, we get the program announced. □ □

We consider a polytime Turing machine $\mathcal{M}$ with witness time polynomial P, n states, and a 3 symbols alphabet (0,1 and blank). for encoding the configurations we use the following 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 $\mathbb{L}_3$ type corresponds to the right part of the tape, starting with scanned symbol.

Lemma 34. *For any transition function δ, there exists a* LPL *basic function* $\underline{\mathtt{conf2conf}}$: $\mathtt{config} \multimap \mathtt{config}$ *representing the corresponding action on configurations.*

Proof. (Sketch) The definition of $\underline{\mathtt{conf2conf}}$ has 2 cases for each line of the table of transition function of the Turing Machine (one if the head reaches the end of the tape, and one if it doesn't).
Let us just take an example :
If $\delta(\underline{\mathtt{q}}, \underline{\mathtt{s}}) = (\underline{\mathtt{q}'}, \mathtt{left}, \underline{\mathtt{s}'})$ where $\underline{q}$ and $\underline{q'}$ are states, $\underline{s}$ is a symbol and $\underline{s'}$ a symbol to write , then we have 2 case definitions lines:

$$\underline{\mathtt{conf2conf}}\ \mathtt{p}\ (\mathtt{p}\ (\mathtt{x} :: \mathtt{l}_1')\ (\underline{\mathtt{s}} :: \mathtt{l}_2'))\ \underline{\mathtt{q}} = \mathtt{p}\ (\mathtt{p}\ \mathtt{l}_1'\ (\mathtt{x} :: \underline{\mathtt{s}'} :: \mathtt{l}_2'))\ \underline{\mathtt{q}'}$$

$$\underline{\texttt{conf2conf}}\ \texttt{p}\ (\texttt{p}\ \underline{\texttt{nil}}\ (\underline{\texttt{s}} :: \texttt{l}_2')) \ \underline{\texttt{q}} = \texttt{p}\ (\texttt{p}\ \underline{\texttt{nil}}\ (\underline{\texttt{b}} :: \underline{\texttt{s}'} :: \texttt{l}_2'))\ \underline{\texttt{q}'}$$

where $\underline{\texttt{b}}$ is the blank symbol (if the head reaches the left part of the tape, the next scanned symbol is blank). Here note that $\underline{\texttt{s}}$, $\underline{\texttt{q}}$, $\underline{\texttt{s}'}$, $\underline{\texttt{q}'}$ and $\underline{\texttt{b}}$ are values (actually constructors of finite types), $\underline{\texttt{nil}}$ is a value also, while $\texttt{x}, \texttt{l}_1', \texttt{l}_2'$ are pattern variables. Finally note that $\underline{\texttt{conf2conf}}$ has type $\texttt{config} \multimap \texttt{config}$. □ □

Note that $\underline{\texttt{conf2conf}}$ satisfies the conditions of a basic function. We will use also an iterator, defined by:

$$\begin{array}{lcl} \texttt{Iter}\ (\texttt{sx})\ \texttt{f}\ \texttt{base} & = & \texttt{f}\ \texttt{Iter}\ \texttt{x}\ \texttt{f}\ \texttt{base} \\ \texttt{Iter}\ \texttt{0}\ \texttt{f}\ \texttt{base} & = & \texttt{base} \end{array}$$

$\texttt{Iter}$ is a recursive function with type : $\mathbb{N} \multimap !(A \multimap A) \multimap \S A \multimap \S A$. We will use $A = \texttt{Config}$.

Theorem 4 (Ptime Completeness)**.** *There exists an* LPL *program of type* $\mathbb{L}_2 \multimap \S^j \mathbb{L}_2$*, representing the same function as* $\mathcal{M}$*.*

Proof. By Lemma 33 we can represent in LPL a polynomial boundary P, with type $\mathbb{N} \multimap \S^m \mathbb{N}$, for some m. Denote it $\texttt{t} : \mathbb{N} \multimap \S^m \mathbb{N}$.
It is also easy to define a function: $\texttt{Length} : \mathbb{L}_2 \multimap \S\mathbb{N}$. We can also define: $\texttt{Init} = \lambda \texttt{w}.\texttt{p}\ (\texttt{p}\ \texttt{nil}\ \texttt{w})$. The projections $\underline{\Pi_i} : \mathbb{D}_1 \times \mathbb{D}_2 \multimap \mathbb{D}$ are easily defined as case-definitions. The desired program is then obtained by iterating $\underline{\texttt{conf2conf}}\ (\texttt{t}\ (\texttt{Length}\ \texttt{x}))$ times, from the base $(\texttt{Init}\ \texttt{w})$. So it is:
$\texttt{letRec}\ \underline{\Pi_1}$ ... $\underline{\Pi_2}$ ... $\underline{\texttt{conf2conf}}$ ... (previous basic functions)
$\texttt{letRec}\ \texttt{Iter}$ ... $\texttt{Length}$ ... (previous recursive functions)
$\texttt{in}$
$\lambda \texttt{w}.\underline{\Pi_1 \Pi_2}(\texttt{Iter}(\texttt{t}(\texttt{Length}\ \texttt{w})))\ \underline{\texttt{conf2conf}}$

To type it we need to use some coercions : $\texttt{coerc} : \mathbb{L}_2 \multimap \S^{\texttt{m}+2}\mathbb{L}_2$.
The typing gives us the following term for the body $\texttt{M}$ of the program with type $\mathbb{L}_2 \multimap \S^{m+3}\mathbb{L}_2$:

$$\begin{array}{l} \lambda \texttt{w}.\texttt{let}\ \texttt{w}\ \texttt{be}\ !\texttt{w}'\ \texttt{in}\ \S(\texttt{let}\ (\texttt{t}(\texttt{Length}\ \texttt{w}'))\ \texttt{be}\ \S^{\texttt{m}+1}\texttt{n}\ \texttt{in}\ \texttt{let}\ (\texttt{coerc}\ \texttt{w}')\ \texttt{be}\ \S^{\texttt{m}+2}\texttt{w}'' \\ \quad \texttt{in}\ \S^{\texttt{m}+1}(\ \texttt{let}\ \texttt{Iter}\ \texttt{n}\ (!\underline{\texttt{conf2conf}})\ (\S\texttt{w}'')\ \texttt{be}\ \S\texttt{final}\ \texttt{in}\ \S(\underline{\Pi_2}(\underline{\Pi_1}\ \texttt{final})))) \end{array}$$

□ □

8 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 (DLAL).
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 [Hof99], 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. We think that, following the lines of what has been done for DLAL, our PTIME proof can be turned into a polynomial time strong normalization proof for LPL, i.e. the polynomial bound could hold for the length of any reduction sequence of the LPL program.
- Comparing the intensional expressiveness of our LPL with the one of other languages for PTIME, notably Safe Recursion on Notation [BC92] and LFPL [Hof99].

Work along those lines is underway.

References

[ABT06] V. Atassi, P. Baillot, and K. Terui. Verification of ptime reducibility for system F terms via dual light affine logic. In *CSL '06*, volume 4207 of *LNCS*. Springer, 2006.

[Bai02] P. Baillot. Checking polynomial time complexity with types. In *Proceedings of the 2nd IFIP International Conference on TCS*, pages 370–382, 2002.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2):97–110, 1992.

[BM04] P. Baillot and V. Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *FoSSaCS'04*, volume 2987 of *LNCS*, pages 27–41. Springer, 2004.

[BMM01] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On lexicographic termination ordering with space bound certifications. In *PSI 2001*, volume 2244 of *LNCS*, pages 482–493. Springer, 2001.

[BNS00] S. Bellantoni, K. H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *APAL*, 104:17–30, 2000.

[BT09] P. Baillot and K. Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 207(1):41–62, 2009.

[CW00] K. Crary and S. Weirich. Resource bound certification. In *ACM POPL'00*, pages 184–198, 2000.

[DL09] U Dal Lago. The geometry of linear higher-order recursion. *ACM TOCL*, 10(2), 2009.

[DLMR03] U. Dal Lago, S. Martini, and L. Roversi. Higher order linear ramified recurrence. In *TYPES'03*, volume 3085 of *LNCS*, pages 178–193. Springer, 2003.

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

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[GR08] M. Gaboardi and S. Ronchi Della Rocca. Type inference for a polynomial lambda calculus. In *TYPES'08*, volume 5497 of *LNCS*, pages 136–152. Springer, 2008.

[GRDR07] M. Gaboardi and S. Ronchi Della Rocca. A soft type assignment system for λ-calculus. In *CSL '07*, volume 4646 of *LNCS*, pages 253–267. Springer, 2007.

[GSS92] J. Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *TCS*, 97(1):1–66, 1992.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap usage for first-order functional programs. In *ACM POPL'03*, New Orleans, LA, USA, 2003.

[Hof99] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In LICS 99, pages 464–473, 1999.

[HP99] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *ACM ICFP'99*, pages 70–81, 1999.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *TCS*, 318(1-2):163–180, 2004.

[Lei91] D. Leivant. A foundational delineation of computational feasibility. In Albert R. Meyer, editor, LICS '91, pages 2–11. IEEE Computer Society, 1991.

[LM93] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of polytime. In (TLCA '93), volume 664 of *LNCS*, pages 274–288. Springer, 1993.

[Mar03] J.-Y. Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183(1):2–18, 2003.

[MM00] J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In LPAR '00, volume 1955 of *LNCS*, pages 25–42. Springer, 2000.

[MP08] J.-Y. Marion and R. Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In *ACM-PPDP'08*, pages 79–88, 2008.

[MP09] J.-Y. Marion and R. Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM TOCL*, 10(4), 2009.

[Ter01] K. Terui. Light affine lambda calculus and polytime strong normalization. In (LICS '01), pages 209–220. IEEE Computer Society, 2001.