Light Types for Polynomial Time Computation in Lambda Calculus

Patrick Baillot [1]

LIPN - UMR 7030 CNRS - Université Paris 13, F-93430 Villetaneuse, France

Kazushige Terui [2]

Research Institute for Mathematical Sciences, Kyoto University, Japan

Abstract

We present a polymorphic type system for lambda calculus ensuring that well-typed programs can be executed in polynomial time: *dual light affine logic* (DLAL). DLAL has a simple type language with a linear and an intuitionistic type arrow, and one modality. It corresponds to a fragment of light affine logic (LAL). We show that contrarily to LAL, DLAL ensures good properties on lambda-terms (and not only on proof-nets): subject reduction is satisfied and a well-typed term admits a polynomial bound on the length of any of its beta reduction sequences. We also give a translation of LAL into DLAL and deduce from it that all polynomial time functions can be represented in DLAL.

Key words: linear logic, light linear logic, lambda calculus, type system, implicit computational complexity, polynomial time complexity.

1 Introduction

Implicit Computational Complexity. Functional languages like ML assist the programmer with prevention of such errors as run-time type errors, thanks

Email addresses: patrick.baillot@ens-lyon.fr (Patrick Baillot), terui@kurims.kyoto-u.ac.jp (Kazushige Terui).

[1] Partially supported by projects NOCoST (ANR, JC05_43380), GEOCAL ACI *Nouvelles interfaces des mathématiques* and CRISS ACI *Sécurité informatique*.

[2] Partially supported by Grant-in-Aid for Scientific Research, MEXT, Japan. Also employed by National Institute of Informatics, Tokyo and by CNRS at Laboratoire d'Informatique de Paris Nord when this paper was written.

to automatic type inference. One could wish to extend this setting to verification of quantitative properties, such as time or space complexity bounds (see for instance [25]). We think that progress on such issues can follow from advances in the topic of *Implicit Computational Complexity*, the field that studies calculi and languages with intrinsic complexity-theoretic properties. In particular some lines of research have explored recursion-based approaches [28,12,24] and approaches based on linear logic to control the complexity of programs [20,26].

Here we are interested in *light linear or affine logic* (resp. **LLL**, **LAL**) [2,20], a logical system designed from linear logic and which characterizes polynomial time computation. By the Curry-Howard correspondence, proofs in this logic can be used as programs. A nice aspect of this system with respect to other approaches is the fact that it includes higher order types as well as polymorphism (in the sense of system **F**); relations with the recursion-based approach have been studied in [33]. Moreover this system naturally extends to a consistent naive set theory, in which one can reason about polynomial time concepts. In particular the provably total functions of that set theory are exactly the polynomial time functions [20,38]. Finally **LLL** and related systems like *elementary linear logic* have also been studied through the approaches of *denotational semantics* [27,5], *geometry of interaction* [17,7] and *realizability* [18].

Programming directly in **LAL** through the Curry-Howard correspondence is quite delicate, in particular because the language has two modalities and is thus quite complicated. A natural alternative idea is to use ordinary lambda calculus as source language and **LAL** as a type system. Then one would like to use a type inference algorithm to perform **LAL** typing automatically (as for elementary linear logic [15,16,14,10]). However following this line one has to face several issues:

(1) β reduction is problematic: subject reduction fails and no polynomial bound holds on the number of β reduction steps;
(2) type inference, though decidable in the propositional case [6], is difficult.

Problem (1) can be avoided by *compiling* **LAL**-typed lambda-terms into an intermediate language with a more fine-grained decomposition of computation: *proof-nets* (see [2,32]) or terms from *light affine lambda calculus* [37,39]. In that case subject-reduction and polytime soundness are recovered, but the price paid is the necessity to deal with a more complicated calculus.

Modal types and restrictions. Now let us recast the situation in a larger perspective. Modal type systems have actually been used extensively for typing lambda calculus (see [19] for a brief survey). It has often been noted that the modalities together with the functional arrow induce a delicate behaviour:

for instance adding them to simple types can break down such properties as principal typing or subject-reduction (see *e.g.* the discussions in [34,23,13]), and make type inference more difficult. However in several cases it is sufficient in order to overcome some of these problems to restrict one's attention to a subclass of the type language, for instance where the modality $\Box$ is used only in combination with arrows, in types $\Box A \to B$; in this situation types of the form $A \to \Box B$ for instance are not considered. This is systematized in Girard's embedding of intuitionistic logic in linear logic by $(A \to B)^* = !A^* \multimap B^*$.

Application to light affine logic. The present work fits in this perspective of taming a modal type system in order to ensure good properties. Here as we said the motivation comes from implicit computational complexity.

In order to overcome the problems (1) and (2), we propose to apply to **LAL** the approach mentioned before of restricting to a subset of the type language ensuring good properties. Concretely we replace the ! modality by two notions of arrows: a linear one ($\multimap$) and an intuitionistic one ($\Rightarrow$). This is in the line of the work of Plotkin ([35]; see also [21]). Accordingly we have two kinds of contexts as in dual intuitionistic linear logic of Barber and Plotkin [11]. Thus we call the resulting system *dual light affine logic*, **DLAL**.

An important point is that even though the new type language is smaller than the previous one, this system is actually computationally as general as **LAL**: indeed we provide a generic encoding of **LAL** into **DLAL**, which is based on the simple idea of translating $(!A)^\bullet = \forall\alpha.(A^\bullet \Rightarrow \alpha) \multimap \alpha$ [35].

Moreover **DLAL** keeps the good properties of **LAL**: the representable functions on binary lists are exactly the polynomial time functions. Finally and more importantly it enjoys new properties:

- subject-reduction w.r.t. β reduction;
- Ptime soundness w.r.t. β reduction;
- efficient type inference: it was shown in [3,4] that type inference in **DLAL** for system **F** lambda-terms can be performed in polynomial time.

Contributions. Besides giving detailed proofs of the results in [8], in particular of the strong polytime bound, the present paper also brings some new contributions:

- a result showing that **DLAL** really corresponds to a fragment of **LAL**;
- a simplified account of coercions for data types;
- a generic encoding of **LAL** into **DLAL**, which shows that **DLAL** is computationally as expressive as **LAL**;
- a discussion about iteration in **DLAL** and, as an example, a **DLAL** program for insertion sort.

Outline. The paper is organized as follows. We first recall some background on light affine logic in Section 2 and define **DLAL** in Section 3. Then in Section 4 we state the main properties of **DLAL** and discuss the use of iteration on examples. Section 5 is devoted to an embedding of **DLAL** into **LAL**, Section 6 to the simulation theorem and its corollaries: the subject reduction theorem and the polynomial time strong normalization theorem. Finally in Section 7 we give a translation of **LAL** into **DLAL** and prove the FP completeness theorem.

2 Background on light affine logic

Notations. Given a lambda-term t we denote by $FV(t)$ the set of its free variables. Given a variable x we denote by $no(x,t)$ the number of occurrences of x in t. We denote by $|t|$ the *size* of t, i.e., the number of nodes in the term formation tree of t. The notation $\longrightarrow$ will stand for β reduction on lambda-terms.

2.1 Light affine logic

The language $\mathcal{L}_{LAL}$ of **LAL** types is given by:

$$A, B ::= \alpha \mid A \multimap B \mid !A \mid \S A \mid \forall \alpha.A.$$

We omit the connective $\otimes$ which is definable. We will write † instead of either ! or §.

Light affine logic is a logic for polynomial time computation in the proofs-as-programs approach to computing. It controls the number of reduction (or cut-elimination) steps of a proof-program using two ideas:

(i) stratification,

(ii) control on duplication.

Stratification means that the proof-program is divided into levels and that the execution preserves this organization. It is managed by the two modalities (also called *exponentials*) ! and §.

Duplication is controlled as in linear logic: an argument can be duplicated only if it has undergone a !-rule (hence has a type of the form $!A$). What is specific to **LAL** with respect to linear logic is the condition under which one

$$\frac{}{x : A \vdash x : A}\ (\mathrm{Id})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ (\multimap \mathrm{i}) \qquad \frac{\Gamma_1 \vdash t : A \multimap B \quad \Gamma_2 \vdash u : A}{\Gamma_1, \Gamma_2 \vdash tu : B}\ (\multimap \mathrm{e})$$

$$\frac{\Gamma_1 \vdash t : A}{\Gamma_1, \Gamma_2 \vdash t : A}\ (\mathrm{Weak}) \qquad \frac{x_1 : !A, x_2 : !A, \Gamma \vdash t : B}{x : !A, \Gamma \vdash t[x/x_1, x/x_2] : B}\ (\mathrm{Cntr})$$

$$\frac{\Gamma, \Delta \vdash t : A}{!\Gamma, \S\Delta \vdash t : \S A}\ (\S \mathrm{i}) \qquad \frac{\Gamma_1 \vdash u : \S A \quad \Gamma_2, x : \S A \vdash t : B}{\Gamma_1, \Gamma_2 \vdash t[u/x] : B}\ (\S \mathrm{e})$$

$$\frac{x : B \vdash t : A}{x : !B \vdash t : !A}\ (! \mathrm{i}) \qquad \frac{\Gamma_1 \vdash u : !A \quad \Gamma_2, x : !A \vdash t : B}{\Gamma_1, \Gamma_2 \vdash t[u/x] : B}\ (! \mathrm{e})$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall \alpha.A}\ (\forall \mathrm{i})\ (*) \qquad \frac{\Gamma \vdash t : \forall \alpha.A}{\Gamma \vdash t : A[B/\alpha]}\ (\forall \mathrm{e})$$

Fig. 1. Natural deduction system for **LAL**

can apply a !-rule to a proof-program: it should have at most one occurrence of free variable (rule (! i) of Figure 1).

We present the system as a natural deduction type assignment system for lambda calculus: see Figure 1. We have:

- for ($\forall$ i): (*) α does not appear free in Γ.
- the (! i) rule can also be applied to a judgement of the form $\vdash u : A$ (u has no free variable).

The notation Γ, Δ will be used for *environments* attributing formulas to variables. If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ then $\dagger\Gamma$ denotes $x_1 : \dagger A_1, \ldots, x_n : \dagger A_n$. In the sequel we write $\Gamma \vdash_{LAL} t : A$ for a judgement derivable in **LAL**.

The *depth* of a derivation $\mathcal{D}$ is the maximal number of (! i) and (§ i) rules in a branch of $\mathcal{D}$. We denote by $|\mathcal{D}|$ the *size* of $\mathcal{D}$ defined as its number of judgments.

Now, light affine logic enjoys the following property:

Theorem 1 ([20,1]) *Given an* **LAL** *proof $\mathcal{D}$ with depth d, its normal form can be computed in $O(|\mathcal{D}|^{2^{d+1}})$ steps.*

This statement refers to reduction performed either on proof-nets [20,2] or on light affine lambda terms [37,39]. If the depth d is fixed and the size of $\mathcal{D}$ might vary (for instance when applying a fixed term to binary integers) then the result can be computed in polynomial steps.

Moreover we have:

Theorem 2 ([20,2]) *If a function $f : \{0,1\}^* \to \{0,1\}^*$ is computable in polynomial time, then there is a proof in* **LAL** *that represents f.*

Here the depth of the proof depends on the degree of the polynomial. See also Theorem 33.

2.2 LAL and β reduction

It was shown in [39] that light affine lambda calculus admits polynomial step strong normalization: the bound of Theorem 1 holds on the length of *any* reduction sequence of light affine lambda terms. However, this property is not true for **LAL**-typed plain lambda terms and β reduction: indeed [2] gives a family of **LAL**-typed terms (with a fixed depth) such that there exists a reduction sequence of exponential length. So the reduction of **LAL**-typed lambda terms is not *strongly* poly-step (when counting the number of β reduction steps).

We stress here with an example the fact that normalization of **LAL**-typed lambda terms is not even *weakly* poly-step nor polytime: there exists a family of **LAL**-typed terms (with fixed depth) such that the computation of their normal form on a Turing machine (using any strategy) will take exponential time. Note that this is however not in contradiction with the statement of Theorem 1, because the data structures considered here and in Theorem 1 are different: lambda calculus in the forthcoming example and proof-nets (or light affine lambda terms) in the theorem mentioned.

Let us define the example. First, observe that the following judgments are derivable:

$$y_i :!A \multimap !A \multimap !A \vdash_{LAL} \lambda x.y_i x x :!A \multimap !A,$$
$$z :!A \vdash_{LAL} z :!A.$$

From this it is easy to check that the following is derivable:

$$y_1 :!A \multimap !A \multimap !A, \ldots, y_n :!A \multimap !A \multimap !A, z :!A \vdash_{LAL} (\lambda x.y_1 x x)(\cdots(\lambda x.y_n x x)z \cdots) :!A.$$

Using (§ i) and (Cntr) we finally get:

$$y :!(!A \multimap !A \multimap !A), z :!!A \vdash_{LAL} (\lambda x.yxx)^n z : \S!A.$$

Denote by t_n the term $(\lambda x.yxx)^n z$ and by u_n its normal form. We have $u_n = y u_{n-1} u_{n-1}$, so $|u_n| = O(2^n)$, whereas $|t_n| = O(n)$: the size of u_n is exponential

in the size of t_n. Hence computing u_n from t_n on a Turing machine will take at least exponential time (if the result is written on the tape as a lambda-term).

It should be noted though that even if u_n is of exponential size, it nevertheless has a type derivation of size $O(n)$. To see this, note that we have $z : !A, y : !A \multimap !A \multimap !A \vdash_{LAL} yzz : !A$. Now make n copies of it and compose them by (! e); each time (! e) is applied, the term size is doubled. Finally, by applying (§ i) and (Cntr) as before, we obtain a linear size derivation for $y : !(!A \multimap !A \multimap !A), z : !!A \vdash_{LAL} u_n : §!A$.

2.3 Discussion

The counter-example of the previous section illustrates a mismatch between lambda calculus and light affine logic. It can be ascribed to the fact that the (! e) rule on lambda calculus not only introduces sharing but also causes duplication. As Asperti neatly points out [1], "while every datum of type $!A$ is eventually sharable, not all of them are actually duplicable." The above yzz gives a typical example. While it is of type $!A$ and thus sharable, it should not be duplicable, as it contains more than one free variable occurrence. The (! e) rule on lambda calculus, however, neglects this delicate distinction, and actually causes duplication.

Light affine lambda calculus remedies this by carefully designing the syntax so that the (! e) rule allows sharing but not duplication. As a result, it offers the properties of subject reduction with respect to **LAL** and polynomial time strong normalization [39]. However it is not as simple as lambda calculus; in particular it includes new constructions !(.), §(.) and let (.) be (.) in (.) corresponding to the management of boxes in proof nets.

The solution we propose here is more drastic: we simply do not allow the (! e) rule to be applied to a term of type $!A$. This is achieved by removing judgments of the form $\Gamma \vdash t : !A$. As a consequence, we also remove types of the form $A \multimap !B$. Bang ! is used only in the form $!A \multimap B$, which we consider as a primitive connective $A \Rightarrow B$. Note that it does not cause much loss of expressiveness in practice, since the standard decomposition of intuitionistic logic by linear logic does not use types of the form $A \multimap !B$.

3 Dual light affine logic

The system we propose does not use the ! connective but distinguishes two kinds of function spaces (linear and non-linear) as in [35] (see also [21]). Our

approach is also analogous to that of dual intuitionistic linear logic of Barber and Plotkin [11] in that it distinguishes two kinds of environments (linear and non-linear). Thus we call our system dual light affine logic (**DLAL**). We will see that it corresponds in fact to a well-behaved fragment of **LAL**.

The language $\mathcal{L}_{DLAL}$ of **DLAL** types is given by:

$$A, B ::= \alpha \mid A \multimap B \mid A \Rightarrow B \mid \S A \mid \forall \alpha.A.$$

Let us now define Π_1 and Σ_1 types. The class of Π_1 (resp. Σ_1) types is the least subset of **DLAL** types satisfying:

- any atomic type α is Π_1 (resp. Σ_1),
- if A is Σ_1 (resp. Π_1) and B is Π_1 (resp. Σ_1), then $A \multimap B$ and $A \Rightarrow B$ are Π_1 (resp. Σ_1),
- if A is Π_1 (resp. Σ_1) then $\S A$ is Π_1 (resp. Σ_1),
- if A is Π_1 then so is $\forall \alpha.A$.

There is an unsurprising translation $(.)^\star$ from **DLAL** to **LAL** given by:

- $(A \Rightarrow B)^\star = !A^\star \multimap B^\star$,
- $(.)^\star$ commutes to the other connectives.

Let $\mathcal{L}_{DLAL^\star}$ denote the image of $\mathcal{L}_{DLAL}$ by $(.)^\star$, and $(.)^- : \mathcal{L}_{DLAL^\star} \to \mathcal{L}_{DLAL}$ stand for the converse map of $(.)^\star$.

For **DLAL** typing we will handle judgements of the form $\Gamma; \Delta \vdash t : C$. The intended meaning is that variables in Δ are (affine) linear, that is to say that they have at most one occurrence in the term, while variables in Γ are non-linear. We give the typing rules as a natural deduction system: see Figure 2. We have:

- for ($\forall$ i): (*) α does not appear free in Γ, Δ.
- in the ($\Rightarrow$ e) rule the r.h.s. premise can also be of the form $; \vdash u : A$ (u has no free variable).

In the rest of the paper we will write $\Gamma; \Delta \vdash_{DLAL} t : A$ for a judgement derivable in **DLAL**.

Observe that the contraction rule (Cntr) is used only on variables on the l.h.s. of the semi-column. It is then straightforward to check the following statement:

Lemma 3 *If $\Gamma; \Delta \vdash_{DLAL} t : A$ then the set $FV(t)$ is included in the variables of $\Gamma \cup \Delta$, and if $x \in \Delta$ then we have $no(x, t) \leqslant 1$.*

We can make the following remarks on **DLAL** rules:

$$\frac{}{;x : A \vdash x : A}\ (\text{Id})$$

$$\frac{\Gamma;\Delta, x : A \vdash t : B}{\Gamma;\Delta \vdash \lambda x.t : A \multimap B}\ (\multimap \text{i}) \qquad \frac{\Gamma_1;\Delta_1 \vdash t : A \multimap B \quad \Gamma_2;\Delta_2 \vdash u : A}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash tu : B}\ (\multimap \text{e})$$

$$\frac{\Gamma, x : A;\Delta \vdash t : B}{\Gamma;\Delta \vdash \lambda x.t : A \Rightarrow B}\ (\Rightarrow \text{i}) \qquad \frac{\Gamma;\Delta \vdash t : A \Rightarrow B \quad ;z : C \vdash u : A}{\Gamma, z : C;\Delta \vdash tu : B}\ (\Rightarrow \text{e})$$

$$\frac{\Gamma_1;\Delta_1 \vdash t : A}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash t : A}\ (\text{Weak}) \qquad \frac{x_1 : A, x_2 : A, \Gamma;\Delta \vdash t : B}{x : A, \Gamma;\Delta \vdash t[x/x_1, x/x_2] : B}\ (\text{Cntr})$$

$$\frac{;\Gamma, \Delta \vdash t : A}{\Gamma;\S\Delta \vdash t : \S A}\ (\S \text{ i}) \qquad \frac{\Gamma_1;\Delta_1 \vdash u : \S A \quad \Gamma_2; x : \S A, \Delta_2 \vdash t : B}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash t[u/x] : B}\ (\S \text{ e})$$

$$\frac{\Gamma;\Delta \vdash t : A}{\Gamma;\Delta \vdash t : \forall\alpha.A}\ (\forall \text{ i})\ (*) \qquad \frac{\Gamma;\Delta \vdash t : \forall\alpha.A}{\Gamma;\Delta \vdash t : A[B/\alpha]}\ (\forall \text{ e})$$

Fig. 2. Natural deduction system for **DLAL**

- Initially the variables are linear (rule (Id)); to convert a linear variable into a non-linear one we can use the (§ i) rule. Note that it adds a § to the type of the result and that the variables that remain linear get a § type too.
- the ($\multimap$ i) (resp. ($\Rightarrow$ i)) rule corresponds to abstraction on a linear variable (resp. non-linear variable);
- observe ($\Rightarrow$ e): a term of type $A \Rightarrow B$ can only be applied to a term u with at most one occurrence of free variable.

Note that the only rules which correspond to substitutions in the term are (Cntr) and (§ e): in (Cntr) only a variable is substituted and in (§ e) substitution is performed on a linear variable. Combined with Lemma 3 this ensures the following important property:

Proposition 4 *If a derivation $\mathcal{D}$ has conclusion $\Gamma;\Delta \vdash_{DLAL} t : A$ then we have $|t| \leq |\mathcal{D}|$.*

This proposition shows that the mismatch between lambda calculus and **LAL** illustrated in the previous section is resolved with **DLAL**.

One can observe that the rules of **DLAL** are obtained from the rules of **LAL** via the $(.)^\star$ translation. As a consequence, **DLAL** can be considered as a subsystem of **LAL**. Let us make this point precise.

Definition 5 *Let $\mathcal{L}_{DLAL^+}$ be the set $\mathcal{L}_{DLAL^\star} \cup \{!A : A \in \mathcal{L}_{DLAL^\star}\}$. We write $\Gamma \vdash_{DLAL^\star} t : A$ if there is an* **LAL** *derivation $\mathcal{D}$ of $\Gamma \vdash t : A$ in which*

- *any formula belongs to $\mathcal{L}_{DLAL^+}$,*
- *any instantiation formula B of a ($\forall$ e) rule in $\mathcal{D}$ belongs to $\mathcal{L}_{DLAL^\star}$.*

We stress that $\mathcal{L}_{DLAL^\star}$ and $\mathcal{L}_{DLAL^+}$ are sublanguages of $\mathcal{L}_{LAL}$ and *not* of $\mathcal{L}_{DLAL}$. Hence when writing $\Gamma \vdash_{DLAL^\star} t : A$, we are handling an **LAL** derivation (and not a **DLAL** one).

Theorem 6 (Embedding) *Let* $\Gamma; \Delta \vdash t : A$ *be a judgment in* **DLAL**. *Then* $\Gamma; \Delta \vdash_{DLAL} t : A$ *if and only if* $!\Gamma^\star, \Delta^\star \vdash_{DLAL^\star} t : A^\star$.

The 'only-if' direction is straightforward and the 'if' one will be given in Section 5.

If $\mathcal{D}$ is a **DLAL** derivation, let us denote by $\mathcal{D}^\star$ the **LAL** derivation obtained by the translation of Theorem 6.

Let us now define the depth of a **DLAL** derivation $\mathcal{D}$. For that, intuitively we need to consider the maximum, among all branches of the derivation, of the added numbers of (§ i) rules and arguments of an application rule ($\Rightarrow$ e) (noted u in the rule). More formally, the *depth* of a **DLAL** derivation $\mathcal{D}$ is thus the maximal number of premises of (§ i) and r.h.s. premises of ($\Rightarrow$ e) in a branch of $\mathcal{D}$.

In this way the depth of a **DLAL** derivation $\mathcal{D}$ corresponds to the depth of its translation $\mathcal{D}^\star$ in **LAL** , that is to say to the maximal nesting of exponential boxes in the corresponding proof-net (see [2] for the definition of proof-nets).

The data types of **LAL** can be directly adapted to **DLAL**. For instance, we have the following data types for booleans, unary integers and binary words in **LAL**:

$$\begin{aligned}\mathbf{B} &= \forall.\alpha.\alpha \multimap \alpha \multimap \alpha,\\ \mathbf{N}_L &= \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha),\\ \mathbf{W}_L &= \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha).\end{aligned}$$

The type **B** is also available in **DLAL** as it stands, while for the latter two, we have **N** and **W** such that $\mathbf{N}^\star = \mathbf{N}_L$ and $\mathbf{W}^\star = \mathbf{W}_L$ in **DLAL**:

$$\begin{aligned}\mathbf{N} &= \forall\alpha.(\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha),\\ \mathbf{W} &= \forall\alpha.(\alpha \multimap \alpha) \Rightarrow (\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha).\end{aligned}$$

The inhabitants of **B**, **N** and type **W** are the familiar Church codings of booleans, integers and words:

$$\frac{\Gamma_1;\Delta_1 \vdash t_1 : A_1 \quad \Gamma_2;\Delta_2 \vdash t_2 : A_2}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash t_1 \otimes t_2 : A_1 \otimes A_2}\ (\otimes\ \text{i})$$

$$\frac{\Gamma_1;\Delta_1 \vdash u : A_1 \otimes A_2 \quad \Gamma_2;x_1 : A_1, x_2 : A_2,\Delta_2 \vdash t : B}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash \mathsf{let}\ u\ \mathsf{be}\ x_1 \otimes x_2\ \mathsf{in}\ t : B}\ (\otimes\ \text{e})$$

Fig. 3. Derived rules

$$\underline{i} = \lambda x_0.\lambda x_1.x_i,$$
$$\underline{n} = \lambda f.\lambda x.\underbrace{f(f\ldots(f}_{n}x)\ldots),$$
$$\underline{w} = \lambda f_0.\lambda f_1.\lambda x.f_{i_1}(f_{i_2}\ldots(f_{i_n}x)\ldots),$$

with $i \in \{0,1\}$, $n \in N$ and $w = i_1 i_2 \cdots i_n \in \{0,1\}^*$. The following terms for addition and multiplication on Church integers are typable in **DLAL**:

$$\mathsf{add} = \lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)) : \quad N \multimap N \multimap N,$$
$$\mathsf{mult} = \lambda n.\lambda m.m(\lambda y.\ \mathsf{add}\ n\ y)\underline{0} : \quad N \Rightarrow N \multimap \S N.$$

It can be useful in practice to use a type $A \otimes B$. It can anyway be defined, thanks to full weakening:

$$A \otimes B = \forall\alpha.((A \multimap B \multimap \alpha) \multimap \alpha).$$

We use as syntactic sugar the following new constructions on terms with the typing rules of Figure 3:

$$t_1 \otimes t_2 = \lambda x.xt_1t_2,$$
$$\mathsf{let}\ u\ \mathsf{be}\ x_1 \otimes x_2\ \mathsf{in}\ t = u(\lambda x_1.\lambda x_2.t).$$

4 Properties of DLAL

4.1 Main properties

We will now present the main properties of **DLAL**. As the proofs of some theorems require some additional notions and definitions, we prefer to start by the statements of the results, and postpone the proofs to the following sections.

First of all, **DLAL** enjoys the subject reduction property with respect to lambda calculus, in contrast to **LAL**:

Theorem 7 (Subject Reduction) *If* $\Gamma; \Delta \vdash_{DLAL} t_0 : A$ *and* $t_0 \longrightarrow t_1$*, then* $\Gamma; \Delta \vdash_{DLAL} t_1 : A$*.*

To prove this property we will use another calculus, light affine lambda calculus and a simulation property of **DLAL** in this calculus (Theorem 22). For the proof see Section 6. Note that a direct proof was also given in [9].

DLAL types ensure the following strong normalization property:

Theorem 8 (Polynomial time strong normalization) *Let* t *be a lambda-term which has a typing derivation* $\mathcal{D}$ *of depth* d *in* **DLAL***. Then* t *reduces to the normal form in at most* $O(|t|^{2^d})$ *reduction steps and in time* $O(|t|^{2^{d+2}})$ *on a multi-tape Turing machine. This result holds independently of which reduction strategy we take.*

The bound $O(|t|^{2^d})$ for the number of reduction steps is slightly better than that of Theorem 1. The reason is that we are working on plain lambda terms and thus in particular do not need commuting reductions. The proof is again based on light affine lambda calculus and the simulation property. It will be given in Section 6.

Finally, even if **DLAL** offers good properties with respect to lambda calculus, in order to be convincing we expect it to be expressive enough to represent all Ptime functions, just as **LAL**; this is the case:

Theorem 9 (FP completeness) *If a function* $f : \{0,1\}^\star \to \{0,1\}^\star$ *is computable in time* $O(n^{2^d})$ *by a multi-tape Turing machine for some* d*, then there exists a lambda-term* t *such that* $\vdash_{DLAL} t : \mathbf{W} \multimap \S^{2d+2}\mathbf{W}$ *and* t *represents* f*.*

The depth of the result type $\S^{2d+2}\mathbf{W}$ above is smaller than the ones given by [20,2,30]. It is mainly due to the refined coding of polynomials, which is also available in **LAL** (see Proposition 11 and Theorem 33).

To prove the above theorem, we will define in Section 7 a translation from **LAL** to **DLAL** and use the FP completeness of **LAL**. Note that in [9] a direct proof was also given.

4.2 Iteration and example

In this section we will discuss the use of iteration in programming in **DLAL**. We will in particular describe the example of insertion sort.

4.2.1 Iteration

As in other polymorphic type systems, iteration schemes can be defined in **DLAL** for various data types. For instance for $\mathbf{N}$ and the following data type corresponding to lists over A:

$$\mathbf{L}(A) = \forall\alpha.(A \multimap \alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha),$$

we respectively have the following iteration schemes, for any type B:

$$\mathsf{iter}_B : (B \multimap B) \Rightarrow \S B \multimap \mathbf{N} \multimap \S B,$$
$$\mathsf{fold}_B : (A \multimap B \multimap B) \Rightarrow \S B \multimap \mathbf{L}(A) \multimap \S B.$$

They are defined by the same term:

$$\mathsf{iter}_B = \lambda F.\lambda b.\lambda n.nFb,$$
$$\mathsf{fold}_B = \lambda F.\lambda b.\lambda l.lFb.$$

Note that with the iter_B scheme for instance, if we iterate on type $B = \mathbf{N}$ a function $\mathbf{N} \multimap \mathbf{N}$ we obtain a term of type $\mathbf{N} \multimap \S\mathbf{N}$, which is thus itself not iterable. Typing therefore prevents nesting of iterations (see [33] for the relation with safe recursion).

This is the case for instance if we try to define addition on unary integers by iteration over one of the two arguments, using the successor function, and obtaining as type $\S\mathbf{N} \multimap \mathbf{N} \multimap \S\mathbf{N}$. However recall that we have given previously a term for addition with type $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$.

We want to show that by choosing suitably the type B on which to do the iteration, one can type in **DLAL** programs with nested iterations. To illustrate that on a simple example we first consider the case of addition. Can we define addition by an iteration giving it the type $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$?

Let us call *open Church integer* a term of the following form:

$$; f_1 : (\alpha \multimap \alpha), \ldots, f_n : (\alpha \multimap \alpha) \vdash \lambda x.f_1(\ldots(f_n x)\ldots) : (\alpha \multimap \alpha)$$

Informally speaking it is a Church integer, where the variables $f_1, \ldots, f_n$ have not been contracted nor bound yet.

Now we can define and type a successor on open Church integers in the following way:

$$; f : (\alpha \multimap \alpha) \vdash \lambda k.\lambda y.f(ky) : (\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)$$

Denote this term by osucc_f, indicating the free variable f. The typing judgement claimed above can be derived in the following way:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{;f:\alpha\multimap\alpha\vdash f:\alpha\multimap\alpha}\quad\dfrac{\dfrac{}{;k:\alpha\multimap\alpha\vdash k:\alpha\multimap\alpha}\quad\dfrac{}{;y:\alpha\vdash y:\alpha}}{;k:\alpha\multimap\alpha,y:\alpha\vdash ky:\alpha}}{;f:(\alpha\multimap\alpha),k:\alpha\multimap\alpha,y:\alpha\vdash f(ky):\alpha}}{;f:(\alpha\multimap\alpha),k:\alpha\multimap\alpha\vdash\lambda y.f(ky):\alpha\multimap\alpha}}{;f:(\alpha\multimap\alpha)\vdash\mathsf{osucc}_f:(\alpha\multimap\alpha)\multimap(\alpha\multimap\alpha)}$$

By applying iter_B on the type $B=\alpha\multimap\alpha$ of open Church integers, we get:

$$\dfrac{\dfrac{\dfrac{f:\alpha\multimap\alpha,f':\alpha\multimap\alpha;n:\mathbf{N},m:\mathbf{N}\vdash\mathsf{iter}_{\alpha\multimap\alpha}\ \mathsf{osucc}_f\ (m\ f')\ n:\S(\alpha\multimap\alpha)}{f:\alpha\multimap\alpha;n:\mathbf{N},m:\mathbf{N}\vdash\mathsf{iter}_{\alpha\multimap\alpha}\ \mathsf{osucc}_f\ (m\ f)\ n:\S(\alpha\multimap\alpha)}}{;n:\mathbf{N},m:\mathbf{N}\vdash\lambda f.\mathsf{iter}_{\alpha\multimap\alpha}\ \mathsf{osucc}_f\ (m\ f)\ n:(\alpha\multimap\alpha)\Rightarrow\S(\alpha\multimap\alpha)}}{;n:\mathbf{N},m:\mathbf{N}\vdash\lambda f.\mathsf{iter}_{\alpha\multimap\alpha}\ \mathsf{osucc}_f\ (m\ f)\ n:\mathbf{N}}$$

Note that a crucial point for the type derivation to be valid is the $(\Rightarrow\ e)$ rule, which forces osucc to have at most one occurrence of free variable. This restricts the osucc term not to increase the size of open integers by more than one unit, which is reminiscent of the *non-size-increasing* discipline of [24].

4.2.2 Insertion sort

We consider here a programming of the insertion sort algorithm in lambda calculus analogous to the one from [24]. The program is defined by two steps of iteration (one for defining the insertion, and one for the sorting) but interestingly enough is typable in **DLAL**. We will proceed in the same way as for addition in the previous section.

We consider the type $\mathbf{L}(A)$ of lists over a type A representing a totally ordered set, and assume given a term:

$$\mathsf{comp}:A\multimap A\multimap A\otimes A,\quad\text{with }\mathsf{comp}\ a_1\ a_2\to\begin{cases}a_1\otimes a_2 & \text{if } a_1\leq a_2\\ a_2\otimes a_1 & \text{if } a_2<a_1\end{cases}$$

Such a term can be programmed for instance if we choose for A a finite type, say the type $\mathbf{B}^{32}=\mathbf{B}\otimes\cdots\otimes\mathbf{B}$ (32 times) for 32-bit binary integers. As for unary integers, we can consider *open lists* of type $(\alpha\multimap\alpha)$ which have free variables of type $(A\multimap\alpha\multimap\alpha)$. Insertion will be defined by iterating a term acting on open lists.

We first define the insertion function by iteration on type $B = A \multimap (\alpha \multimap \alpha)$:

$$\mathsf{hcomp}_g = \lambda a^A f^B a'^A.\ \mathsf{let}\ (\mathsf{comp}\ \ a\ a')\ \mathsf{be}\ a_1 \otimes a_2\ \mathsf{in} \\ \lambda x^\alpha . g^B {a_1}^A (f a_2 x)^\alpha \quad : A \multimap B \multimap B$$

Observe that hcomp_g has only one occurrence of free variable g, so it can be used as argument of non-linear application. Then:

$$\mathsf{insert} = \lambda a_0^{\S A} l^{\mathbf{L}(A)}.\lambda g^B.(\mathsf{fold}_B\ \mathsf{hcomp}_g\ g\ l)\ a_0\ :\ \S A \multimap \mathbf{L}(A) \multimap \mathbf{L}(A)$$

Finally the sorting function is obtained by iteration on type $\mathbf{L}(A)$:

$$\mathsf{sort} = \lambda l^{\mathbf{L}(\S A)}.\mathsf{fold}_{\mathbf{L}(A)}\ \mathsf{insert}\ \mathsf{nil}\ l\ : \mathbf{L}(\S A) \multimap \S \mathbf{L}(A).$$

When working on lists over a standard data type such as $\mathbf{B}^{32}$, the coercion map $A \multimap \S A$ is always available (see Proposition 10). Hence in practice the sorting function admits a simpler type: $\mathbf{L}(A) \multimap \S \mathbf{L}(A)$.

Let us now give a general scheme for iteration over open lists. We start by the simple case which produces a function of type $\mathbf{L}(A) \multimap \mathbf{L}(A)$. We write it as a derivable rule, omitting the notation of terms to simplify readability:

$$\frac{;A \multimap \alpha \multimap \alpha \vdash A \multimap (\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha) \quad \Gamma, A \multimap \alpha \multimap \alpha; \Delta \vdash \S(\alpha \multimap \alpha)}{\Gamma; \Delta \vdash \mathbf{L}(A) \multimap \mathbf{L}(A)}$$

However insertion does not fit into this scheme, because it is defined by iteration on a functional type. Consider thus the following modified scheme, based on iteration over the type $C \multimap (\alpha \multimap \alpha)$:

$$\frac{\begin{array}{c};f : A \multimap \alpha \multimap \alpha \vdash t_f : A \multimap (C \multimap \alpha \multimap \alpha) \multimap (C \multimap \alpha \multimap \alpha) \\ \Gamma, f : A \multimap \alpha \multimap \alpha; \Delta \vdash u_f : \S(C \multimap \alpha \multimap \alpha)\end{array}}{\Gamma; \Delta \vdash v : \S C \multimap \mathbf{L}(A) \multimap \mathbf{L}(A)}$$

with $v = \lambda c l f.((\mathsf{fold}_{C \multimap \alpha \multimap \alpha}\ t_f\ u_f\ l)\ c)$.

To check that this scheme is derivable, first observe that the following judgement can be derived from the two premises:

$$f : A \multimap \alpha \multimap \alpha, \Gamma; \Delta, l : \mathbf{L}(A) \vdash \mathsf{fold}_{C \multimap \alpha \multimap \alpha}\ t_f\ u_f\ l : \S(C \multimap \alpha \multimap \alpha).$$

The conclusion can then easily be derived.

Finally the previous insertion function can be obtained applying this scheme, by taking $C = A$.

4.2.3 Coercion

Composition of programs in **DLAL** is quite delicate in the presence of modality § and non-linear arrow ⇒. When composing two programs involving § and ⇒, one frequently uses the coercion maps over data types. Below we only describe the coercion maps for booleans and integers, but in fact they can be easily generalised to other data types.

Proposition 10 (Coercion)

(1) There is a lambda-term $\mathsf{coer}_b : \mathbf{B} \multimap \S\mathbf{B}$ *such that* $\mathsf{coer}_b\underline{i} \longrightarrow^* \underline{i}$ *for* $i \in \{0,1\}$.
(2) There is a lambda-term $\mathsf{coer}_1 : \mathbf{N} \multimap \S\mathbf{N}$ *such that* $\mathsf{coer}_1\underline{n} \longrightarrow^* \underline{n}$ *for every integer* n.
(3) For any type A*, there is a lambda-term* $\mathsf{coer}_2 : \S(\mathbf{N} \Rightarrow A) \multimap (\mathbf{N} \multimap \S A)$ *such that* $\mathsf{coer}_2 t\underline{n} \longrightarrow^* t\underline{n}$ *for every term* t *and every integer* n.

Proof. 1. Let $\mathsf{coer}_b = \lambda x.x\underline{0}\underline{1}$.
2. Let $\mathsf{coer}_1 = \mathsf{iter}_{\mathbf{N}}\ \mathsf{succ}\ \underline{0}$, where succ is the usual successor $\lambda n.\lambda fx.f(nfx) : \mathbf{N} \multimap \mathbf{N}$.
3. We have 'lifted' versions of the successor and the zero:

$$\begin{aligned} \mathsf{lsucc} &= \lambda fx.f(\ \mathsf{succ}\ x) : (\mathbf{N} \Rightarrow A) \multimap (\mathbf{N} \Rightarrow A) \\ \mathsf{lzero} &= \lambda f.f\underline{0} \qquad\qquad : \S(\mathbf{N} \Rightarrow A) \multimap \S A. \end{aligned}$$

We therefore have

$$\mathsf{coer}_2 = \lambda fn.\mathsf{lzero}(\mathsf{iter}_{\mathbf{N}\Rightarrow A}\ \mathsf{lsucc}\ f\ n) : \S(\mathbf{N} \Rightarrow A) \multimap (\mathbf{N} \multimap \S A).$$

Given a term $t : \S(\mathbf{N} \Rightarrow A)$ and an integer n, it works as follows:

$$\begin{aligned} \mathsf{coer}_2 t\underline{n} &\longrightarrow^* \mathsf{lzero}(\mathsf{iter}\ \mathsf{lsucc}\ t\ \underline{n}) \\ &\longrightarrow^* \mathsf{lzero}(\mathsf{lsucc}^n t) \\ &\longrightarrow^* \mathsf{lzero}(\lambda x.t(\mathsf{succ}^n x)) \\ &\longrightarrow^* t(\mathsf{succ}^n \underline{0}) \quad \longrightarrow^* \quad t\underline{n}. \end{aligned}$$

As a slight variant of coer_2, we have a contraction map $\mathsf{cont} : \S(\mathbf{N} \Rightarrow \mathbf{N} \multimap A) \multimap (\mathbf{N} \multimap \S A)$ defined by

$$\begin{aligned} \mathsf{csucc} &= \lambda fxy.f(\ \mathsf{succ}\ x)(\ \mathsf{succ}\ y) && : (\mathbf{N} \Rightarrow \mathbf{N} \multimap A) \multimap (\mathbf{N} \Rightarrow \mathbf{N} \multimap A), \\ \mathsf{czero} &= \lambda f.f\underline{0}\underline{0} && : \S(\mathbf{N} \Rightarrow \mathbf{N} \multimap A) \multimap \S A, \\ \mathsf{cont} &= \lambda fn.\mathsf{czero}(\mathsf{iter}_{\mathbf{N}\Rightarrow\mathbf{N}\multimap A}\ \mathsf{csucc}\ f\ n) && : \S(\mathbf{N} \Rightarrow \mathbf{N} \multimap A) \multimap (\mathbf{N} \multimap \S A). \end{aligned}$$

By applying cont to the multiplication function $\mathsf{mult} : \mathbf{N} \Rightarrow \mathbf{N} \multimap \S\mathbf{N}$, the squaring function on unary integers can be represented: $\mathsf{square} = \mathsf{cont}(\mathsf{mult}) : \mathbf{N} \multimap \S^2\mathbf{N}$. We therefore obtain:

Proposition 11 *There is a closed term of type* $\mathbf{N} \multimap \S^{2d}\mathbf{N}$ *representing the function* n^{2^d} *for each* $d \geq 0$.

5 Embedding of DLAL into DLAL*

We are now going to prove Theorem 6, establishing that basically the system **DLAL** is the restriction of **LAL** to the language of $\mathcal{L}_{DLAL^+}$. Let us begin with the ordinary substitution lemma.

Lemma 12 (Substitution) *We consider derivations in* **DLAL**.

(1) *If* $\Gamma_1; \Delta_1 \vdash u : A$ *and* $\Gamma_2; x : A, \Delta_2 \vdash t : B$, *then* $\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t[u/x] : B$.
(2) *If* $; \Gamma_1, \Delta_1 \vdash u : A$ *and* $\Gamma_2; x : \S A, \Delta_2 \vdash t : B$, *then* $\Gamma_1, \Gamma_2; \S\Delta_1, \Delta_2 \vdash t[u/x] : B$.
(3) *If* $; z : C \vdash u : A$ *and* $x : A, \Gamma; \Delta \vdash t : B$, *then* $z : C, \Gamma; \Delta \vdash t[u/x] : B$.

Lemma 13 *If* $A, H \in \mathcal{L}_{DLAL^\star}$, *we have* $A[H/\alpha]^- = A^-[H^-/\alpha]$.

In order to prove Theorem 6 we will first prove a stronger lemma. The theorem will then follow directly.

Lemma 14

(1) *If* $!\Gamma, \Delta \vdash_{DLAL^\star} t : A$ *with* $\Gamma, \Delta, A \in \mathcal{L}_{DLAL^\star}$, *then* $\Gamma^-; \Delta^- \vdash_{DLAL} t : A^-$.
(2) *If* $!\Gamma, \Delta \vdash_{DLAL^\star} t : !A$ *with* $\Gamma, \Delta, A \in \mathcal{L}_{DLAL^\star}$, *then for any* **DLAL** *derivable judgement* $x : A^-, \Theta; \Pi \vdash_{DLAL} u : B$ *with* $\Theta, \Pi, B \in \mathcal{L}_{DLAL}$, *we have* $\Gamma^-, \Theta; \Delta^-, \Pi \vdash_{DLAL} u[t/x] : B$.

Proof. We prove these statements by a single induction over derivations in **DLAL***.

Let $\mathcal{D}$ be a **DLAL*** derivation of $!\Gamma, \Delta \vdash t : C$ with $\Gamma, \Delta \in \mathcal{L}_{DLAL^\star}$. We have to prove that: if $C \in \mathcal{L}_{DLAL^\star}$ then claim (1) holds; otherwise if $C = !A$ and $A \in \mathcal{L}_{DLAL^\star}$ then claim (2) holds. Consider the last rule of $\mathcal{D}$:
(Case 1) The derivation consists of the identity axiom. Straightforward.

(Case 2) The last rule is $(\multimap\ i)$:

$$\frac{y : A_1, !\Gamma, \Delta \vdash v : A_2}{!\Gamma, \Delta \vdash \lambda y.v : A_1 \multimap A_2}\ (\multimap \text{i})$$

and $A = A_1 \multimap A_2$, $t = \lambda y.v$. Therefore we are in the case of claim (1). By i.h. we have:

- if $A_1 \in \mathcal{L}_{DLAL^\star}$:
$$\Gamma^-; y : {A_1}^-, \Delta^- \vdash_{DLAL} v : {A_2}^-.$$
so, by applying the rule $(\multimap \text{i})$ in **DLAL**, we get:
$$\Gamma^-; \Delta^- \vdash_{DLAL} \lambda y.v : {A_1}^- \multimap {A_2}^-,$$
and ${A_1}^- \multimap {A_2}^- = (A_1 \multimap A_2)^-$.
- if $A_1 = !A_1'$, with $A_1' \in \mathcal{L}_{DLAL^\star}$:
$$\Gamma^-, y : {A_1'}^-; \Delta^- \vdash_{DLAL} v : {A_2}^-.$$
so, by applying the rule $(\Rightarrow \text{i})$ in **DLAL**, we get:
$$\Gamma^-; \Delta^- \vdash_{DLAL} \lambda y.v : {A_1'}^- \Rightarrow {A_2}^-,$$
and ${A_1'}^- \Rightarrow {A_2}^- = (A_1 \multimap A_2)^-$.

(Case 3) The last rule is $(\multimap\ e)$:

$$\frac{!\Gamma_1, \Delta_1 \vdash t_1 : C \multimap A \quad !\Gamma_2, \Delta_2 \vdash t_2 : C}{!\Gamma_1, !\Gamma_2, \Delta_1, \Delta_2 \vdash t_1 t_2 : A}\ (\multimap \text{e})$$

As by assumption we have $C \multimap A \in \mathcal{L}_{DLAL^+}$, we get that $A \in \mathcal{L}_{DLAL^\star}$. Thus we are in the case of claim (1). We now distinguish two possible subcases:

- first subcase: $C \in \mathcal{L}_{DLAL^\star}$. Then $(C \multimap A)^- = C^- \multimap A^-$. We apply the i.h. to the two subderivations, which are both in the case of claim (1), and apply the **DLAL** rule:
$$\frac{\Gamma_1^-; \Delta_1^- \vdash t_1 : C^- \multimap A^- \quad \Gamma_2^-; \Delta_2^- \vdash t_2 : C^-}{\Gamma_1^-, \Gamma_2^-; \Delta_1^-, \Delta_2^- \vdash t_1 t_2 : A^-}\ (\multimap \text{e})$$
- second subcase: $C = !C'$, with $C' \in \mathcal{L}_{DLAL^\star}$. Then $(C \multimap A)^- = {C'}^- \Rightarrow A^-$. We apply the i.h. to the l.h.s. premise, which is in the case of claim (1), and a $(\Rightarrow \text{e})$ rule in **DLAL**, to get:
$$\frac{\Gamma_1^-; \Delta_1^- \vdash t_1 : {C'}^- \Rightarrow A^- \quad ; z : C^- \vdash z : C^-}{z : C^-, \Gamma_1^-; \Delta_1^- \vdash t_1 z : A^-}\ (\Rightarrow \text{e})$$

Then by applying the i.h. to the **DLAL*** subderivation of $!\Gamma_2, \Delta_2 \vdash t_2 : C$, which is the case of claim (2), and to the **DLAL** judgement $z : C^-, \Gamma_1^-; \Delta_1^- \vdash_{DLAL} t_1 z : A^-$ we get:

$$\Gamma_1^-, \Gamma_2^-; \Delta_1^-, \Delta_2^- \vdash_{DLAL} t_1 t_2 : A^-.$$

(Case 4) The last rule is $(\forall\ i)$. This case is easy, so we omit it here.
(Case 5) The last rule is $(\forall\ e)$:

$$\frac{!\Gamma, \Delta \vdash t : \forall\alpha.A'}{!\Gamma, \Delta \vdash t : A'[H/\alpha]}\ (\forall\ \mathrm{e})$$

with $A = A'[H/\alpha]$. By assumption we have $\forall\alpha.A' \in \mathcal{L}_{DLAL^+}$, so $A' \in \mathcal{L}_{DLAL^\star}$. Moreover, also by assumption we know that $H \in \mathcal{L}_{DLAL^\star}$. Therefore we get: $A'[H/\alpha] \in \mathcal{L}_{DLAL^\star}$, and by Lemma 13, $A'[H/\alpha]^- = A'^-[H^-/\alpha]$. Thus we are in the case of claim (1). By applying the i.h. and a $(\forall$ e$)$ rule in **DLAL** we get:

$$\frac{\Gamma^-; \Delta^- \vdash t : \forall\alpha.A'^-}{\Gamma^-; \Delta^- \vdash t : A'^-[H^-/\alpha]}\ (\forall\ \mathrm{e})$$

(Case 6) The last rule is $(!\ i)$. We have to prove claim (2). Let us assume the judgement is of the form $y : !D \vdash t :!A$ (the case of $\vdash t :!A$ is similar). Its premise is $y : D \vdash t : A$. By induction hypothesis there is a **DLAL** derivation of $; y : D^- \vdash t : A^-$. Let $x : A^-, \Theta; \Pi \vdash u : B$ be a **DLAL** derivable judgement. By the substitution lemma (Lemma 12) we get $y : D^-, \Theta; \Pi \vdash_{DLAL} u[t/x] : B$.
(Case 7) The last rule is $(!\ e)$. Let us assume for instance that we are in the situation of claim (2) (claim (1) is easier anyway); the last rule is:

$$\frac{!\Gamma_1, \Delta_1 \vdash t_1 : !D \quad y : !D, !\Gamma_2, \Delta_2 \vdash t_2 : !A}{!\Gamma_1, !\Gamma_2, \Delta_1, \Delta_2 \vdash t_2[t_1/y] : !A}\ (!\ \mathrm{e})$$

Consider a **DLAL** derivable judgement $x : A^-, \Theta; \Pi \vdash u : B$. By i.h. on the r.h.s. premise of $(!\ e)$ we get that the following judgement is **DLAL** derivable:

$$y : D^-, \Gamma_2{}^-, \Theta; \Delta_2{}^-, \Pi \vdash u[t_2/x] : B.$$

Then by using this judgement with the i.h. on the l.h.s. premise of $(!\ e)$ we get that the following judgement is **DLAL** derivable:

$$\Gamma^-, \Theta; \Delta^-, \Pi \vdash (u[t_2/x])[t_1/y] : B.$$

Observe that the last term is equivalent to $u[t_2[t_1/y]/x] = u[t/x]$, hence the statement is proved.
(Case 8) The last rule is one of $(\S\ e)$, $(\S\ i)$, (Weak) and (Cntr). They are easy, so we omit them here.

6 Light affine lambda calculus and the simulation theorem

In this section, we will recall *light affine lambda calculus* (λLA) from [39] and give a simulation of **DLAL** typable lambda terms by λLA-terms. More specifically, we show that every **DLAL** typable lambda-term t translates to a λLA-term $t^\star$ (depending on the typing derivation for t) which is typable in $\mathbf{DLAL}^\star$, and that any β reduction sequence from t can be simulated by a *longer* λLA reduction sequence from $t^\star$. This simulation property directly implies the subject reduction theorem and the polynomial time strong normalization theorem for **DLAL**.

6.1 Light affine lambda calculus

The set of (pseudo) terms of λLA is defined by the following grammar:

$$M, N ::= x \mid \lambda x.M \mid MN \mid !M \mid \mathsf{let}\ N\ \mathsf{be}\ !x\ \mathsf{in}\ M \mid \S M \mid \mathsf{let}\ N\ \mathsf{be}\ \S x\ \mathsf{in}\ M.$$

The *depth* of M is the maximal number of occurrences of subterms of the form $!N$ and $\S N$ in a branch of the term tree for M. The *size* $|M|$ of the term M is the number of nodes of its term tree.

LAL, or more importantly its subsystem $\mathbf{DLAL}^\star$, can be considered as a type system for λLA. The system we present below uses two sorts of *discharged types* $[A]_!$ and $[A]_\S$ with $A \in \mathcal{L}_{LAL}$. Thus an environment Γ may contain a declaration $x : [A]_\dagger$. If $\Gamma = x_1 : A_1, \ldots, x : A_n$, then $[\Gamma]_\dagger$ denotes $x_1 : [A_1]_\dagger, \ldots, x_n : [A_n]_\dagger$. For a discharged type $[A]_\dagger$, $\widehat{[A]_\dagger}$ denotes the nondischarged type $\dagger A$. We also define $\widehat{A} = A$ and extend the notation to environments Γ in a natural way. We write $\Gamma \vdash_{LAL} M : A$ if M is a term of λLA and $\Gamma \vdash M : A$ is derivable by the type assignment rules in Figure 4. As before, the eigenvariable condition is imposed on the rule ($\forall$ i).

We also write $\Gamma \vdash_{DLAL^\star} M : A$ if it has a derivation in $\mathbf{DLAL}^\star$ (as in Definition 5); here we allow that a judgment may have a discharged declaration $x : [A]_\dagger$ with $A \in \mathcal{L}_{DLAL^\star}$ in its environment.

The reduction rules of λLA are given on Figure 5.

A term M is $(\S, !, com)$-*normal* if neither of the reduction rules ($\S$), (!), ($com1$) and ($com2$) applies to M. We write $M \xrightarrow{(\overline{\beta})} N$ when M reduces to N by one application of the (β) reduction rule followed by zero or several applications of the ($\S$), (!), (com1) and (com2) rules.

$$\frac{}{x : A \vdash x : A}\ (\mathrm{Id})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}\ (\multimap \mathrm{i}) \qquad \frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash MN : B}\ (\multimap \mathrm{e})$$

$$\frac{\Gamma_1 \vdash M : A}{\Gamma_1, \Gamma_2 \vdash M : A}\ (\mathrm{Weak}) \qquad \frac{x_1 : [A]_!, x_2 : [A]_!, \Gamma \vdash M : B}{x : [A]_!, \Gamma \vdash M[x/x_1, x/x_2] : B}\ (\mathrm{Cntr})$$

$$\frac{\Gamma, \Delta \vdash M : A}{[\Gamma]_!, [\Delta]_\S \vdash \S M : \S A}\ (\S\ \mathrm{i}) \qquad \frac{\Gamma_1 \vdash N : \S A \quad \Gamma_2, x : [A]_\S \vdash M : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ N\ \mathsf{be}\ \S x\ \mathsf{in}\ M : B}\ (\S\ \mathrm{e})$$

$$\frac{x : B \vdash M : A}{x : [B]_! \vdash\ !M :\ !A}\ (!\ \mathrm{i}) \qquad \frac{\Gamma_1 \vdash N :\ !A \quad \Gamma_2, x : [A]_! \vdash M : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ N\ \mathsf{be}\ !x\ \mathsf{in}\ M : B}\ (!\ \mathrm{e})$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha.A}\ (\forall\ \mathrm{i})\ (*) \qquad \frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M : A[B/\alpha]}\ (\forall\ \mathrm{e})$$

Fig. 4. LAL as a type system for λ_{LA}

$$\begin{array}{lrcl}
(\beta) & (\lambda x.M)N & \longrightarrow & M[N/x] \\
(\S) & \mathsf{let}\ \S N\ \mathsf{be}\ \S x\ \mathsf{in}\ M & \longrightarrow & M[N/x] \\
(!) & \mathsf{let}\ !N\ \mathsf{be}\ !x\ \mathsf{in}\ M & \longrightarrow & M[N/x] \\
(com1) & (\mathsf{let}\ N\ \mathsf{be}\ \dagger x\ \mathsf{in}\ M)L & \longrightarrow & \mathsf{let}\ N\ \mathsf{be}\ \dagger x\ \mathsf{in}\ (ML) \\
(com2) & \mathsf{let}\ (\mathsf{let}\ N\ \mathsf{be}\ \dagger x\ \mathsf{in}\ M)\ \mathsf{be}\ \dagger y\ \mathsf{in}\ L & \longrightarrow & \\
 & & & \mathsf{let}\ N\ \mathsf{be}\ \dagger x\ \mathsf{in}\ (\mathsf{let}\ M\ \mathsf{be}\ \dagger y\ \mathsf{in}\ L)
\end{array}$$

Fig. 5. Reduction rules of λ_{LA}

Given a λ_{LA}-term M, its *erasure* M^- is defined by:

$$\begin{array}{rr}
x^- = x & (MN)^- = M^-N^- \\
(\lambda x.M)^- = \lambda x.(M^-) & (\dagger M)^- = M^- \\
(\mathsf{let}\ N\ \mathsf{be}\ \dagger x\ \mathsf{in}\ M)^- = M^-[N^-/x]. &
\end{array}$$

The following holds quite naturally.

Lemma 15 *If $\Gamma \vdash_{DLAL^\star} M : A$ with M a λ_{LA}-term, then $\widehat{\Gamma} \vdash_{DLAL^\star} M^- : A$. In addition:*

(1) When M contains a subterm $\mathsf{let}\ N_1\ \mathsf{be}\ \S x\ \mathsf{in}\ N_2$, $no(x, N_2^-) \leq 1$.
(2) When M is $(\S, !, com)$-normal, $|M^-| \leq |M|$.

The following is the main result of [39]:

Theorem 16 (Polytime strong normalization for λLA) *A λLA-term M of depth d typable in* **LAL** *reduces to the normal form in $O(|M|^{2^{d+1}})$ reduction steps, and in time $O(|M|^{2^{d+2}})$ on a Turing machine. Moreover, any term N in the reduction sequence from M has a size bounded by $O(|M|^{2^d})$. These results hold independently of which reduction strategy we take.*

The upper bound $O(|M|^{2^{d+1}})$ for the length of reduction sequences is concerned with all reduction rules. When only the (β) reduction rule is concerned, it can be sharpened. In fact, Lemma 14 of [39] claims that given a λLA-term M, the length of any reduction sequence at fixed depth is bounded by $|M|^2$. In that proof, it is actually shown that the number of (β) *reductions* is bounded by $|M|$. This observation leads to the following improvement:

Lemma 17 *Let M be an* **LAL***-typable λLA-term M of depth d. Then the number of (β) reduction steps in any reduction sequence $M \longrightarrow M_1 \longrightarrow \cdots \longrightarrow M_n$ is bounded by $|M|^{2^d}$.*

The subject reduction theorem for λLA with respect to **LAL** is also proved in [39]. By inspecting the proof and since $\mathcal{L}_{DLAL^\star}$ is a subclass of $\mathcal{L}_{LAL}$, one can in fact observe:

Theorem 18 (Subject Reduction for λLA and DLAL*) *If $\Gamma \vdash_{DLAL^\star} M_0 : A$ and $M_0 \longrightarrow M_1$, then $\Gamma \vdash_{DLAL^\star} M_1 : A$.*

6.2 Simulation theorem

First of all, let us rephrase the 'only-if' direction of Theorem 6 in terms of λLA-terms rather than plain lambda terms.

Lemma 19 *If $\Gamma; \Delta \vdash t : A$ has a derivation of depth d in* **DLAL***, then there is a λLA-term $t^\star$ such that $!\Gamma^\star, \Delta^\star \vdash_{DLAL^\star} t^\star : A^\star$ and $t^{\star -} = t$. Moreover,*

(1) the depth of $t^\star$ is not greater than d;
(2) $x \in FV(t)$ if and only if $x \in FV(t^\star)$ for any variable x declared in Δ;
(3) $|t^\star| \leq 6(d+1)|t|$.

Proof. Let us denote by $|K|'$ the number of non-leaf nodes in the term formation tree of K, where K is either a lambda-term or a λLA-term. Namely,

$$|K|' = |K| - \text{the number of (bound and free) variable occurrences in } K.$$

We prove the existence of $t^\star$, claims (1), (2) and

(3') $|t^\star|' \leq 3(d+1)|t|'$

by induction on the structure of the derivation of $\Gamma; \Delta \vdash t : A$ in **DLAL**. Since $|K| \leq 2|K|' + 1$ and $|K|' + 1 \leq |K|$, (3) follows from (3'), by:

$$\begin{aligned}|t^\star| &\leq 2|t^\star|' + 1 \leq 6(d+1)|t|' + 1 \leq 6(d+1)(|t| - 1) + 1 \\ &\leq 6(d+1)|t| - 6(d+1) + 1 \leq 6(d+1)|t|.\end{aligned}$$

(Case 1) The derivation consists of the identity axiom. Straightforward.
(Case 2) The last rule is ($\Rightarrow$ e):

$$\frac{\Gamma_1; \Delta_1 \vdash t : A \Rightarrow B \quad ; z : C \vdash u : A}{\Gamma_1, z : C; \Delta_1 \vdash tu : B} \ (\Rightarrow \text{e})$$

Define

$$(tu)^\star \ = \ \mathsf{let}\ z\ \mathsf{be}\ !z'\ \mathsf{in}\ t^\star !(u^\star[z'/z])$$

with z' fresh. Since the immediate subderivations for t and u respectively have depth d and $d-1$, we have $|t^\star|' \leq 3(d+1)|t|'$ and $|u^\star|' \leq 3d|u|'$ by the induction hypothesis. Therefore we have

$$\begin{aligned}|(tu)^\star|' &= |t^\star|' + |u^\star|' + 3 \\ &\leq 3(d+1)|t|' + 3d|u|' + 3 \\ &\leq 3(d+1)(|t|' + |u|' + 1) = 3(d+1)|tu|',\end{aligned}$$

establishing claim (3'). The other claims are easy to show.
(Case 3) The last rule is (§ i):

$$\frac{; \Gamma, \Delta \vdash t : A}{\Gamma; \S\Delta \vdash t : \S A} \ (\S \text{ i})$$

If $t = x$, we set $t^\star = x$ and the claims are trivially satisfied. Otherwise, let $x_1, \ldots, x_m$ ($y_1, \ldots, y_n$, resp.) be the free variables declared in Γ (Δ, resp.) that actually occur in t as free variables. By i.h., we have $t^\star$ associated to the subderivation ending with the premise of the above rule. To the whole derivation, we associate a new λLA-term M defined by

$$\begin{aligned}M = {} & \mathsf{let}\ x_1\ \mathsf{be}\ !x_1'\ \mathsf{in}\ \cdots \mathsf{let}\ x_m\ \mathsf{be}\ !x_m'\ \mathsf{in} \\ & \quad \mathsf{let}\ y_1\ \mathsf{be}\ \S y_1'\ \mathsf{in}\ \cdots \mathsf{let}\ y_n\ \mathsf{be}\ \S y_n'\ \mathsf{in}\ \S t^\star[x_i'/x_i, y_j'/y_j].\end{aligned}$$

It is easy to see that $!\Gamma^\star, \S\Delta^\star \vdash_{DLAL^\star} M : \S A^\star$ and the claims (1) and (2) are satisfied. As to (3'), notice that we have $m + n + 1 \leq |t| \leq 2|t|' + 1 \leq 3|t|'$ since $|t|' \neq 0$. Hence we have

$$|M|' = |t^\star|' + m + n + 1 \leq 3d|t|' + 3|t|' = 3(d+1)|t|'.$$

(Case 4) The last rule is (§ e):

$$\frac{\Gamma_1; \Delta_1 \vdash u : \S A \quad \Gamma_2; x : \S A, \Delta_2 \vdash t : B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t[u/x] : B} \ (\S \text{ e})$$

Define $(t[u/x])^\star = t^\star[u^\star/x]$. If $x \in FV(t)$, we have $|(t[u/x])^\star|' = |t^\star|' + |u^\star|' \leq 3(d+1)|t|' + 3(d+1)|u|' \leq 3(d+1)|t[u/x]|'$. If not, x does not occur free in $FV(t^\star)$ either, by i.h. (2). Hence $|(t[u/x])^\star|' = |t^\star|' \leq 3(d+1)|t|' = 3(d+1)|t[u/x]|'$.

All other cases are straightforward.

To prove the simulation theorem, we need two technical lemmas.

Lemma 20 *Let M be a term of λLA.*

(1) If $\Gamma \vdash_{DLAL^\star} M : \forall\alpha_1 \cdots \forall\alpha_n.A \multimap B$ $(n \geq 0)$, then M is in one of the following forms: x, $M_1 M_2$, $\mathsf{let}\ M_1\ \mathsf{be}\ \dagger\, x\ \mathsf{in}\ M_2$, $\lambda x.M_0$.
(2) If $\Gamma \vdash_{DLAL^\star} M : \forall\alpha_1 \cdots \forall\alpha_n.\S A$ $(n \geq 0)$, then M is in one of the following forms: x, $M_1 M_2$, $\mathsf{let}\ M_1\ \mathsf{be}\ \dagger\, x\ \mathsf{in}\ M_2$, $\S M_0$.
(3) If $\Gamma \vdash_{DLAL^\star} M : !A$, then M is in one of the following forms: x, $\mathsf{let}\ M_1\ \mathsf{be}\ \dagger\, x\ \mathsf{in}\ M_2$, $!M_0$.

Proof. By induction on the structure of the derivation. The difference between (2) and (3) is due to the restriction that $B \multimap !A \notin \mathcal{L}_{DLAL^\star}$.

Lemma 21

(1) If $\Gamma \vdash_{DLAL^\star} MN : A$ and MN is $(\S, !, com)$-normal, then M is a variable x, an application $M_1 M_2$ or an abstraction $\lambda x.M_0$.
(2) If $\Gamma \vdash_{DLAL^\star} \mathsf{let}\ M\ \mathsf{be}\ \S x\ \mathsf{in}\ N : A$ and $(\mathsf{let}\ M\ \mathsf{be}\ \S x\ \mathsf{in}\ N)$ is $(\S, !, com)$-normal, then M is either a variable x or an application $M_1 M_2$.
(3) If $\Gamma \vdash_{DLAL^\star} \mathsf{let}\ M\ \mathsf{be}\ !x\ \mathsf{in}\ N : A$ and $(\mathsf{let}\ M\ \mathsf{be}\ !x\ \mathsf{in}\ N)$ is $(\S, !, com)$-normal, then M is a variable x.

Proof. (1) By induction on the structure of the derivation. If the last inference rule is $(\multimap e)$ of the form:

$$\frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash MN : B} \ (\multimap \text{ e})$$

then M cannot be of the form $\mathsf{let}\ M_1\ \mathsf{be}\ \dagger\, x\ \mathsf{in}\ M_2$ since MN is (com)-normal. Hence by Lemma 20 (1), M is a variable, an application or an abstraction. The other cases are obvious.

(2) By induction on the structure of the derivation. If the last rule is

$$\frac{\Gamma_1 \vdash M : \S A \quad \Gamma_2, x : [A]_\S \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ M\ \mathsf{be}\ \S x\ \mathsf{in}\ N : B}\ (\S\ \mathrm{e})$$

then M cannot be of the form $\mathsf{let}\ M_1\ \mathsf{be}\ \dagger\, y\ \mathsf{in}\ M_2$ since $\mathsf{let}\ M\ \mathsf{be}\ \S x\ \mathsf{in}\ N$ is (com)-normal. Likewise, M cannot be of the form $\S M_0$. Hence by Lemma 20 (2), M must be either a variable or an application.

(3) Similarly to (2).

Theorem 22 (Simulation) *Let M be a* **DLAL*** *typable* λLA*-term which is $(\S, !, com)$-normal. Let $t = M^-$. If t reduces to u by one (β) reduction step, then there is a* λLA*-term N such that $M \xrightarrow{(\overline{\beta})} N$, $N^- = u$ and N is $(\S, !, com)$-normal:*

$$\begin{array}{ccc} t & \xrightarrow{(\beta)} & u \\ \uparrow & & \vdots \\ M & \overset{(\overline{\beta})}{\cdots\cdots\rightarrow} & N \end{array}$$

Proof. It is clearly sufficient to find a λLA-term N' such that $M \xrightarrow{(\overline{\beta})} N'$ and $(N')^- = u$. A suitable $(\S, !, com)$-normal term N can then be obtained by reduction rules $(\S)$, $(!)$ and (com). The proof proceeds by induction on the structure of M.
(Case 1) M is a variable. Trivial.
(Case 2) M is of the form $\lambda x.M_0$. By the induction hypothesis.
(Case 3) M is of the form $M_1 M_2$. In this case, t is of the form $t_1 t_2$ with $M_1^- = t_1$ and $M_2^- = t_2$. When the redex is inside t_1 or t_2, the induction hypothesis applies. When the redex is t itself, then t_1 must be of the form $\lambda x.t_0$. By the definition of erasure, M_1 cannot be a variable nor an application. Therefore, by Lemma 21 (1), M_1 must be of the form $\lambda x.M_0$ with $M_0^- = t_0$. We therefore have

$$\begin{array}{ccc} (\lambda x.t_0)t_2 & \xrightarrow{(\beta)} & t_0[t_2/x] \\ \uparrow & & \vdots \\ (\lambda x.M_0)M_2 & \overset{(\overline{\beta})}{\cdots\cdots\rightarrow} & M_0[M_2/x] \end{array}$$

as required.
(Case 4) M is of the form $\dagger M_0$. By the induction hypothesis.
(Case 5) M is of the form $\mathsf{let}\ M_1\ \mathsf{be}\ \S x\ \mathsf{in}\ M_2$. In this case, t is of the form $t_2[t_1/x]$ with $M_1^- = t_1$ and $M_2^- = t_2$. By Lemma 21 (2), M_1 is either a variable or an application, and so is t_1. Therefore, no new redex is created by the substitution $t_2[t_1/x]$; the redex in t is either inside t_1 (with $x \in FV(t_2)$) or

results from a redex in t_2 by substituting t_1 for x.

In the former case, let $t_1 \longrightarrow u_1$. Then by the induction hypothesis, there is some N_1 such that

$$\begin{array}{ccc} t_1 & \xrightarrow{(\beta)} & u_1 \\ \uparrow & & \vdots \\ M_1 & \overset{(\overline{\beta})}{\cdots\cdots\rightarrow} & N_1 \end{array}$$

Therefore, we have

$$\begin{array}{ccc} t_2[t_1/x] & \xrightarrow{(\beta)} & t_2[u_1/x] \\ \uparrow & & \vdots \\ \mathsf{let}\ M_1\ \mathsf{be}\ \S x\ \mathsf{in}\ M_2 & \overset{(\overline{\beta})}{\cdots\cdots\rightarrow} & \mathsf{let}\ N_1\ \mathsf{be}\ \S x\ \mathsf{in}\ M_2 \end{array}$$

by noting that x occurs at most once in t_2 (Lemma 15 (1)).

In the latter case, let $t_2 \longrightarrow u_2$. By the induction hypothesis, there is N_2 such that

$$\begin{array}{ccc} t_2 & \xrightarrow{(\beta)} & u_2 \\ \uparrow & & \vdots \\ M_2 & \overset{\overline{(\beta)}}{\cdots\cdots\rightarrow} & N_2 \end{array}$$

We therefore have

$$\begin{array}{ccc} t_2[t_1/x] & \xrightarrow{(\beta)} & u_2[t_1/x] \\ \uparrow & & \vdots \\ \mathsf{let}\ M_1\ \mathsf{be}\ \S x\ \mathsf{in}\ M_2 & \overset{\overline{(\beta)}}{\cdots\cdots\rightarrow} & \mathsf{let}\ M_1\ \mathsf{be}\ \S x\ \mathsf{in}\ N_2 \end{array}$$

as required.
(Case 6) M is of the form $\mathsf{let}\ M_1\ \mathsf{be}\ !x\ \mathsf{in}\ M_2$. By Lemma 21 (3), M_1 is a variable y. Hence in this case, t is of the form $t_2[y/x]$. Therefore, the redex in t results from a redex in t_2 by substituting y for x. The rest is analogous to the previous case.

Remark 23 *Note that the statement of Theorem 22 could not be extended to arbitrary* **LAL** *typable* λLA*-terms. Indeed sometimes one* (β) *step in a* **LAL** *typed lambda-term might not be simulated by a sequence of* $(\overline{\beta})$ *steps in the*

corresponding λLA*-term. The reason for that is that the* let (.) be !x in (.) *construct allows for sharing in* λLA*-terms. This is directly related to the fact that* **LAL** *lambda-terms do not admit subject-reduction for* (β) *reduction (recall also the discussion of Sections 2.2 and 2.3).*

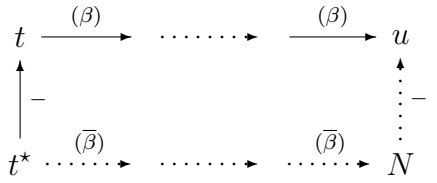
6.3 Subject reduction and Ptime strong normalization

Having obtained the simulation theorem, we are now ready to prove the subject reduction theorem (Theorem 7) and the polynomial time strong normalization theorem (Theorem 8).

Proof of Theorem 7. Suppose that $\Gamma; \Delta \vdash_{DLAL} t : A$ and $t \longrightarrow u$. By Lemma 19, there is a λLA-term $t^\star$ such that $!\Gamma^\star, \Delta^\star \vdash_{DLAL^\star} t^\star : A^\star$. Moreover, the simulation theorem implies that there is a λLA-term N such that $t^\star \xrightarrow{(\overline{\beta})} N$ and $N^- = u$. Finally, Theorem 18, Lemma 15 and Theorem 6 together imply that $\Gamma; \Delta \vdash_{DLAL} u : A$.

Proof of Theorem 8. By Lemma 19, there is a λLA-term $t^\star$ such that $t^{\star -} = t$ and $|t^\star| \leq 6(d+1)|t|$ where d is the depth of the typing derivation for t.

By the simulation theorem, we have:



Since by Lemma 17 the length of the $(\overline{\beta})$ reduction sequence from $t^\star$ to N is bounded by $O(|t^\star|^{2^d}) = O(|t|^{2^d})$, so is the one from t to u.

To show that the normal form can be computed in time $O(|t|^{2^{d+2}})$ by a Turing machine, notice that any term u such that $t \longrightarrow^* u$ has a size bounded by $O(|t|^{2^d})$ by Theorems 16, 18 and Lemma 15 (2). Since a beta reduction step can be performed in time quadratic in the size of a term, the overall time for normalization is bounded by the order of

$$|t|^{2^d} \cdot (|t|^{2^d})^2 \leq |t|^{2^{d+2}}.$$

7 Translation of LAL into DLAL

Although **DLAL** is a proper subsystem of **LAL**, it is as expressive as **LAL**, both logically and computationally. As a consequence, all polynomial time

functions are representable in **DLAL**.

7.1 Definability of **LAL** *types in* **DLAL**

First of all, we show that the types of **LAL** are definable in **DLAL** by using second order quantification. Our coding below is essentially the same as Plotkin's [35,21]). Define a mapping $(\cdot)^\bullet$ from $\mathcal{L}_{LAL}$ to $\mathcal{L}_{DLAL}$ as follows:

- $(!A)^\bullet = \forall\alpha.((A^\bullet \Rightarrow \alpha) \multimap \alpha)$, where α is fresh.
- $(\cdot)^\bullet$ commutes to the other connectives.

Define also a mapping from λLA-terms to lambda-terms by:

$$\begin{aligned}
x^\bullet &= x \\
(\lambda x.M)^\bullet &= \lambda x.M^\bullet \\
(MN)^\bullet &= M^\bullet N^\bullet \\
(!M)^\bullet &= \lambda x.xM^\bullet, \text{ where } x \text{ is fresh.} \\
(\mathsf{let}\ N\ \mathsf{be}\ !y\ \mathsf{in}\ M)^\bullet &= N^\bullet(\lambda y.M^\bullet) \\
(\S M)^\bullet &= M^\bullet \\
(\mathsf{let}\ N\ \mathsf{be}\ \S y\ \mathsf{in}\ M)^\bullet &= M^\bullet[N^\bullet/y]
\end{aligned}$$

This mapping preserves typing:

Proposition 24 *Let* $[\Gamma]_!$ *be an environment* $x_1 : [A_1]_!, \ldots, x_m : [A_m]_!$ *and* Δ *be* $y_1 : [B_1]_\S, \ldots, y_n : [B_n]_\S, z_1 : C_1, \ldots, z_k : C_k$. *Then,*

$$[\Gamma]_!, \Delta \vdash_{LAL} M : A \qquad \textit{implies} \qquad \Gamma^\bullet; \Delta^\bullet \vdash_{DLAL} M^\bullet : A^\bullet,$$

where $\Gamma^\bullet = x_1 : A_1^\bullet, \ldots, x_m : A_m^\bullet$ *and* $\Delta^\bullet = y_1 : \S B_1^\bullet, \ldots, y_n : \S B_n^\bullet, z_1 : C_1^\bullet, \ldots, z_k : C_k^\bullet$.

Proof. By induction on the structure of the derivation.
(Case 1) The last rule is (! i):

$$\frac{y : B \vdash M : A}{y : [B]_! \vdash !M : !A}$$

We have

$$\dfrac{\dfrac{\dfrac{\overline{; x : A^\bullet \Rightarrow \alpha \vdash x : A^\bullet \Rightarrow \alpha} \quad ; y : B^\bullet \vdash M^\bullet : A^\bullet}{y : B^\bullet; x : A^\bullet \Rightarrow \alpha \vdash xM^\bullet : \alpha}}{y : B^\bullet; \vdash \lambda x.xM^\bullet : (A^\bullet \Rightarrow \alpha) \multimap \alpha}}{y : B^\bullet; \vdash \lambda x.xM^\bullet : \forall\alpha.(A^\bullet \Rightarrow \alpha) \multimap \alpha}$$

(Case 2) The last rule is (! e):

$$\frac{[\Gamma_1]_!, \Delta_1 \vdash N : !A \quad y : [A]_!, [\Gamma_2]_!, \Delta_2 \vdash M : B}{[\Gamma_1]_!, [\Gamma_2]_!, \Delta_1, \Delta_2 \vdash \mathsf{let}\ N\ \mathsf{be}\ !y\ \mathsf{in}\ M : B}$$

We have

$$\frac{\dfrac{\Gamma_1^\bullet; \Delta_1^\bullet \vdash N^\bullet : \forall\alpha.(A^\bullet \Rightarrow \alpha) \multimap \alpha}{\Gamma_1^\bullet; \Delta_1^\bullet \vdash N^\bullet : (A^\bullet \Rightarrow B^\bullet) \multimap B^\bullet} \quad \dfrac{y : A^\bullet, \Gamma_2^\bullet; \Delta_2^\bullet \vdash M^\bullet : B^\bullet}{\Gamma_2^\bullet; \Delta_2^\bullet \vdash \lambda y.M^\bullet : A^\bullet \Rightarrow B^\bullet}}{\Gamma_1^\bullet, \Gamma_2^\bullet; \Delta_1^\bullet, \Delta_2^\bullet \vdash N^\bullet(\lambda y.M^\bullet) : B^\bullet}$$

The other cases are straightforward.

This translation also preserves the reduction rules of λLA other than ($com1$) and ($com2$) for the let-! operator.

Lemma 25 *Let M and N be terms of λLA. Then we have $(M[N/x])^\bullet = M^\bullet[N^\bullet/x]$.*

Proof. By induction on the structure of M.

Let ($com\S$) stand for the commuting reduction rules for let-§ constructs.

Proposition 26 *Let $M \xrightarrow{(r)} N$, where (r) is (β), (!), (§) or ($com\S$). Then we have*

$$M^\bullet \longrightarrow^* N^\bullet.$$

Proof. For instance, the (!) reduction rule $\mathsf{let}\ !N\ \mathsf{be}\ !y\ \mathsf{in}\ M \longrightarrow M[N/y]$ can be simulated as follows:

$$\begin{aligned}(\mathsf{let}\ !N\ \mathsf{be}\ !y\ \mathsf{in}\ M)^\bullet &= (\lambda x.xN^\bullet)(\lambda y.M^\bullet)\\ &\longrightarrow (\lambda y.M^\bullet)N^\bullet\\ &\longrightarrow M^\bullet[N^\bullet/y] = (M[N/y])^\bullet.\end{aligned}$$

The other reduction rules are easily handled.

However, this is not true of the commuting reduction rules for the let-! operator.

7.2 Preservation of representable functions

We would like to show that **DLAL** expresses as many algorithms as **LAL** based on the translation $(\,\cdot\,)^\bullet$. But there are two obstacles.

(1) The translation does not preserve the commuting reduction rules for let-!.
(2) $M^\bullet$ does not represent a function over data types even though M does, because the translation modifies the input/output types.

To overcome the first difficulty, we consider, not a λLA term M itself, but its erasure M^-, which does not need any commuting reductions. To overcome the second, we introduce an *encoder* and a *decoder*. Namely, for each $A \in \mathcal{L}_{DLAL}$, we define two lambda terms

$$\mathsf{en}_A : A \multimap A^{\star\bullet}, \qquad \mathsf{de}_A : A^{\star\bullet} \multimap A.$$

$$\begin{array}{rlrl}
\mathsf{en}_\alpha(t) &= t & \mathsf{de}_\alpha(t) &= t \\
\mathsf{en}_{\forall\alpha.A}(t) &= \mathsf{en}_A(t) & \mathsf{de}_{\forall\alpha.A}(t) &= \mathsf{de}_A(t) \\
\mathsf{en}_{\S A}(t) &= \mathsf{en}_A(t) & \mathsf{de}_{\S A}(t) &= \mathsf{de}_A(t) \\
\mathsf{en}_{A\multimap B}(t) &= \lambda x.\mathsf{en}_B(t\ \mathsf{de}_A(x)) & \mathsf{de}_{A\multimap B}(t) &= \lambda x.\mathsf{de}_B(t\ \mathsf{en}_A(x)) \\
\mathsf{en}_{A\Rightarrow B}(t) &= \lambda x.\mathsf{en}_B(x(\lambda y.t\ \mathsf{de}_A(y))) & \mathsf{de}_{A\Rightarrow B}(t) &= \lambda x.\mathsf{de}_B(t(\lambda y.y\ \mathsf{en}_A(x)))
\end{array}$$

We have:

Lemma 27 *For any $A \in \mathcal{L}_{DLAL}$, the following judgements are derivable in* **DLAL** *:*

$$;z : A \vdash \mathsf{en}_A(z) : A^{\star\bullet}, \qquad ;z : A^{\star\bullet} \vdash \mathsf{de}_A(z) : A.$$

Proof. By induction on the structure of A. Here we only deal with $\mathsf{en}_{A\Rightarrow B}$ and $\mathsf{de}_{A\Rightarrow B}$. For both of them, notice

$$(A \Rightarrow B)^{\star\bullet} = (!A^\star)^\bullet \multimap B^{\star\bullet} = (\forall\alpha.(A^{\star\bullet} \Rightarrow \alpha) \multimap \alpha) \multimap B^{\star\bullet}.$$

As to $\mathsf{en}_{A\Rightarrow B}$, we have:

$$\dfrac{\dfrac{;x : (!A^\star)^\bullet \vdash x : (!A^\star)^\bullet}{;x : (!A^\star)^\bullet \vdash x : (A^{\star\bullet} \Rightarrow B) \multimap B} \qquad \dfrac{\dfrac{;z : A \Rightarrow B \vdash z : A \Rightarrow B \quad ;y : A^{\star\bullet} \vdash \mathsf{de}_A(y) : A}{y : A^{\star\bullet}; z : A \Rightarrow B \vdash z\ \mathsf{de}_A(y) : B}}{;z : A \Rightarrow B \vdash \lambda y.z\ \mathsf{de}_A(y) : A^{\star\bullet} \Rightarrow B}}{;z : A \Rightarrow B, x : (!A^\star)^\bullet \vdash x(\lambda y.z\ \mathsf{de}_A(y)) : B}$$

By substituting the above conclusion into $w : B \vdash \mathsf{en}_B(w) : B^{\star\bullet}$, one obtains the desired judgement.

As to $\mathsf{de}_{A\Rightarrow B}$, we have:

$$\dfrac{;z : (A \Rightarrow B)^{\star\bullet} \vdash z : (!A^\star)^\bullet \multimap B^{\star\bullet} \qquad \dfrac{\dfrac{\dfrac{;y : A^{\star\bullet} \Rightarrow \alpha \vdash y : A^{\star\bullet} \Rightarrow \alpha \quad ;x : A \vdash \mathsf{en}_A(x) : A^{\star\bullet}}{x : A; y : A^{\star\bullet} \Rightarrow \alpha \vdash y\ \mathsf{en}_A(x) : \alpha}}{x : A;\vdash \lambda y.y\ \mathsf{en}_A(x) : (A^{\star\bullet} \Rightarrow \alpha) \multimap \alpha}}{x : A;\vdash \lambda y.y\ \mathsf{en}_A(x) : (!A^\star)^\bullet}}{x : A; z : (A \Rightarrow B)^{\star\bullet} \vdash z(\lambda y.y\ \mathsf{en}_A(x)) : B^{\star\bullet}}$$

By substituting the above conclusion into $w : B^{\star\bullet} \vdash \mathsf{de}_B(w) : B$, one obtains the desired judgement.

We then show that $M^\bullet$ and M^- are related via de_A, when A is a Π_1 type in $\mathcal{L}_{DLAL}$ and M is a λLA term of type $A^\star$. Here, a direct induction argument would not work, because the type derivation for $\vdash M : A^\star$ might involve types outside $\mathcal{L}_{DLAL^\star}$. We therefore employ logical relations to work on a stronger induction hypothesis.

Consider a binary relation R over the set of plain lambda terms: $R \subseteq \Lambda \times \Lambda$. Such a relation R is *$\beta\eta$-closed* if $(t, u) \in R$, $t =_{\beta\eta} t'$ and $u =_{\beta\eta} u'$ imply $(t', u') \in R$. A *valuation* φ maps each type variable α to a *$\beta\eta$-closed* relation.

Given a valuation φ, one can inductively define a binary relation $[\![A]\!]_\varphi$ for each **LAL** type A:

- $[\![\alpha]\!]_\varphi = \varphi(\alpha)$. $[\![\S A]\!]_\varphi = [\![A]\!]_\varphi$.
- $(t, u) \in [\![A \multimap B]\!]_\varphi \iff$ for any $(v_1, v_2) \in [\![A]\!]_\varphi$, $(tv_1, uv_2) \in [\![B]\!]_\varphi$.
- $(t, u) \in [\![\forall\alpha.A]\!]_\varphi \iff$ for any $\beta\eta$-closed relation R, $(t, u) \in [\![A]\!]_{\varphi[\alpha\mapsto R]}$, where $\varphi[\alpha \mapsto R]$ denotes the valuation which maps α to R and agrees with φ on other propositional variables.
- $(t, u) \in [\![!A]\!]_\varphi \iff$ there is a lambda-term t' such that $t =_{\beta\eta} \lambda z.zt'$ and $(t', u) \in [\![A]\!]_\varphi$.

It is then easy to see that $[\![A]\!]_\varphi$ is $\beta\eta$-closed for any A. Moreover, it commutes with the substitution:

Lemma 28 *For any* **LAL** *types A and B, $[\![A[B/\alpha]]\!]_\varphi$ is equal to $[\![A]\!]_{\varphi[\alpha\mapsto[\![B]\!]_\varphi]}$.*

The proof is by an easy induction on the structure of A.

As usual with logical relations, we have the basic lemma.

Lemma 29 (Basic Lemma) *Let Γ be an environment $x_1 : A_1, \ldots, x_n : A_n, x_{n+1} : [A_{n+1}]_\dagger, \ldots, x_m : [A_m]_\dagger$. If $\Gamma \vdash_{LAL} M : B$, then for any valuation φ and any $(u_i, v_i) \in [\![A_i]\!]_\varphi$ $(1 \le i \le m)$, we have $(M^\bullet[\vec{u}/\vec{x}], M^-[\vec{v}/\vec{x}]) \in [\![B]\!]_\varphi$.*

Proof. By induction on the structure of the derivation.
(Case 1) The derivation consists of the identity axiom. Straightforward.
(Case 2) The last rule is $(\multimap\ i)$:

$$\frac{y : A, \Gamma \vdash M : B}{\Gamma \vdash \lambda y.M : A \multimap B}$$

Let $(u_i, v_i) \in [\![A_i]\!]_\varphi$ for each $1 \le i \le m$. Then for any $(u, v) \in [\![A]\!]_\varphi$, we have $(M^\bullet[u/y, \vec{u}/\vec{x}], M^-[v/y, \vec{v}/\vec{x}]) \in [\![B]\!]_\varphi$ by i.h. This shows that we have $((\lambda y.M)^\bullet[\vec{u}/\vec{x}], (\lambda y.M)^-[\vec{v}/\vec{x}]) \in [\![A \multimap B]\!]_\varphi$, because $(\lambda y.M)^\bullet[\vec{u}/\vec{x}]u =_{\beta\eta}$

$M^\bullet[u/y, \vec{u}/\vec{x}]$ and $(\lambda y.M)^-[\vec{v}/\vec{x}]v =_{\beta\eta} M^-[v/y, \vec{v}/\vec{x}]$.
(Case 3) The last rule is $(\multimap\ e)$:

$$\frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\ N : B}$$

For simplicity, we assume Γ_1 and Γ_2 are empty (the argument is then easily adapted to the general case). By i.h., $(M^\bullet, M^-) \in [\![A \multimap B]\!]_\varphi$ and $(N^\bullet, N^-) \in [\![A]\!]_\varphi$. Hence $((MN)^\bullet, (MN)^-) \in [\![B]\!]_\varphi$.
(Case 4) The last rule is $(!\ i)$:

$$\frac{y : B \vdash M : A}{y : [B]_! \vdash\ !M : !A}$$

Let $(u, v) \in [\![B]\!]_\varphi$. Then $(!M)^\bullet[u/y] = \lambda z.zM^\bullet[u/y]$ and $(M^\bullet[u/y], M^-[v/y]) \in [\![A]\!]_\varphi$ by i.h. Hence $((!M)^\bullet[u/y], (!M)^-[v/y]) \in [\![!A]\!]_\varphi$.
(Case 5) The last rule is $(!\ e)$:

$$\frac{\Gamma_1 \vdash N : !A \quad y : [A]_!, \Gamma_2 \vdash M : C}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ N\ \mathsf{be}\ !y\ \mathsf{in}\ M : C}$$

For simplicity, we assume Γ_1 and Γ_2 are empty. By i.h., $(N^\bullet, N^-) \in [\![!A]\!]_\varphi$, so there is some u such that $N^\bullet =_{\beta\eta} \lambda z.zu$ (with z fresh) and $(u, N^-) \in [\![A]\!]_\varphi$. Hence by i.h. we have $(M^\bullet[u/y], M^-[N^-/y]) \in [\![C]\!]_\varphi$. Since

$$(\mathsf{let}\ N\ \mathsf{be}\ !y\ \mathsf{in}\ M)^\bullet = N^\bullet(\lambda y.M^\bullet) =_{\beta\eta} (\lambda z.zu)(\lambda y.M^\bullet) =_{\beta\eta} M^\bullet[u/y]$$

and $(\mathsf{let}\ N\ \mathsf{be}\ !y\ \mathsf{in}\ M)^- = M^-[N^-/y]$, we are done.
(Case 6) The last rule is $(\forall\ i)$:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall\alpha.A}$$

Let φ be a valuation and let $(u_i, v_i) \in [\![A_i]\!]_\varphi$ for every $1 \leq i \leq m$. Since α does not appear free in any A_i, we have $(u_i, v_i) \in [\![A_i]\!]_{\varphi[\alpha \mapsto R]}$ for any $\beta\eta$-closed relation R. Therefore, we have $(M^\bullet[\vec{u}/\vec{x}], M^-[\vec{v}/\vec{x}]) \in [\![A]\!]_{\varphi[\alpha \mapsto R]}$ by i.h. Hence $(M^\bullet[\vec{u}/\vec{x}], M^-[\vec{v}/\vec{x}]) \in [\![\forall\alpha.A]\!]_\varphi$.
(Case 7) The last rule is $(\forall\ e)$:

$$\frac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash M : A[B/\alpha]}$$

For simplicity, we assume Γ is empty. By i.h., we have $(M^\bullet, M^-) \in [\![\forall\alpha.A]\!]_\varphi$, and hence $(M^\bullet, M^-) \in [\![A]\!]_{\varphi[\alpha \mapsto R]}$ for any $\beta\eta$-closed relation R. In particular, $(M^\bullet, M^-) \in [\![A]\!]_{\varphi[\alpha \mapsto [\![B]\!]_\varphi]}$, and by Lemma 28, $(M^\bullet, M^-) \in [\![A[B/\alpha]]\!]_\varphi$.

Other cases are straightforward.

Note that the $\beta\eta$-equality $=_{\beta\eta}$ itself is a $\beta\eta$-closed relation. So let us define a valuation φ by taking $\varphi(\alpha)$ to be $=_{\beta\eta}$ for every variable α. We denote the resulting logical relations $[\![A]\!]_\varphi$ simply by $[\![A]\!]$. With this canonical valuation, we have the following:

Lemma 30 *Let D be a* **DLAL** *type.*

(1) If D is Π_1 and $(t,u) \in [\![D^\star]\!]$, then $\mathsf{de}_D(t) =_{\beta\eta} u$.
(2) If D is Σ_1, then for any lambda-term t of the form $xt_1 \cdots t_n$ ($n \geq 0$), we have $(\mathsf{en}_D(t), t) \in [\![D^\star]\!]$.

Proof. By induction on the structure of D.
(Case 1) D is a variable α. Straightforward because $[\![\alpha]\!]$ is just $=_{\beta\eta}$ and $\mathsf{de}_\alpha, \mathsf{en}_\alpha$ are just identity.
(Case 2) D is of the form $A \multimap B$.

(1) Suppose that $A \multimap B$ is Π_1 and $(t,u) \in [\![A^\star \multimap B^\star]\!]$. Let x be a fresh variable. Since $(\mathsf{en}_A(x), x) \in [\![A^\star]\!]$ by i.h. (with A a Σ_1 type), we have $(t\ \mathsf{en}_A(x), ux) \in [\![B^\star]\!]$. Hence by i.h. (with B a Π_1 type), $\mathsf{de}_B(t\ \mathsf{en}_A(x)) =_{\beta\eta} ux$. Therefore,

$$\mathsf{de}_{A\multimap B}(t) = \lambda x.\mathsf{de}_B(t\ \mathsf{en}_A(x)) =_{\beta\eta} \lambda x.ux =_{\beta\eta} u.$$

(2) Suppose that $A \multimap B$ is Σ_1 and let $t = xt_1 \cdots t_n$. To prove $(\mathsf{en}_{A\multimap B}(t), t) \in [\![A^\star \multimap B^\star]\!]$, it is sufficient to show that $(\mathsf{en}_{A\multimap B}(t)u, tv)) \in [\![B^\star]\!]$ holds for any $(u,v) \in [\![A^\star]\!]$. By i.h., $\mathsf{de}_A(u) =_{\beta\eta} v$. Hence

$$\mathsf{en}_{A\multimap B}(t)u = (\lambda x.\mathsf{en}_B(t\ \mathsf{de}_A(x)))u =_{\beta\eta} \mathsf{en}_B(t\ \mathsf{de}_A(u)) =_{\beta\eta} \mathsf{en}_B(tv).$$

Therefore by i.h., we have $(\mathsf{en}_{A\multimap B}(t)u, tv) \in [\![B^\star]\!]$.

(Case 3) D is of the form $A \Rightarrow B$.

(1) Suppose that $A \Rightarrow B$ is Π_1 and $(t,u) \in [\![!A^\star \multimap B^\star]\!]$. Let x be a fresh variable. Since $(\mathsf{en}_A(x), x) \in [\![A^\star]\!]$ by i.h, we have $(\lambda y.y\ \mathsf{en}_A(x), x) \in [\![!A^\star]\!]$. Hence $(t(\lambda y.y\ \mathsf{en}_A(x)), ux) \in [\![B^\star]\!]$, and so $\mathsf{de}_B(t(\lambda y.y\ \mathsf{en}_A(x))) =_{\beta\eta} ux$. We therefore have

$$\mathsf{de}_{A\Rightarrow B}(t) = \lambda x.\mathsf{de}_B(t(\lambda y.y\ \mathsf{en}_A(x))) =_{\beta\eta} \lambda x.ux =_{\beta\eta} u.$$

(2) Suppose that $A \Rightarrow B$ is Σ_1. Let $t = xt_1 \cdots t_n$ and $(u,v) \in [\![!A^\star]\!]$. Our purpose is to show $(\mathsf{en}_{A\Rightarrow B}(t)u, tv) \in [\![B^\star]\!]$.

There is some u' such that $u =_{\beta\eta} \lambda y.yu'$ and $(u', v) \in [\![A^\star]\!]$. By i.h. on A we have $\mathsf{de}_A(u') =_{\beta\eta} v$. Hence

$$\mathsf{en}_{A\Rightarrow B}(t)u =_{\beta\eta} (\lambda x.\mathsf{en}_B(x\lambda y.t\ \mathsf{de}_A(y)))(\lambda z.zu')$$

$$=_{\beta\eta} \mathsf{en}_B((\lambda z.zu')(\lambda y.t\ \mathsf{de}_A(y))) =_{\beta\eta} \mathsf{en}_B(t\ \mathsf{de}_A(u')) =_{\beta\eta} \mathsf{en}_B(tv).$$

Moreover by i.h. on B we have $(\mathsf{en}_B(tv), tv) \in [\![B^\star]\!]$, so as $\mathsf{en}_{A\Rightarrow B}(t)u =_{\beta\eta} \mathsf{en}_B(tv)$ we get that: $(\mathsf{en}_{A\Rightarrow B}(t)u, tv) \in [\![B^\star]\!]$.

(Case 4) D is of the form $\forall\alpha.A$. It is sufficient to show 1. But this is obvious because $(t,u) \in [\![\forall\alpha.A]\!]$ implies $(t,u) \in [\![A]\!]$.
(Case 5) D is of the form $\S A$. Straightforward.

Combined with the basic lemma, it results in:

Corollary 31 *If A is a Π_1 type of* **DLAL** *and $\vdash_{LAL} M : A^\star$, then $\mathsf{de}_A(M^\bullet) =_{\beta\eta} M^-$.*

7.3 Ptime completeness

Let us now show that **DLAL** has the same expressive power as **LAL**, at least when algorithms on binary words are concerned.

Recall that any word $w = i_1 \cdots i_n$ in $\{0,1\}^*$ can be encoded both as a lambda-term and as a λLA-term:

$$\underline{w} = \lambda f_0.\lambda f_1.\lambda x.f_{i_1}(f_{i_2}\dots(f_{i_n}x)\dots) : \mathbf{W}$$
$$\underline{w}_L = \lambda f_0.\mathsf{let}\ f_0\ \mathsf{be}\ !g_0\ \mathsf{in}\ (\mathsf{let}\ f_1\ \mathsf{be}\ !g_1\ \mathsf{in}\ \S(\lambda x.g_{i_1}(g_{i_2}\dots(g_{i_n}x)\dots))) : \mathbf{W}_L$$

The former can be obtained from the latter by applying the erasure operator: $\underline{w} = (\underline{w}_L)^-$ for every $w \in \{0,1\}^*$. For the converse direction, we have the following term $\mathsf{dtol}_{\mathbf{W}} : \mathbf{W} \multimap \mathbf{W}_L^\bullet$ which maps $\underline{w}$ to $\underline{w}_L^\bullet$:

$$\mathsf{dtol}_{\mathbf{W}} = \lambda x^{\mathbf{W}}.\lambda f_0.f_0(\lambda g_0.\lambda f_1.f_1(\lambda g_1.xg_0g_1)).$$

(Here we cannot use $\mathsf{en}_{\mathbf{W}}$, as it returns a slightly different term when applied to $\underline{w}$.) To see that $\mathsf{dtol}_{\mathbf{W}}$ is surely of type $\mathbf{W} \multimap \mathbf{W}_L^\bullet$, note that

$$\mathbf{W}_L^\bullet = \forall\alpha.(!(\alpha \multimap \alpha))^\bullet \multimap (!(\alpha \multimap \alpha))^\bullet \multimap \S(\alpha \multimap \alpha),$$

where $(!(\alpha \multimap \alpha))^\bullet = (\forall\beta.((\alpha \multimap \alpha) \Rightarrow \beta) \multimap \beta)$. Hence the variables f_0, f_1 must be typed $(\forall\beta.((\alpha \multimap \alpha) \Rightarrow \beta) \multimap \beta)$. By instantiating β into $\S(\alpha \multimap \alpha)$ for the type of f_1 and into $(!(\alpha \multimap \alpha))^\bullet \multimap \S(\alpha \multimap \alpha)$ for the type of f_0 (and assuming that g_0 and g_1 are of type $\alpha \multimap \alpha$), one obtains:

$$
\begin{aligned}
xg_0g_1 &: \S(\alpha \multimap \alpha)\\
\lambda g_1.xg_0g_1 &: (\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)\\
f_1(\lambda g_1.xg_0g_1) &: \S(\alpha \multimap \alpha)\\
\lambda f_1.f_1(\lambda g_1.xg_0g_1) &: (!(\alpha \multimap \alpha))^\bullet \multimap \S(\alpha \multimap \alpha)\\
\lambda g_0.\lambda f_1.f_1(\lambda g_1.xg_0g_1) &: (\alpha \multimap \alpha) \Rightarrow (!(\alpha \multimap \alpha))^\bullet \multimap \S(\alpha \multimap \alpha)\\
f_0(\lambda g_0.\lambda f_1.f_1(\lambda g_1.xg_0g_1)) &: (!(\alpha \multimap \alpha))^\bullet \multimap \S(\alpha \multimap \alpha)\\
\lambda f_0.f_0(\lambda g_0.\lambda f_1.f_1(\lambda g_1.xg_0g_1)) &: (!(\alpha \multimap \alpha))^\bullet \multimap (!(\alpha \multimap \alpha))^\bullet \multimap \S(\alpha \multimap \alpha).
\end{aligned}
$$

It is easy to see that $\mathsf{dtol}_{\mathbf{W}}(\underline{w})$ reduces to $(\underline{w}_L)^\bullet$, by noting:

$$(\underline{w}_L)^\bullet = \lambda f_0.f_0(\lambda g_0.\lambda f_1.f_1(\lambda g_1.\lambda x.g_{i_1}(g_{i_2}\dots(g_{i_n}x)\dots))).$$

Theorem 32 (Preservation of representable functions) *Suppose that a function $f : \{0,1\}^* \to \{0,1\}^*$ is represented by a λLA-term M with $\vdash_{LAL} M : \mathbf{W}_L \multimap \S^n\mathbf{W}_L$ for some $n \geq 0$. Define a lambda-term t by*

$$t = \lambda x.\mathsf{de}_{\S^n\mathbf{W}}(M^\bullet \mathsf{dtol}_{\mathbf{W}}(x)).$$

Then we have that $\vdash_{DLAL} t : \mathbf{W} \multimap \S^n\mathbf{W}$ and t also represents f.

Proof. The term t gets type $\mathbf{W} \multimap \S^n\mathbf{W}$ in **DLAL** as follows (notice that $\mathbf{W}^\star = \mathbf{W}_L$), using Proposition 24 and Lemma 27:

$$\dfrac{\dfrac{\dfrac{;\vdash M^\bullet : \mathbf{W}^{\star\bullet} \multimap \S^n\mathbf{W}^{\star\bullet} \quad ; x : \mathbf{W} \vdash \mathsf{dtol}_{\mathbf{W}}(x) : \mathbf{W}^{\star\bullet}}{; x : \mathbf{W} \vdash M^\bullet \mathsf{dtol}_{\mathbf{W}}(x) : \S^n\mathbf{W}^{\star\bullet}}}{; x : \mathbf{W} \vdash \mathsf{de}_{\S^n\mathbf{W}}(M^\bullet \mathsf{dtol}_{\mathbf{W}}(x)) : \S^n\mathbf{W}}}{\vdash t : \mathbf{W} \multimap \S^n\mathbf{W}}\ .$$

Since M represents f, $M\underline{w}_L$ reduces to $\underline{f(w)}_L$ for any $w \in \{0,1\}^*$. This implies that $(M\underline{w}_L)^- =_{\beta\eta} (\underline{f(w)}_L)^-$. On the other hand,

$$t\underline{w} =_{\beta\eta} \mathsf{de}_{\S^n\mathbf{W}}(M^\bullet \mathsf{dtol}_{\mathbf{W}}(\underline{w})) =_{\beta\eta} \mathsf{de}_{\S^n\mathbf{W}}(M^\bullet(\underline{w}_L)^\bullet) = \mathsf{de}_{\S^n\mathbf{W}}((M\underline{w}_L)^\bullet).$$

Applying Corollary 31, as $\S^n\mathbf{W}$ is Π_1, we thus get:

$$t\underline{w} =_{\beta\eta} (M\underline{w}_L)^- =_{\beta\eta} (\underline{f(w)}_L)^- = \underline{f(w)}.$$

Theorem 9 is then a direct consequence of the above theorem together with the following fact known from [39] (the depth bound is sharpened):

Theorem 33 (FP completeness of LAL) *If a function $f : \{0,1\}^\star \to \{0,1\}^\star$ is computable in time $O(n^{2^d})$ by a multi-tape Turing machine for some d, then there exists a λLA-term M such that $\vdash_{LAL} M : \mathbf{W}_L \multimap \S^{2d+2}\mathbf{W}_L$ and M represents f.*

Proof (Sketch). Following the idea of [2], let **Conf** be the **LAL**-type

$$\forall\alpha.!(\alpha \multimap \alpha)^m \multimap \S((\alpha \multimap \alpha)^{2n} \otimes \mathbf{B}^k),$$

which serves as a type for the configurations of a given Turing machine with m symbols, n tapes and k states. As in [2], one can define the following λLA-terms:

- $\mathsf{trans} : \mathbf{Conf} \multimap \mathbf{Conf}$ for one-step transition of a Turing Machine;
- $\mathsf{init} : \mathbf{W}_L \multimap \mathbf{Conf}$ for initialization;
- $\mathsf{output} : \mathbf{Conf} \multimap \mathbf{W}_L$ for output extraction;
- $\mathsf{length} : \mathbf{W}_L \multimap \mathbf{N}_L$ for the length map;
- $\mathsf{dupl} : \mathbf{W}_L \multimap \S(\mathbf{W}_L \otimes \mathbf{W}_L)$ for duplication.

Furthermore, Proposition 11 yields a term $\mathsf{prod}(2^d) : \mathbf{N}_L \multimap \S^{2d}\mathbf{N}_L$ for the map $n \mapsto n^{2^d}$. By applying $\mathsf{iter}_{\mathbf{Conf}}$ to trans and init, we obtain

$$x : \mathbf{N}_L, y : \S\mathbf{W}_L \vdash_{LAL} M_0 : \S\mathbf{Conf}.$$

The term M_0 transforms a given input y into an initial configuration and iterates the one-step transition x times. By applying the rule (§ i) $2d$ times and composing the outcome with $\mathsf{prod}(2^d)$, we obtain

$$x : \mathbf{N}_L, y : \S^{2d+1}\mathbf{W}_L \vdash_{LAL} M_1 : \S^{2d+1}\mathbf{Conf},$$

which iterates the transition x^{2^d} times. Finally, by using output, length, a variant of coer_1 and dupl, we obtain the desired term $M : \mathbf{W}_L \multimap \S^{2d+2}\mathbf{W}_L$ representing the function f.

We remark that it is possible to improve the type of M as $\mathbf{W}_L \multimap \S^{2d+1}\mathbf{W}_L$ for $d \geq 1$, by combining $\mathsf{prod}(2^d)$ and dupl in a clever way.

8 Conclusion and perspectives

We have presented a polymorphic type system for lambda calculus which guarantees that typed terms can be reduced in a polynomial number of steps, and in polynomial time. This system, **DLAL**, has been designed as a subsystem of **LAL**. It offers the advantage of recasting the main ideas and achievements of light linear logic into a plain lambda calculus setting.

We have also shown that **DLAL** is not more constrained computationally than **LAL**, by describing a generic encoding of **LAL** typed terms into **DLAL** ones, which preserves the denotation of functions over binary integers. This has shown that **DLAL** is complete for the class of polynomial time functions.

We think that the techniques we used to relate **DLAL** and **LAL**, based on logical relations, could probably be applied to similar situations in other linear logic like or modal type systems, to encode the general language into a smaller subsystem.

Finally the interest of **DLAL** has also been confirmed by another work, [3,4], which showed that type inference in **DLAL** for system **F** terms can be performed in polynomial time, using an algorithm based on constraints solving.

Other approaches to characterization of complexity classes in lambda calculus have considered restrictions on type orders (see [22,29,36]); it would be interesting to examine the possible relations between this line of work and the present setting based on linear logic.

References

[1] A. Asperti. Light affine logic. In *Proceedings of LICS'98*, pages 300–308. IEEE Computer Society, 1998.

[2] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Transactions on Computational Logic*, 3(1):1–39, 2002.

[3] V. Atassi, P. Baillot, and K. Terui. Verification of Ptime reducibility for system F terms via Dual Light Affine Logic. In *Proceedings of CSL'06*, volume 4207 of *LNCS*, pages 150–166. Springer, 2006.

[4] V. Atassi, P. Baillot, and K. Terui. Verification of Ptime reducibility for system F terms: type inference in Dual Light Affine Logic. *Logical Methods in Computer Science*, 3(4):1–32, 2007. Special issue on CSL'06.

[5] P. Baillot. Stratified coherence spaces: a denotational semantics for Light Linear Logic. *Theoretical Computer Science*, 318(1-2):29–55, 2004.

[6] P. Baillot. Type inference for Light Affine Logic via constraints on words. *Theoretical Computer Science*, 328(3):289–323, 2004.

[7] P. Baillot and M. Pedicini. Elementary complexity and geometry of interaction. *Fundamenta Informaticae*, 45(1-2):1–31, 2001.

[8] P. Baillot and K. Terui. Light types for polynomial time computation in lambda-calculus. In *Proceedings of LICS'04*, pages 266–275, 2004.

[9] P. Baillot and K. Terui. Light types for polynomial time computation in lambda-calculus (long version). Technical Report cs.LO/0402059, arXiv, april 2004. available from http://arXiv.org.

[10] P. Baillot and K. Terui. A feasible algorithm for typing in Elementary Affine Logic. In *Proceedings of TLCA'05*, volume 3461 of *LNCS*, pages 55–70. Springer, 2005.

[11] A. Barber and G. Plotkin. Dual intuitionistic linear logic. Technical report, LFCS, University of Edinburgh, 1997.

[12] S. Bellantoni and S. Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[13] G. M. Bierman and V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3):383–416, 2000.

[14] P. Coppola, U. Dal Lago, and S. Ronchi Della Rocca. Elementary affine logic and the call-by-value lambda calculus. In *Proceedings of TLCA'05*, volume 3461 of *LNCS*, pages 131–145, 2005.

[15] P. Coppola and S. Martini. Optimizing optimal reduction. a type inference algorithm for elementary affine logic. *ACM Transactions on Computational Logic*, 7(2):219–260, 2006.

[16] P. Coppola and S. Ronchi Della Rocca. Principal typing for lambda calculus in elementary affine logic. *Fundamenta Informaticae*, 65(1-2):87–112, 2005.

[17] U. Dal Lago. Context semantics, linear logic and computational complexity.. In *Proceedings of LICS'06*, pages 169–178. IEEE Computer Society, 2006.

[18] U. Dal Lago and M. Hofmann. Quantitative models and implicit complexity. In *Proceedings of FSTTCS'05*, volume 3821 of *LNCS*, pages 189–200. Springer, 2005.

[19] V. de Paiva, R. Goré, and M. Mendler. Preface to the special issue on intuitionistic modal logic and application. *Journal of Logic and Computation*, 14(4), 2004.

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

[21] M. Hasegawa. Classical linear logic of implications. *Mathematical Structures in Computer Science*, 15(2):323–342, 2005.

[22] G. Hillebrand and P. C. Kannelakis. On the expressive power of simply typed and let-polymorphic lambda calculi. In *Proceedings of LICS'96*, pages 253–263. IEEE Computer Society, 1996.

[23] M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.

[24] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.

[25] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of ACM POPL'03*, 2003.

[26] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1–2):163–180, 2004.

[27] O. Laurent and L. Tortora de Falco. Obsessional cliques: a semantic characterization of bounded time complexity. In *Proceedings of LICS'06*, pages 179–188. IEEE Computer Society, 2006.

[28] D. Leivant. Predicative recurrence and computational complexity I: word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhauser, 1994.

[29] D. Leivant. Calibrating computational feasibility by abstraction rank. In *Proceedings LICS'02*, pages 345–353. IEEE Computer Society, 2002.

[30] H. Mairson and P. M. Neergard. LAL is square: Representation and expressiveness in light affine logic, 2002. Presented at the 4th International Workshop on Implicit Computational Complexity.

[31] F. Maurel. Nondederministic Light Logics and NP-time. In *Proceedings of TLCA'03*, LNCS. Springer, 2003.

[32] D. Mazza. Linear logic and polynomial time. *Mathematical Structures in Computer Science*, 16(6):947–988, 2006.

[33] A. S. Murawski and C.-H. L. Ong. On an interpretation of safe recursion in light affine logic. *Theoretical Computer Science*, 318(1-2):197–223, 2004.
[34] F. Pfenning and H.-C. Wong. On a modal lambda calculus for S4. *Electric Notes on Theoretical Computer Science*, 1, 1995.
[35] G. D. Plotkin. Type theory and recursion (extended abstract). In *Proceedings of LICS'93*, page 374, 1993.
[36] A. Schubert. The complexity of beta-reduction in low orders. In *Proceedings of TLCA'01*, LNCS, pages 400–414. Springer, 2001.
[37] K. Terui. Light affine lambda calculus and polytime strong normalization. In *Proceedings of LICS'01*, pages 209–220. IEEE Computer Society, 2001.
[38] K. Terui. Light affine set theory: a naive set theory of polynomial time. *Studia Logica*, 77:9–40, 2004.
[39] K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3):253–280, 2007.