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. $\beta$ reduction is problematic: subject reduction fails and no polynomial bound holds on the number of $\beta$ 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 □ is used only in combination with arrows, in types □A → B; in this situation types of the form A → □B for instance are not considered. This is systematized in Girard’s embedding of intuitionistic logic in linear logic by $(A → B)^* = !A^* → B^*$.

**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 ($\otimes$) 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 $(!A)^* = \forall \alpha.(A^* \Rightarrow \alpha) \otimes \alpha$ [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. $\beta$ reduction;
- Ptime soundness w.r.t. $\beta$ 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 \textbf{DLAL} in Section 3. Then in Section 4 we state the main properties of \textbf{DLAL} and discuss the use of iteration on examples. Section 5 is devoted to an embedding of \textbf{DLAL} into \textbf{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 \textbf{LAL} into \textbf{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 \( \beta \) reduction on lambda-terms.

#### 2.1 Light affine logic

The language \( \mathcal{L}_{LAL} \) of \textbf{LAL} types is given by:

\[
A, B ::= \alpha \mid A \rightarrow B \mid !A \mid \& A \mid \forall \alpha.A.
\]

We omit the connective \( \otimes \) which is definable. We will write \( \dagger \) 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 \textit{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 \textbf{LAL} with respect to linear logic is the condition under which one
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): ($*$) $\alpha$ does not appear free in $\Gamma$.
- the (! i) rule can also be applied to a judgement of the form $\vdash u : A$ ($u$ has no free variable).

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

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

Now, light affine logic enjoys the following property:

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

\[
y_i :! A \rightarrow_<> A \vdash_{LAL} \lambda x . y_i x x :! A \rightarrow_<> A,
\]

\[
z :! A \vdash_{LAL} z :! A.
\]

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

\[
y_1 :! A \rightarrow_<> A \rightarrow_<> A, \ldots, y_n :! A \rightarrow_<> A \rightarrow_<> A, z :! A \vdash_{LAL} (\lambda x . y_1 x x) (\cdots (\lambda x . y_n x x) z \cdots) :! A.
\]

Using ($\S$ i) and (Cntr) we finally get:

\[
y :(! A \rightarrow_<> A \rightarrow_<> A), z :!! A \vdash_{LAL} (\lambda x . y x x)^n z :!! A.
\]

Denote by $t_n$ the term $(\lambda x . y x x)^n z$ 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 \( z : !A, y : !A \Rightarrow !A \vdash_{LAL} yzz : !A \). Now make \( n \) copies of it and compose them by \((! e)\); each time \((! e)\) is applied, the term size is doubled. Finally, by applying \((\S \ i)\) and \((\text{Cntr})\) as before, we obtain a linear size derivation for \( y : (!A \Rightarrow !A) \Rightarrow !A \vdash_{LAL} u_n : \S !A \).

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 \( \text{LAL} \) and polynomial time strong normalization [39]. However it is not as simple as lambda calculus; in particular it includes new constructions \(!()\), \(\S()\) and \(\text{let }() \text{ be }() \text{ in }()\) 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 \( A \Rightarrow !B \). Bang \(!\) is used only in the form \(!A \Rightarrow !B\), 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 \( A \Rightarrow !B \).

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 $L_{DLAL}$ of DLAL types is given by:

$$A, B ::= \alpha \mid A \rightarrow B \mid A \Rightarrow B \mid \mathbf{\$}A \mid \forall \alpha.A.$$ 

Let us now define $\Pi_1$ and $\Sigma_1$ types. The class of $\Pi_1$ (resp. $\Sigma_1$) types is the least subset of DLAL types satisfying:

- any atomic type $\alpha$ is $\Pi_1$ (resp. $\Sigma_1$),
- if $A$ is $\Sigma_1$ (resp. $\Pi_1$) and $B$ is $\Pi_1$ (resp. $\Sigma_1$), then $A \rightarrow B$ and $A \Rightarrow B$ are $\Pi_1$ (resp. $\Sigma_1$),
- if $A$ is $\Pi_1$ (resp. $\Sigma_1$) then $\mathbf{\$}A$ is $\Pi_1$ (resp. $\Sigma_1$),
- if $A$ is $\Pi_1$ then so is $\forall \alpha.A$.

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

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

Let $L_{DLAL}^*$ denote the image of $L_{DLAL}$ by $(.)^*$, and $(.)^- : L_{DLAL}^* \rightarrow L_{DLAL}$ stand for the converse map of $(.)^*$.

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

- for $\forall i$: (*) $\alpha$ does not appear free in $\Gamma, \Delta$.
- 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 $\text{no}(x,t) \leq 1$.

We can make the following remarks on DLAL rules:
\[ \frac{x : A \vdash x : A}{(\text{Id})} \]

\[ \frac{\Gamma; \Delta; x : A \vdash t : B}{\Gamma; \Delta \vdash \lambda x . t : A \rightarrow B} \quad (\rightarrow \text{ i}) \]

\[ \frac{\Gamma; \Delta \vdash \lambda x . t : A \Rightarrow B}{\Gamma; \Delta; x : A \vdash t : B} \quad (\Rightarrow \text{ i}) \]

\[ \frac{\Gamma_1; \Delta_1 \vdash t : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t : A} \quad (\text{Weak}) \]

\[ \frac{\Gamma, \Delta \vdash t : A}{\Gamma; \Delta ; \Delta \vdash t : \Delta} \quad (\text{§ i}) \]

\[ \frac{\Gamma; \Delta \vdash t : A}{\Gamma; \Delta; \Delta \vdash t : \forall \alpha . A} \quad (\text{\forall i} \ (\ast)) \]

\[ \frac{\Gamma_1; \Delta_1 \vdash t : A \rightarrow B ; \Gamma_2; \Delta_2 \vdash u : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash tu : B} \quad (\rightarrow \text{ e}) \]

\[ \frac{\Gamma; \Delta \vdash t : A \Rightarrow B ; z : C \vdash u : A}{\Gamma; \Delta; z : C \vdash tu : B} \quad (\Rightarrow \text{ e}) \]

\[ \frac{\Gamma_1; \Delta_1 \vdash t : A \Rightarrow B}{\Gamma_1; \Delta_2 ; x_1 : A, x_2 : A, \Gamma; \Delta \vdash t : B}{\Gamma_1; \Delta_1, \Delta_2 \vdash t [x_1 / x_1, x_2 / x_2] : B} \quad (\text{Cntr}) \]

\[ \frac{x_1 : A, x_2 : A, \Gamma; \Delta \vdash t : B}{\Gamma; \Delta \vdash \lambda x_1 . t [x_1 / x_1, x_2 / x_2] : B} \quad (\text{§ e}) \]

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

### 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 (\(\rightarrow\) 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 \(\Gamma; \Delta \vdash_{\text{DLAL}} t : A\) 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 (\(\cdot\))\(^*\) translation. As a consequence, DLAL can be considered as a subsystem of LAL. Let us make this point precise.

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

- any formula belongs to \(\mathcal{L}_{\text{DLAL}}^+\),
- any instantiation formula \(B\) of a (\(\forall\) e) rule in \(\mathcal{D}\) belongs to \(\mathcal{L}_{\text{DLAL}}^+\).
We stress that $L_{DLAL^*}$ and $L_{DLAL^+}$ are sublanguages of $L_{LAL}$ and not of $L_{DLAL}$. Hence when writing $\Gamma \vdash_{DLAL^*} t : A$, we are handling an $L_{LAL}$ 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 $!\Gamma^*, \Delta^* \vdash_{DLAL^*} t : A^*$.

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 $\mathcal{D}^*$ the $L_{LAL}$ 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 ($\xi$ ii) 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 ($\xi$ 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 $\mathcal{D}^*$ in $L_{LAL}$, 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 $L_{LAL}$ can be directly adapted to $DLAL$. For instance, we have the following data types for booleans, unary integers and binary words in $L_{LAL}$:

\[
B = \forall \alpha. \alpha \to \alpha \to \alpha,
\]
\[
N_L = \forall \alpha. (\alpha \to \alpha) \to \xi(\alpha \to \alpha),
\]
\[
W_L = \forall \alpha. (\alpha \to \alpha) \to \xi(\alpha \to \alpha) \to \xi(\alpha \to \alpha).
\]

The type $B$ is also available in $DLAL$ as it stands, while for the latter two, we have $N$ and $W$ such that $N^* = N_L$ and $W^* = W_L$ in $DLAL$:

\[
N = \forall \alpha. (\alpha \to \alpha) \Rightarrow \xi(\alpha \to \alpha),
\]
\[
W = \forall \alpha. (\alpha \to \alpha) \Rightarrow (\alpha \to \alpha) \Rightarrow \xi(\alpha \to \alpha).
\]

The inhabitants of $B$, $N$ and type $W$ are the familiar Church codings of booleans, integers and words:
$$\Gamma_1; \Delta_1 \vdash t_1 : A_1 \quad \Gamma_2; \Delta_2 \vdash t_2 : A_2$$

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

$$\Gamma_1; \Delta_1 \vdash u : A_1 \otimes A_2 \quad \Gamma_2; x_1 : A_1, x_2 : A_2, \Delta_2 \vdash t \colon B$$

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

Fig. 3. Derived rules

$$\hat{i} = \lambda x_0. \lambda x_1.x_i,$$

$$\hat{n} = \lambda f. \lambda x.f(f \ldots (f x) \ldots),$$

$$\hat{w} = \lambda f_0. \lambda f_1. \lambda x.f_i(f_{i_2} \ldots (f_{i_n}x) \ldots),$$

with \( i \in \{0, 1\}, n \in \mathbb{N} \) and \( w = i_1i_2 \cdots i_n \in \{0, 1\}^* \). The following terms for addition and multiplication on Church integers are typable in DLAL:

\[
\begin{align*}
\text{add} &= \lambda n. \lambda m. \lambda f. \lambda x. n(f(mf x)) : N \rightarrow N \rightarrow N, \\
\text{mult} &= \lambda n. \lambda m. m(\lambda y. \text{add } n y)0 : N \Rightarrow N \rightarrow \mathbb{N}.
\end{align*}
\]

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

\[ A \otimes B = \forall \alpha.((A \rightarrow B \rightarrow \alpha) \rightarrow \alpha). \]

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

\[ t_1 \otimes t_2 = \lambda x. xt_1t_2, \]

\[ \text{let } u \text{ be } x_1 \otimes x_2 \text{ in } t = u(\lambda x_1. \lambda x_2. t). \]

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 $t_0 \rightarrow t_1$, 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 $D$ of depth $d$ in **DLAL**. Then $t$ reduces to the normal form in at most $O(|t|^{2d})$ reduction steps and in time $O(|t|^{2d+2})$ on a multi-tape Turing machine. This result holds independently of which reduction strategy we take.

The bound $O(|t|^{2d})$ 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 $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is computable in time $O(n^{2d})$ by a multi-tape Turing machine for some $d$, then there exists a lambda-term $t$ such that $\vdash_{DLAL} t : W \rightarrow ^{2d+2}W$ and $t$ represents $f$.

The depth of the result type $^{2d+2}W$ 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 $\mathbb{N}$ and the following data type corresponding to lists over $A$:

$$L(A) = \forall \alpha. (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha),$$

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

$$\text{iter}_B : (B \rightarrow B) \Rightarrow \#B \rightarrow \mathbb{N} \rightarrow \#B,$$
$$\text{fold}_B : (A \rightarrow B \rightarrow B) \Rightarrow \#B \rightarrow L(A) \rightarrow \#B.$$

They are defined by the same term:

$$\text{iter}_B = \lambda F. \lambda b. \lambda n. n F b,$$
$$\text{fold}_B = \lambda F. \lambda b. \lambda l. l F b.$$

Note that with the $\text{iter}_B$ scheme for instance, if we iterate on type $B = \mathbb{N}$ a function $\mathbb{N} \rightarrow \mathbb{N}$ we obtain a term of type $\mathbb{N} \rightarrow \#\mathbb{N}$, 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 $\#\mathbb{N} \rightarrow \mathbb{N} \rightarrow \#\mathbb{N}$. However recall that we have given previously a term for addition with type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$.

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 $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$?

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

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

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:

$$; f : (\alpha \rightarrow \alpha) \vdash \lambda k. \lambda y. f(ky) : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$
Denote this term by $\text{osucc}_f$, indicating the free variable $f$. The typing judgement claimed above can be derived in the following way:

\[
\frac{
\frac{
\frac{
\frac{
\frac{
\frac{
\frac{
f : \alpha \rightarrow \alpha \vdash f : \alpha \rightarrow \alpha}{k : \alpha \rightarrow \alpha \vdash k : \alpha \rightarrow \alpha}{y : \alpha \vdash y : \alpha}}{\frac{k : \alpha \rightarrow \alpha, y : \alpha \vdash ky : \alpha}{\frac{f : (\alpha \rightarrow \alpha), k : \alpha \rightarrow \alpha, y : \alpha \vdash f(ky) : \alpha}{\frac{f : (\alpha \rightarrow \alpha) \vdash \text{osucc}_f : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}}}}}}}{\frac{f : \alpha \rightarrow \alpha, f' : \alpha \rightarrow \alpha; n : \mathbb{N}, m : \mathbb{N} \vdash \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f') n : \varepsilon(\alpha \rightarrow \alpha)}{\frac{f : \alpha \rightarrow \alpha; n : \mathbb{N}, m : \mathbb{N} \vdash \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f) n : \varepsilon(\alpha \rightarrow \alpha)}{\frac{n : \mathbb{N}, m : \mathbb{N} \vdash \lambda f. \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f) n : (\alpha \rightarrow \alpha) \Rightarrow \varepsilon(\alpha \rightarrow \alpha)}{\frac{n : \mathbb{N}, m : \mathbb{N} \vdash \lambda f. \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f) n : \mathbb{N}}}}}}
\]

By applying $\text{iter}_B$ on the type $B = \alpha \rightarrow \alpha$ of open Church integers, we get:

\[
\frac{f : \alpha \rightarrow \alpha, f' : \alpha \rightarrow \alpha; n : \mathbb{N}, m : \mathbb{N} \vdash \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f') n : \varepsilon(\alpha \rightarrow \alpha)}{\frac{f : \alpha \rightarrow \alpha; n : \mathbb{N}, m : \mathbb{N} \vdash \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f) n : \varepsilon(\alpha \rightarrow \alpha)}{\frac{n : \mathbb{N}, m : \mathbb{N} \vdash \lambda f. \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f) n : (\alpha \rightarrow \alpha) \Rightarrow \varepsilon(\alpha \rightarrow \alpha)}{\frac{n : \mathbb{N}, m : \mathbb{N} \vdash \lambda f. \text{iter}_{\alpha \rightarrow \alpha} \text{osucc}_f (m, f) n : \mathbb{N}}}}
\]

Note that a crucial point for the type derivation to be valid is the $(\Rightarrow \varepsilon)$ rule, which forces $\text{osucc}$ to have at most one occurrence of free variable. This restricts the $\text{osucc}$ 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:

\[
\text{comp} : A \rightarrow A \rightarrow A \otimes A, \quad \text{with } \text{comp} a_1 a_2 \rightarrow a_1 \otimes a_2 \quad \text{if } a_1 \leq a_2
\]

\[
\quad a_2 \otimes a_1 \quad \text{if } a_2 < a_1
\]

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

$$hcomp_g = \lambda \alpha^A f^B a^A. \text{let } (\text{comp } a a') \text{ be } a_1 \otimes a_2 \text{ in } \lambda x^\alpha. g^B a_1^A (fa_2 x)^\alpha : A \rightarrow B \rightarrow B$$

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

$$\text{insert } = \lambda a_0^{\mathcal{L}(A)}. \lambda y^B. (\text{fold}_B hcomp_g g l) a_0 : \mathcal{L}(A) \rightarrow \mathcal{L}(A)$$

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

$$\text{sort } = \lambda l^{\mathcal{L}(\mathcal{L}(A))}. \text{fold}_{\mathcal{L}(\mathcal{L}(A))} \text{ insert } \text{ nil } : \mathcal{L}(\mathcal{L}(A)) \rightarrow \mathcal{L}(\mathcal{L}(A))$$

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

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

$$; A \rightarrow \alpha \rightarrow \alpha \vdash A \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha); \Gamma, A \rightarrow \alpha \rightarrow \alpha; \Delta \vdash \mathcal{L}(A) \rightarrow \mathcal{L}(A)$$

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 $C \rightarrow (\alpha \rightarrow \alpha)$:

$$; f : A \rightarrow \alpha \rightarrow \alpha \vdash t_f : A \rightarrow (C \rightarrow \alpha \rightarrow \alpha) \rightarrow (C \rightarrow \alpha \rightarrow \alpha); \Gamma, f : A \rightarrow \alpha \rightarrow \alpha; \Delta \vdash u_f : \mathcal{L}(C \rightarrow \alpha \rightarrow \alpha)$$

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

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

$$f : A \rightarrow \alpha \rightarrow \alpha, \Gamma; \Delta, l : \mathcal{L}(A) \vdash \text{fold}_{C \rightarrow \alpha \rightarrow \alpha} t_f u_f l : \mathcal{L}(C \rightarrow \alpha \rightarrow \alpha)$$

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 \( \text{coer}_b : B \rightarrow §B \) such that \( \text{coer}_b i \rightarrow^* i \) for \( i \in \{0, 1\} \).
2. There is a lambda-term \( \text{coer}_1 : N \rightarrow §N \) such that \( \text{coer}_1 n \rightarrow^* n \) for every integer \( n \).
3. For any type \( A \), there is a lambda-term \( \text{coer}_2 : §(N \Rightarrow A) \rightarrow (N \Rightarrow §A) \) such that \( \text{coer}_2 t_n \rightarrow^* t_n \) for every term \( t \) and every integer \( n \).

**Proof.**

1. Let \( \text{coer}_b = \lambda x.x0 \).
2. Let \( \text{coer}_1 = \text{iter}_N \text{succ} 0 \), where \( \text{succ} \) is the usual successor \( \lambda n.\lambda fx.f(nfx) : N \rightarrow N \).
3. We have ‘lifted’ versions of the successor and the zero:

\[
\text{l_succ} = \lambda fx.f(\text{succ} x) : (N \Rightarrow A) \rightarrow (N \Rightarrow A)
\]
\[
\text{l_zero} = \lambda f.f00 : §(N \Rightarrow A) \rightarrow §A.
\]

We therefore have

\[
\text{coer}_2 = \lambda fn.\text{l_zero}(\text{iter}_{N \Rightarrow A} \text{l_succ} f n) : §(N \Rightarrow A) \rightarrow (N \Rightarrow §A).
\]

Given a term \( t : §(N \Rightarrow A) \) and an integer \( n \), it works as follows:

\[
\text{coer}_2 t_n \rightarrow^* \text{l_zero}(\text{iter} \text{l_succ} t n)
\]
\[
\rightarrow^* \text{l_zero}(\text{l_succ}^n t)
\]
\[
\rightarrow^* \text{l_zero}(\lambda x.t(\text{succ}^n x))
\]
\[
\rightarrow^* t(\text{succ}^n 0) \rightarrow^* t_n.
\]

As a slight variant of \( \text{coer}_2 \), we have a contraction map \( \text{cont} : §(N \Rightarrow N \rightarrow A) \rightarrow (N \rightarrow §A) \) defined by

\[
\text{c_succ} = \lambda f xy.f(\text{succ} x)(\text{succ} y) : (N \Rightarrow N \rightarrow A) \rightarrow (N \Rightarrow N \rightarrow A),
\]
\[
\text{c_zero} = \lambda f.f00 : §(N \Rightarrow N \rightarrow A) \rightarrow §A,
\]
\[
\text{cont} = \lambda fn.\text{c_zero}(\text{iter}_{N \Rightarrow N \rightarrow A} \text{c_succ} f n) : §(N \Rightarrow N \rightarrow A) \rightarrow (N \rightarrow §A).
\]
By applying \texttt{cont} to the multiplication function \texttt{mult} : \mathbb{N} \Rightarrow \mathbb{N} \rightarrow \mathbb{N}, the squaring function on unary integers can be represented: \texttt{square} = \texttt{cont(mult)} : \mathbb{N} \rightarrow \mathbb{N}. We therefore obtain:

\textbf{Proposition 11} There is a closed term of type \( \mathbb{N} \rightarrow \mathbb{N} \) representing the function \( n^2 \) 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 \( L_{DLAL^*} \). Let us begin with the ordinary substitution lemma.

\textbf{Lemma 12 (Substitution)} We consider derivations in DLAL.

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

\textbf{Lemma 13} If \( A, H \in L_{DLAL^*} \), we have \( A[H/\alpha]^\sim = A[H^-/\alpha] \).

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

\textbf{Lemma 14}

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

\textbf{Proof.} We prove these statements by a single induction over derivations in DLAL\(^*\).

Let \( D \) be a DLAL\(^*\) derivation of \( \Gamma, \Delta \vdash t : C \) with \( \Gamma, \Delta \in L_{DLAL^*} \). We have to prove that: if \( C \in L_{DLAL^*} \) then claim (1) holds; otherwise if \( C = !A \) and \( A \in L_{DLAL^*} \) then claim (2) holds. Consider the last rule of \( D \):

(Case 1) The derivation consists of the identity axiom. Straightforward.
(Case 2) The last rule is \((-o \ i)\):

\[
\frac{y : A_1, !\Gamma, \Delta \vdash v : A_2}{!\Gamma, \Delta \vdash \lambda y. v : A_1 \rightarrow A_2} \quad (-o \ i)
\]

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

- if \(A_1 \in L_{DLAL}^*\):
  \[
  \frac{!\Gamma ; y : A_1^-, \Delta^- \vdash_{DLAL} v : A_2^-}{!\Gamma; \Delta^- \vdash_{DLAL} \lambda y. v : A_1^- \rightarrow A_2^-}
  \]

  and \(A_1^- \rightarrow A_2^- = (A_1 \rightarrow A_2)^-\).

- if \(A_1 = !A'_1\), with \(A'_1 \in L_{DLAL}^*\):
  \[
  \frac{!\Gamma; \Delta^- \vdash_{DLAL} \lambda y. v : A_1' \rightarrow A_2^-}{!\Gamma; \Delta^- \vdash_{DLAL} A_1' \rightarrow A_2^-}
  \]

  and \(A_1' \Rightarrow A_2^- = (A_1 \rightarrow A_2)^-\).

(Case 3) The last rule is \((-o \ e)\):

\[
\frac{!\Gamma_1, \Delta_1 \vdash t_1 : C \rightarrow A \quad !\Gamma_2, \Delta_2 \vdash t_2 : C}{!\Gamma_1, !\Gamma_2, \Delta_1, \Delta_2 \vdash t_1 t_2 : A} \quad (-o \ e)
\]

As by assumption we have \(C \rightarrow A \in L_{DLAL}^+\), we get that \(A \in L_{DLAL}^*\). Thus we are in the case of claim (1). We now distinguish two possible subcases:

- first subcase: \(C \in L_{DLAL}^*\). Then \((C \rightarrow A)^- = C^- \rightarrow A^-\). We apply the i.h. to the two subderivations, which are both in the case of claim (1), and apply the \(DLAL\) rule:

\[
\frac{\Gamma_1^-; \Delta_1^- \vdash t_1 : C^- \rightarrow A^- \quad \Gamma_2^-; \Delta_2^- \vdash t_2 : C^-}{\Gamma_1^-; \Gamma_2^-; \Delta_1^-; \Delta_2^- \vdash t_1 t_2 : A^-} \quad (-o \ e)
\]

- second subcase: \(C = !C',\) with \(C' \in L_{DLAL}^*\). Then \((C \rightarrow A)^- = C'^- \Rightarrow A^-\). We apply the i.h. to the l.h.s. premise, which is in the case of claim (1), and a \((\Rightarrow \ e)\) rule in \(DLAL\), to get:

\[
\frac{\Gamma_1^-; \Delta_1^- \vdash t_1 : C'^- \Rightarrow A^- \quad z : C^- \vdash z : C^-}{z : C^-; \Gamma_1^-; \Delta_1^- \vdash t_1 z : A^-} \quad (\Rightarrow \ e)
\]
Then by applying the i.h. to the $\mathbf{DLAL}^*$ subderivation of $!\Gamma_2, \Delta_2 \vdash t_2 : C$, which is the case of claim (2), and to the $\mathbf{DLAL}$ 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 $(\forall e)$:

$$!\Gamma, \Delta \vdash t : \forall \alpha.A'$$

$$!\Gamma, \Delta \vdash t : A'\lbrack H/\alpha \rbrack \quad (\forall e)$$

with $A = A'[H/\alpha]$. By assumption we have $\forall \alpha.A' \in \mathcal{L}_{DLAL}^+$, so $A' \in \mathcal{L}_{DLAL}^*$. Moreover, also by assumption we know that $H \in \mathcal{L}_{DLAL}^*$. Therefore we get: $A'[H/\alpha] \in \mathcal{L}_{DLAL}^*$, 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 $\mathbf{DLAL}$ we get:

$$\Gamma^-; \Delta^- \vdash t : \forall \alpha.A'^-$$

$$\Gamma^-; \Delta^- \vdash t : A''[H^-/\alpha] \quad (\forall e)$$

(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 $\mathbf{DLAL}$ derivation of $; y : D^- \vdash t : A^-$. Let $x : A^-, \Theta; \Pi \vdash u : B$ be a $\mathbf{DLAL}$ 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:

$$\begin{array}{c}
!\Gamma_1, \Delta_1 \vdash t_1 : !D \\
!\Gamma_1, !\Gamma_2, \Delta_2 \vdash t_2 : !A \\
\end{array}$$

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

Consider a $\mathbf{DLAL}$ 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 $\mathbf{DLAL}$ derivable:

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

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

$$\begin{array}{c}
\Gamma^-, \Theta; \Delta^- \vdash (u[t_2/x])[t_1/y] \\
\end{array}$$

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 $(\forall e)$, $(\forall i)$, (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 \( \lambdaLa \)-term \( t^{*} \) (depending on the typing derivation for \( t \)) which is typable in \( \lambdaLa^{*} \), and that any \( \beta \) reduction sequence from \( t \) can be simulated by a longer \( \lambdaLa \) reduction sequence from \( t^{*} \). 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 \( \lambdaLa \) is defined by the following grammar:

\[
M, N ::= x | \lambda x. M | MN | !M | \text{let } N \text{ be } !x \text{ in } M \mid \#M \mid \text{let } N \text{ be } \#x \text{ in } M.
\]

The depth of \( M \) is the maximal number of occurrences of subterms of the form \( !N \) and \( \#N \) 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 DLAL*, can be considered as a type system for \( \lambdaLa \). The system we present below uses two sorts of discharged types \([A]!\) and \([A]\#\) with \( A \in \mathcal{L}_{LAL} \). Thus an environment \( \Gamma \) may contain a declaration \( x : [A]! \). If \( \Gamma = x_{1} : A_{1}, \ldots, x : A_{n} \), then \([\Gamma]!\) denotes \( x_{1} : [A_{1}]!, \ldots, x_{n} : [A_{n}]! \). For a discharged type \([A]!, [\hat{A}]\) denotes the nondischarged type \( \uparrow A \). We also define \( \hat{A} = A \) and extend the notation to environments \( \Gamma \) in a natural way. We write \( \Gamma \vdash_{\lambdaLa} M : A \) if \( M \) is a term of \( \lambdaLa \) 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 \( \Gamma \vdash_{DLAL^{*}} M : A \) if it has a derivation in DLAL* (as in Definition 5); here we allow that a judgment may have a discharged declaration \( x : [A]! \) with \( A \in \mathcal{L}_{DLAL^{*}} \) in its environment.

The reduction rules of \( \lambdaLa \) are given on Figure 5.

A term \( M \) is (\( \#!, \text{com} \))-normal if neither of the reduction rules (\( \#! \)), (!), (com1) and (com2) applies to \( M \). We write \( M \overset{(\beta)}{\rightarrow} N \) when \( M \) reduces to \( N \) by one application of the (\( \beta \)) reduction rule followed by zero or several applications of the (\( \#! \)), (!), (com1) and (com2) rules.
(Id) \[ x : A \vdash x : A \]

(Weak) \[
\Gamma, x : A \vdash M : A \\
\Gamma \vdash \lambda x . M : A \to B
\]

(\to i) \[
\Gamma_1 \vdash M : A \to B \\
\Gamma_2 \vdash N : A \\
\Gamma_1, \Gamma_2 \vdash MN : B
\]

(\to e) \[
\Gamma_1 \vdash M : A \to B \\
\Gamma_2 \vdash N : A \\
\Gamma_1, \Gamma_2 \vdash MN : B
\]

(Cntr) \[
\Gamma_1 \vdash M : A \to B \\
\Gamma_1, \Gamma_2 \vdash M[x/x_1, x/x_2] : B
\]

(\to i) \[
\Gamma_1 \vdash M : A \\
\Gamma \vdash \lambda x . M : A \to B
\]

(\to e) \[
\Gamma_1 \vdash M : A \to B \\
\Gamma_1, \Gamma_2 \vdash N : A \\
\Gamma_1, \Gamma_2 \vdash let\ N be \ circ \ x \ in \ M : B
\]

(\circ i) \[
\Gamma \vdash M : A \\
\Gamma \vdash M : \forall \alpha . A
\]

(\circ e) \[
\Gamma \vdash M : \forall \alpha . A \\
\Gamma \vdash M : A[B/\alpha]
\]

\[\text{Fig. 4. LAL as a type system for } \lambda\text{la}\]

(\beta) \[
(\lambda x . M)N \rightarrow M[N/x]
\]

(\circ) \[
\text{let } \circ N \text{ be } \circ x \text{ in } M \rightarrow M[N/x]
\]

(!) \[
\text{let } ! N \text{ be } ! x \text{ in } M \rightarrow M[N/x]
\]

(com1) \[
\text{let } N \text{ be } \uparrow x \text{ in } M \text{L } \rightarrow \text{let } N \text{ be } \uparrow x \text{ in } (ML)
\]

(com2) \[
\text{let } (\text{let } N \text{ be } \uparrow x \text{ in } M) \text{ be } \uparrow y \text{ in } L \rightarrow \text{let } N \text{ be } \uparrow x \text{ in } (\text{let } M \text{ be } \uparrow y \text{ in } L)
\]

\[\text{Fig. 5. Reduction rules of } \lambda\text{la}\]

Given a \(\lambda\text{la}\)-term \(M\), its erasure \(M^-\) is defined by:

\[
x^- = x \\
(\lambda x . M)^- = \lambda x . (M^-) \\
(\text{let } N \text{ be } \uparrow x \text{ in } M)^- = M^-[N^-/x].
\]

The following holds quite naturally.

**Lemma 15** If \(\Gamma \vdash_{\text{DLAL}} M : A\) with \(M\) a \(\lambda\text{la}\)-term, then \(\widehat{\Gamma} \vdash_{\text{DLAL}} M^- : A\).

In addition:

(1) When \(M\) contains a subterm \(N_1 \text{ be } \circ x \text{ in } N_2\), \(\text{no}(x, N_2^-) \leq 1\).

(2) When \(M\) is (\circ, !, com)-normal, \(|M^-| \leq |M|\).
The following is the main result of [39]:

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

The upper bound \( O(|M|^{2d+1}) \) for the length of reduction sequences is concerned with all reduction rules. When only the \((\beta)\) reduction rule is concerned, it can be sharpened. In fact, Lemma 14 of [39] claims that given a \( \lambda \text{LA} \)-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 \((\beta)\) reductions is bounded by \( |M| \). This observation leads to the following improvement:

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

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

**Theorem 18 (Subject Reduction for \( \lambda \text{LA} \) and \( \text{DLAL}^* \))** If \( \Gamma \vdash_{\text{DLAL}^*} M_0 : A \) and \( M_0 \rightarrow M_1 \), then \( \Gamma \vdash_{\text{DLAL}^*} M_1 : A \).

### 6.2 Simulation theorem

First of all, let us rephrase the ‘only-if’ direction of Theorem 6 in terms of \( \lambda \text{LA} \)-terms rather than plain lambda terms.

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

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

**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 \( \lambda \text{LA} \)-term. Namely,

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

We prove the existence of \( t^* \), claims (1), (2) and
(3') \(|t^*|' \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:

\(|t^*| \leq 2|t^*|' + 1 \leq 6(d+1)|t|' + 1 \leq 6(d+1)(|t| - 1) + 1 \leq 6(d+1)|t|'.

(Case 1) The derivation consists of the identity axiom. Straightforward.

(Case 2) The last rule is (\(\Rightarrow \) e):

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

Define 

\((tu)^* = \text{let } z \text{ be } !z' \text{ in } t^*(u^*[z'/z])\)

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

\(|(tu)^*|' = |t^*|' + |u^*|' + 3
\leq 3(d+1)|t|' + 3d|u|' + 3
\leq 3(d+1)(|t|' + |u|' + 1) = 3(d+1)|tu|'\),
establishing claim (3'). The other claims are easy to show.

(Case 3) The last rule is (\(\S \) i):

\[
\frac{\Gamma, \Delta \vdash t : A}{\Gamma; \S \Delta \vdash t : \S A} \quad (\S \ i)
\]

If \(t = x\), we set \(t^* = x\) 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 \(\Gamma\) (\(\Delta\), resp.)
that actually occur in \(t\) as free variables. By i.h., we have \(t^*\) associated to 
the subderivation ending with the premise of the above rule. To the whole 
derivation, we associate a new \(\lambda\)-term \(M\) defined by

\[
M = \text{let } x_1 \text{ be } !x'_1 \text{ in } \cdots \text{let } x_m \text{ be } !x'_m \text{ in } \\
\text{let } y_1 \text{ be } \S y'_1 \text{ in } \cdots \text{let } y_n \text{ be } \S y'_n \text{ in } \S t^*[x'_1/x_1, y'_1/y_1].
\]

It is easy to see that \(!\Gamma^*; \S \Delta^* \vdash_{DLAL} \S A^*\) 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 \(|t|' \neq 0\) since we have

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

23
(Case 4) The last rule is ($\exists$ e):

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

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

All other cases are straightforward.

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

**Lemma 20** Let $M$ be a term of $\lambda_{\text{LA}}$.

1. If $\Gamma \vdash_{\text{DLAL}*} M : \forall \alpha_1 \cdots \forall \alpha_n.A \rightarrow B \ (n \geq 0)$, then $M$ is in one of the following forms: $x$, $M_1M_2$, let $M_1$ be $\uparrow x$ in $M_2$, $\lambda x.M_0$.
2. If $\Gamma \vdash_{\text{DLAL}*} M : \forall \alpha_1 \cdots \forall \alpha_n.\exists A \ (n \geq 0)$, then $M$ is in one of the following forms: $x$, $M_1M_2$, let $M_1$ be $\uparrow x$ in $M_2$, $\exists M_0$.
3. If $\Gamma \vdash_{\text{DLAL}*} M : !A$, then $M$ is in one of the following forms: $x$, let $M_1$ be $\uparrow x$ 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 $B \rightarrow !A \notin \mathcal{L}_{\text{DLAL}*}$.

**Lemma 21**

1. If $\Gamma \vdash_{\text{DLAL}*} MN : A$ and $MN$ is ($\exists$, $\neg$, $\downarrow$, com)-normal, then $M$ is a variable $x$, an application $M_1M_2$ or an abstraction $\lambda x.M_0$.
2. If $\Gamma \vdash_{\text{DLAL}*}$ let $M$ be $\exists x$ in $N : A$ and (let $M$ be $\exists x$ in $N$) is ($\exists$, $\neg$, $\downarrow$, com)-normal, then $M$ is either a variable $x$ or an application $M_1M_2$.
3. If $\Gamma \vdash_{\text{DLAL}*}$ let $M$ be $!x$ in $N : A$ and (let $M$ be $!x$ in $N$) is ($\exists$, $\neg$, $\downarrow$, com)-normal, then $M$ is a variable $x$.

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

\[
\frac{\Gamma_1 \vdash M : A \rightarrow B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash MN : B} \quad (\rightarrow e)
\]

then $M$ cannot be of the form let $M_1$ be $\uparrow x$ 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.

24
By induction on the structure of the derivation. If the last rule is
\[
\Gamma_1 \vdash M : \xi A \quad \Gamma_2, x : [A]_\xi \vdash N : B
\]
\[
\frac{}{\Gamma_1, \Gamma_2 \vdash \text{let } M \text{ be } \xi x \text{ in } N : B} (\xi e)
\]
then M cannot be of the form \(\text{let } M_1 \text{ be } \xi y \text{ in } M_2\) since \(\text{let } M \text{ be } \xi x \text{ in } N\) is (com)-normal. Likewise, \(M\) cannot be of the form \(\xi M_0\). Hence by Lemma 20 (2), \(M\) must be either a variable or an application.

Similarly to (2).

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

\[
\begin{array}{c}
M \xrightarrow{\beta}(\xi) N \\
\downarrow \quad \downarrow \\
(\xi) \\
M \xrightarrow{\beta} N
\end{array}
\]

Proof. It is clearly sufficient to find a \(\lambda\text{LA}\)-term \(\xi N'\) such that \(M \xrightarrow{\beta}(\xi) N\) and \((\xi N')^- = u\). A suitable \((\xi, !, \text{com})\)-normal term \(N\) can then be obtained by reduction rules \((\xi), (!)\) and \((\text{com})\). 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

\[
(\lambda x. t_0) t_2 \xrightarrow{\beta} t_0[t_2/x] \\
(\lambda x. M_0) M_2 \xrightarrow{\beta} M_0[M_2/x]
\]
as required.

(Case 4) \(M\) is of the form \(\lambda y.M_0\). By the induction hypothesis.

(Case 5) \(M\) is of the form \text{let } M_1 \text{ be } \xi x \text{ 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 \(x \in \text{FV}(t_2)\)) or
results from a redex in \( t_2 \) by substituting \( t_1 \) for \( x \).

In the former case, let \( t_1 \xrightarrow{} u_1 \). Then by the induction hypothesis, there is some \( N_1 \) such that

\[
\begin{align*}
t_1 & \xrightarrow{(\beta)} u_1 \\
M_1 & \xrightarrow{(\beta)} N_1
\end{align*}
\]

Therefore, we have

\[
\begin{align*}
t_2[t_1/x] & \xrightarrow{(\beta)} t_2[u_1/x] \\
& \\
\text{let } M_1 & \text{ be } \beta x \text{ in } M_2 \xrightarrow{(\beta)} N_1 \text{ be } \beta x \text{ in } M_2
\end{align*}
\]

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

In the latter case, let \( t_2 \xrightarrow{} u_2 \). By the induction hypothesis, there is \( N_2 \) such that

\[
\begin{align*}
t_2 & \xrightarrow{(\beta)} u_2 \\
M_2 & \xrightarrow{(\beta)} N_2
\end{align*}
\]

We therefore have

\[
\begin{align*}
t_2[t_1/x] & \xrightarrow{(\beta)} u_2[t_1/x] \\
& \\
\text{let } M_1 & \text{ be } \beta x \text{ in } M_2 \xrightarrow{(\beta)} \text{ let } M_1 \text{ be } \beta x \text{ in } N_2
\end{align*}
\]

as required.

(Case 6) \( M \) is of the form let \( M_1 \) be \(!x\) 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 \( \lambda \)La-terms. Indeed sometimes one \((\beta)\) step in a LAL typed lambda-term might not be simulated by a sequence of \((\beta)\) steps in the
corresponding \(\lambda\)-term. The reason for that is that \texttt{let} \(\lambda x\) in \(\cdot\) construct allows for sharing in \(\lambda\)-terms. This is directly related to the fact that \texttt{LAL} lambda-terms do not admit subject-reduction for \((\beta)\) 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 \(\Gamma; \Delta \vdash \texttt{DLAL} t : A\) and \(t \rightarrow u\). By Lemma 19, there is a \(\lambda\)-term \(t^*\) such that \(!\Gamma^*, \Delta^* \vdash \texttt{DLAL}^* t^* : A^*\). Moreover, the simulation theorem implies that there is a \(\lambda\)-term \(N\) such that \(t^* \xrightarrow{(\beta)} N\) and \(N^- = u\). Finally, Theorem 18, Lemma 15 and Theorem 6 together imply that \(\Gamma; \Delta \vdash \texttt{DLAL} u : A\).

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

By the simulation theorem, we have:

\[
\begin{array}{c}
t \\ \xrightarrow{(\beta)} t^* \\
\end{array} \xrightarrow{(\beta)} u
\]

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

To show that the normal form can be computed in time \(O(|t|^{2d+2})\) by a Turing machine, notice that any term \(u\) such that \(t \rightarrow^* u\) 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|^{2d} \cdot (|t|^{2^d})^2 \leq |t|^{2d+2}\.

7 Translation of LAL into DLAL

Although \texttt{DLAL} is a proper subsystem of \texttt{LAL}, it is as expressive as \texttt{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 \(( \cdot )^*\) from \(L_{LAL}\) to \(L_{DLAL}\) as follows:

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

Define also a mapping from \(\lambda\text{LAL}-\text{terms}\) to lambda-terms by:

\[x^* = x\]
\[(\lambda x.M)^* = \lambda x.M^*\]
\[(MN)^* = M^*N^*\]
\[(!M)^* = \lambda x.xM^*, \text{ where } x \text{ is fresh.}\]
\[(\text{let } N \text{ be } !y \text{ in } M)^* = N^*(\lambda y.M^*)\]
\[(\$M)^* = M^*\]
\[(\text{let } N \text{ be } \$y \text{ in } M)^* = M^*[N^*/y]\]

This mapping preserves typing:

**Proposition 24** Let \([\Gamma]\) be an environment \(x_1 : [A_1], \ldots, x_m : [A_m]\) and \(\Delta\) be \(y_1 : [B_1]\), \ldots, \(y_n : [B_n]\), \(z_1 : C_1\), \ldots, \(z_k : C_k\). Then,

\([\Gamma]!, \Delta \vdash_{LAL} M : A \quad \text{implies} \quad \Gamma^*; \Delta^* \vdash_{DLAL} M^* : A^*\),

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

**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

\[\frac{\vdash x : A^* \Rightarrow \alpha \vdash x : A^* \Rightarrow \alpha}{y : B^*; x : A^* \vdash \alpha \vdash xM^* : \alpha}\]
\[\frac{y : B^*; x : A^* \Rightarrow \alpha \vdash xM^* : \alpha}{y : B^*; \vdash \lambda x.M^* : (A^* \Rightarrow \alpha) \Rightarrow \alpha}\]
\[\frac{y : B^*; \vdash \lambda x.M^* : \forall \alpha. (A^* \Rightarrow \alpha) \Rightarrow \alpha}{y : B^*; \vdash \lambda x.M^* : \forall \alpha. (A^* \Rightarrow \alpha) \Rightarrow \alpha}\]
(Case 2) The last rule is (! e):

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

We have

\[ \frac{\Gamma_1^*; \Delta_1^* \vdash N^* : \forall \alpha.(A^* \Rightarrow \alpha) \sigma \alpha \quad y : A^*, \Gamma_2^*; \Delta_2^* \vdash M^* : B^*}{\Gamma_1^*; \Delta_1^* \vdash N^* : (A^* \Rightarrow B^*) \sigma B^* \quad \Gamma_2^*; \Delta_2^* \vdash \lambda y.M^* : A^* \Rightarrow B^*}{\Gamma_1^*; \Gamma_2^*; \Delta_1^*, \Delta_2^* \vdash N^*(\lambda y.M^*) : B^*} \]

The other cases are straightforward.

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

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

**Proof.** By induction on the structure of \( M \).

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

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

\[ M^* \xrightarrow{} N^*. \]

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

\[ \text{(let } !N \text{ be } !y \text{ in } M)^* = (\lambda x.xN^*)(\lambda y.M^*) \xrightarrow{} (\lambda y.M^*)N^* \xrightarrow{} M^*[N^*/y] = (M[N/y])^*. \]

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 \( \text{DLAL} \) expresses as many algorithms as \( \text{LAL} \) based on the translation \((\cdot)^*\). But there are two obstacles.
To overcome the first difficulty, we consider, not a λ we define two lambda terms

\[ \text{en}_A : A \rightarrow A^\star, \quad \text{de}_A : A^\star \rightarrow A. \]

\[
\begin{align*}
\text{en}_a(t) &= t & \text{de}_a(t) &= t \\
\text{en}_{\alpha A}(t) &= \text{en}_A(t) & \text{de}_{\alpha A}(t) &= \text{de}_A(t) \\
\text{en}_{\beta A}(t) &= \text{en}_A(t) & \text{de}_{\beta A}(t) &= \text{de}_A(t) \\
\text{en}_{A \rightarrow B}(t) &= \lambda x. \text{en}_B(t \text{ de}_A(x)) & \text{de}_{A \rightarrow B}(t) &= \lambda x. \text{de}_B(t \text{ en}_A(x)) \\
\text{en}_{A \equiv B}(t) &= \lambda x. \text{en}_B(x(\lambda y. t \text{ de}_A(y))) & \text{de}_{A \equiv B}(t) &= \lambda x. \text{de}_B(t(\lambda y. y \text{ en}_A(x)))
\end{align*}
\]

We have:

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

\[ z : A \vdash \text{en}_A(z) : A^\star, \quad z : A^\star \vdash \text{de}_A(z) : A. \]

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

\[ (A \Rightarrow B)^\star = (\forall \alpha. (A^\star \Rightarrow \alpha) \Rightarrow \alpha) \Rightarrow B^\star. \]

As to \( \text{en}_{A \Rightarrow B} \), we have:

\[
\begin{array}{c}
\vdash x : (A^\star) \Rightarrow x : (A^\star) \\
\vdash y : A^\star; z : A \Rightarrow B \vdash z \text{ de}_A(y) : B \\
\vdash z : A \Rightarrow B, x : (\lambda y. z \text{ de}_A(y)) : B
\end{array}
\]

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

As to \( \text{de}_{A \Rightarrow B} \), we have:

\[
\begin{array}{c}
\vdash y : A^\star \Rightarrow \alpha \vdash y : A^\star \Rightarrow \alpha \\
\vdash x : A \vdash \text{en}_A(x) : A^\star \\
\vdash x : A; y : A^\star \Rightarrow \alpha \vdash \text{en}_A(x) : \alpha \\
\vdash x : A; \lambda y. \text{en}_A(x) : (A^\star \Rightarrow \alpha) \Rightarrow \alpha \\
\vdash x : A; \lambda y. \text{en}_A(x) : (A^\star) \Rightarrow z(\lambda y. y \text{ en}_A(x)) : B^\star
\end{array}
\]

30
By substituting the above conclusion into \( w : B \leadsto \text{de}_B(w) : B \), one obtains the desired judgement.

We then show that \( M^\star \) and \( M^- \) are related via \( \text{de}_A \), when \( A \) is a \( \Pi_1 \) type in \( L_{DLAL} \) and \( M \) is a \( \lambda \Lambda A \) 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 \( L_{DLAL} \). 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 \( \varphi \) maps each type variable \( \alpha \) to a \( \beta\eta \)-closed relation.

Given a valuation \( \varphi \), one can inductively define a binary relation \( [A]_\varphi \) for each \( \Lambda \)L type \( A \):

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

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 \( \Lambda \Lambda \)L types \( A \) and \( B \), \( [A[B/\alpha]]_\varphi \) is equal to \( [A]_\varphi[\alpha \rightarrow [B]_\varphi] \).

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 \( \Gamma \) be an environment \( x_1 : A_1, \ldots, x_n : A_n, x_{n+1} : [A_{n+1}], \ldots, x_m : [A_m] \). If \( \Gamma \vdash_{DLAL} M : B \), then for any valuation \( \varphi \) and any \( (u_i, v_i) \in [A_i]_\varphi \) (1 \( \leq i \leq m \)), we have \( (M^\star[u_i/\bar{x}], M^-[v_i/\bar{x}]) \in [B]_\varphi \).

**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 \( (\rightarrow \ i) \):

\[
\frac{y : A, \Gamma \vdash M : B}{\Gamma \vdash \lambda y.M : A \rightarrow B}
\]

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

(Case 3) The last rule is (\(\to\ e\)):

\[
\frac{\Gamma_1 \vdash M : A \to B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M N : B}
\]

For simplicity, we assume \(\Gamma_1\) and \(\Gamma_2\) are empty (the argument is then easily adapted to the general case). By i.h., \((M^\bullet, M^-) \in [A \to B]_\varphi\) and \((N^\bullet, N^-) \in [A]_\varphi\). Hence \(((MN)^\bullet, (MN)^-) \in [B]_\varphi\).

(Case 4) The last rule is (\(! i\)):

\[
\frac{y : B \vdash M : A}{y : \lfloor B \rfloor \vdash ! M : !A}
\]

Let \((u, v) \in \lfloor B \rfloor_\varphi\). 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 \(((!M)[u/y], (!M)^-[v/y]) \in \lfloor A \rfloor_\varphi\). By Lemma 28, \((M^\bullet, M^-) \in [A[u/y]]_\varphi\).

(Case 5) The last rule is (\(! e\)):

\[
\frac{\Gamma_1 \vdash N : !A \quad y : \lfloor A \rfloor, \Gamma_2 \vdash M : C}{\Gamma_1, \Gamma_2 \vdash \text{let } \lambda y. M \text{ be } ! y \text{ in } M : C}
\]

For simplicity, we assume \(\Gamma_1\) and \(\Gamma_2\) are empty. By i.h., \((N^\bullet, N^-) \in \lfloor !A \rfloor_\varphi\), so there is some \(u\) such that \(N^\bullet =_{\beta\eta} \lambda z.zu\) (with \(z\) fresh) and \((u, N^-) \in \lfloor A \rfloor_\varphi\). Hence by i.h. we have \((M^\bullet[u/y], M^-[N^-/y]) \in [C]_\varphi\). Since \(\lambda z.z[\lambda y.M^\bullet][u/y]\)

and \((\text{let } \lambda y. M \text{ be } ! y \text{ in } M)^- = M^-[N^-/y]\), we are done.

(Case 6) The last rule is (\(\forall i\)):

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

Let \(\varphi\) be a valuation and let \((u_i, v_i) \in [A_i]_\varphi\) for every \(1 \leq i \leq m\). Since \(\alpha\) does not appear free in any \(A_i\), we have \((u_i, v_i) \in [A_i]_{\varphi[\alpha \to R]}\) for any \(\beta\eta\)-closed relation \(R\). Therefore, we have \((M^\bullet[\vec{u}/\vec{x}], M^-[\vec{v}/\vec{x}]) \in [\forall \alpha.A]_\varphi\) by i.h. Hence \((\lambda z.z[\lambda y.M^\bullet][\vec{u}/\vec{x}], \lambda z.z[\lambda y.M^\bullet][\vec{v}/\vec{x}]) \in [\forall \alpha.A]_\varphi\).

(Case 7) The last rule is (\(\forall e\)):

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

For simplicity, we assume \(\Gamma\) is empty. By i.h., we have \((M^\bullet, M^-) \in [\forall \alpha.A]_\varphi\), and hence \((M^\bullet, M^-) \in [A]_{\varphi[\alpha \to R]}\) for any \(\beta\eta\)-closed relation \(R\). In particular, \((M^\bullet, M^-) \in [A]_{\varphi[\alpha \to [B]_\varphi]}\), 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 $\varphi$ by taking $\varphi(\alpha)$ to be $=_{\beta\eta}$ for every variable $\alpha$. 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 $\Pi_1$ and $(t, u) \in [D^*]$, then $\text{de}_D(t) =_{\beta\eta} u$.

2. If $D$ is $\Sigma_1$, then for any lambda-term $t$ of the form $xt_1 \cdots t_n$ ($n \geq 0$), we 

have $(\text{en}_D(t), t) \in [D^*]$.

**Proof.** By induction on the structure of $D$.

(Case 1) $D$ is a variable $\alpha$. Straightforward because $[\alpha]$ is just $=_{\beta\eta}$ and $\text{de}_\alpha$, $\text{en}_\alpha$ 

are just identity.

(Case 2) $D$ is of the form $A \rightarrow B$.

1. Suppose that $A \rightarrow B$ is $\Pi_1$ and $(t, u) \in [A^* \rightarrow B^*]$. Let $x$ be a fresh variable. Since $(\text{en}_A(x), x) \in [A^*]$ by i.h. (with $A$ a $\Sigma_1$ type), we have 

$(t \ \text{en}_A(x), ux) \in [B^*]$. Hence by i.h. (with $B$ a $\Pi_1$ type), $\text{de}_B(t \ \text{en}_A(x)) =_{\beta\eta} ux$. Therefore, 

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

2. Suppose that $A \rightarrow B$ is $\Sigma_1$ and let $t = xt_1 \cdots t_n$. To prove $(\text{en}_{A \rightarrow B}(t), t) \in 

[\left[A^* \rightarrow B^*\right]]$, it is sufficient to show that $(\text{en}_{A \rightarrow B}(t), uv) \in [B^*]$ holds 

for any $(u, v) \in [A^*]$. By i.h., $\text{de}_A(u) =_{\beta\eta} v$. Hence 

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

Therefore by i.h., we have $(\text{en}_{A \rightarrow B}(t), tv) \in [B^*]$.

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

1. Suppose that $A \Rightarrow B$ is $\Pi_1$ and $(t, u) \in [\neg A^* \rightarrow B^*]$. Let $x$ be a fresh variable. Since $(\text{en}_A(x), x) \in [\neg A^*]$ by i.h. we have $(\lambda y. y \ \text{en}_A(x), x) \in 

[\neg A^*]$. Hence $(t(\lambda y. y \ \text{en}_A(x)), ux) \in [B^*]$, and so $\text{de}_B(t(\lambda y. y \ \text{en}_A(x))) =_{\beta\eta} ux$. We therefore have 

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

2. Suppose that $A \Rightarrow B$ is $\Sigma_1$. Let $t = xt_1 \cdots t_n$ and $(u, v) \in [A^*]$. Our 

purpose is to show $(\text{en}_{A \Rightarrow B}(t), tv) \in [B^*]$.

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

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

$$=_{\beta\eta} \text{en}_B((\lambda x. u')(\lambda y. t \ \text{de}_A(y))) =_{\beta\eta} \text{en}_B(t \ \text{de}_A(u')) =_{\beta\eta} \text{en}_B(tv).$$
Moreover by i.h. on $B$ we have $(\text{en}_B(tv), tv) \in [B^*]$, so as $\text{en}_{A \Rightarrow B}(t)u = \beta\eta \text{en}_B(tv)$ we get that: $(\text{en}_{A \Rightarrow B}(t), tv) \in [B^*]$. 

(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 $§A$. Straightforward.

Combined with the basic lemma, it results in:

**Corollary 31** If $A$ is a $\Pi_1$ type of $\text{DLAL}$ and $\vdash_{\text{LAL}} M : A^*$, then $\text{de}_A(M^*) = \beta\eta M^*$.

### 7.3 Ptime completeness

Let us now show that $\text{DLAL}$ has the same expressive power as $\text{LAL}$, 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 $\lambda$A-term:

\[
\begin{align*}
\overline{w} &= \lambda f_0.\lambda f_1.\lambda x.f_i(f_{i_1} \cdots (f_{i_n}x) \cdots) : W \\
\overline{w}_L &= \lambda f_0.\text{let } f_0 \text{ be } !g_0 \text{ in } (\text{let } f_1 \text{ be } !g_1 \text{ in } \not(\lambda x.g_i(g_{i_1} \cdots (g_{i_n}x) \cdots))) : W_L
\end{align*}
\]

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

\[
\text{dtol}_W = \lambda x.\overline{w}.\lambda f_0.\lambda g_0.\lambda f_1.\lambda g_1.\lambda f_2.\lambda g_2.\lambda f_3.\lambda g_3.(\lambda x.g_i(g_{i_1} \cdots (g_{i_n}x) \cdots)).
\]

(Here we cannot use $\text{en}_W$, as it returns a slightly different term when applied to $\overline{w}$.) To see that $\text{dtol}_W$ is surely of type $W \rightarrow W_L^*$, note that

\[
W_L^* = \forall \alpha.(\neg(\alpha \Rightarrow \alpha))^* \rightarrow (\neg(\alpha \Rightarrow \alpha))^* \rightarrow \neg(\alpha \Rightarrow \alpha),
\]

where $(\neg(\alpha \Rightarrow \alpha))^* = (\forall \beta.((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta)$. Hence the variables $f_0, f_1$ must be typed $(\forall \beta.((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta)$. By instantiating $\beta$ into $\neg(\alpha \Rightarrow \alpha)$ for the type of $f_1$ and into $(\neg(\alpha \Rightarrow \alpha))^* \rightarrow \neg(\alpha \Rightarrow \alpha)$ for the type of $f_0$ (and assuming that $g_0$ and $g_1$ are of type $\alpha \Rightarrow \alpha$), one obtains:
\[
x_{g_{0}g_{1}} : \xi(\alpha \rightarrow \alpha)
\]
\[
\lambda_{g_{1}.x_{g_{0}g_{1}}} : (\alpha \rightarrow \alpha) \Rightarrow \xi(\alpha \rightarrow \alpha)
\]
\[
f_{1}(\lambda_{g_{1}.x_{g_{0}g_{1}}}) : \xi(\alpha \rightarrow \alpha)
\]
\[
\lambda_{f_{1}.f_{1}(\lambda_{g_{1}.x_{g_{0}g_{1}}})(!(\alpha \rightarrow \alpha))} \cdot \Rightarrow \xi(\alpha \rightarrow \alpha)
\]
\[
f_{0}(\lambda_{g_{0}.f_{1}.f_{1}(\lambda_{g_{1}.x_{g_{0}g_{1}}}))(!(\alpha \rightarrow \alpha))} \cdot \Rightarrow \xi(\alpha \rightarrow \alpha)
\]
\[
f_{0}(\lambda_{g_{0}.f_{1}.f_{1}(\lambda_{g_{1}.x_{g_{0}g_{1}}}))(!(\alpha \rightarrow \alpha))} \cdot \Rightarrow \xi(\alpha \rightarrow \alpha)
\]
\[
f_{0}(\lambda_{g_{0}.f_{1}.f_{1}(\lambda_{g_{1}.x_{g_{0}g_{1}}}))(!(\alpha \rightarrow \alpha))} \cdot \Rightarrow \xi(\alpha \rightarrow \alpha)
\]

It is easy to see that \(\text{dtol}_{W}(w)\) reduces to \((w_{L})^{\ast}\), by noting:

\[
(w_{L})^{\ast} = \lambda_{f_{0}.f_{0}(\lambda_{g_{0}.f_{1}.f_{1}(\lambda_{g_{1}.x_{g_{1}}}(g_{i_{2}} \cdots (g_{i_{n}}x) \cdots)))}.
\]

**Theorem 32 (Preservation of representable functions)** Suppose that a function \(f : \{0, 1\}^{\ast} \rightarrow \{0, 1\}^{\ast}\) is represented by a \(\lambda\text{LA}\)-term \(M\) with \(\vdash_{\text{LAL}} M : W_{L} \rightarrow \xi^{n}W_{L}\) for some \(n \geq 0\). Define a lambda-term \(t\) by

\[
t = \lambda x.\text{d}_{\xi^{n}}W(M^{\ast}\text{dtol}_{W}(x)).
\]

Then we have that \(\vdash_{\text{DLAL}} t : W \rightarrow \xi^{n}W\) and \(t\) also represents \(f\).

**Proof.** The term \(t\) gets type \(W \rightarrow \xi^{n}W\) in \(\text{DLAL}\) as follows (notice that \(W^{\ast} = W_{L}\), using Proposition 24 and Lemma 27:

\[
\vdash M^{\ast} : W^{\ast} \rightarrow \xi^{n}W^{\ast} ; x : W \vdash \text{dtol}_{W}(x) : W^{\ast}
\]

\[
; x : W \vdash M^{\ast}\text{dtol}_{W}(x) : \xi^{n}W^{\ast}
\]

\[
; x : W \vdash \text{d}_{\xi^{n}}W(M^{\ast}\text{dtol}_{W}(x)) : \xi^{n}W
\]

\[
\vdash t : W \rightarrow \xi^{n}W
\]

Since \(M\) represents \(f\), \(Mw_{L}\) reduces to \((f(w))_{L}^{\ast}\) for any \(w \in \{0, 1\}^{\ast}\). This implies that \((Mw_{L})^{-} = \beta_{n} (f(w))_{L}^{\ast}\). On the other hand,

\[
tw = \beta_{n} \text{d}_{\xi^{n}}W(M^{\ast}\text{dtol}_{W}(w)) = \beta_{n} \text{d}_{\xi^{n}}W(M^{\ast}(w_{L}))^{\ast} = \text{d}_{\xi^{n}}W((Mw_{L})^{\ast}).
\]

Applying Corollary 31, as \(\xi^{n}W\) is \(\Pi_{1}\), we thus get:

\[
tw = \beta_{n} (Mw_{L})^{-} = \beta_{n} (f(w))_{L}^{-} = f(w).
\]

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 \(\lambda\text{LA}\))** If a function \(f : \{0, 1\}^{\ast} \rightarrow \{0, 1\}^{\ast}\) is computable in time \(O(n^{2d})\) by a multi-tape Turing machine for some \(d\), then there exists a \(\lambda\text{LA}\)-term \(M\) such that \(\vdash_{\text{LAL}} M : W_{L} \rightarrow \xi^{2d+2}W_{L}\) and \(M\) represents \(f\).
Proof (Sketch). Following the idea of [2], let $\text{Conf}$ be the LAL-type
\[
\forall \alpha. !(\alpha \to \alpha)^m \to \$((\alpha \to \alpha)^{2n} \otimes B^k),
\]
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 $\lambda$LaL-terms:

- $\text{trans} : \text{Conf} \to \text{Conf}$ for one-step transition of a Turing Machine;
- $\text{init} : W_L \to \text{Conf}$ for initialization;
- $\text{output} : \text{Conf} \to W_L$ for output extraction;
- $\text{length} : W_L \to N_L$ for the length map;
- $\text{dupl} : W_L \to \$((W_L \otimes W_L)$ for duplication.

Furthermore, Proposition 11 yields a term $\text{prod}(2^d) : N_L \to \$^{2d}N_L$ for the
map $n \mapsto n^{2^d}$. By applying $\text{iter}_\text{Conf}$ to $\text{trans}$ and $\text{init}$, we obtain
\[
x : N_L, y : \$W_L \vdash \text{LAL} M_0 : \$\text{Conf}.
\]
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 $\text{prod}(2^d)$, we obtain
\[
x : N_L, y : \$^{2d+1}W_L \vdash \text{LAL} M_1 : \$^{2d+1}\text{Conf},
\]
which iterates the transition $x^{2^d}$ times. Finally, by using $\text{output}$, $\text{length}$, a
variant of $\text{coer}_1$ and $\text{dupl}$, we obtain the desired term $M : W_L \to \$^{2d+2}W_L$
representing the function $f$.

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

8 Conclusion and perspectives

We have presented a polymorphic type system for lambda calculus which guar-
antees 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


