Combining Linear Logic and Size Types for Implicit Complexity

Patrick Baillot, Alexis Ghyselen

Combining Linear Logic and Size Types for Implicit Complexity (Long Version)

Patrick Baillot[1] Alexis Ghyselen[2]

[1] Univ Lyon, CNRS, ENS de Lyon, Université Claude-Bernard Lyon 1, LIP, F-69342, Lyon Cedex 07, France
[2] ENS Paris-Saclay, France

Abstract. Several type systems have been proposed to statically control the time complexity of lambda-calculus programs and characterize complexity classes such as FPTIME or FEXPTIME. A first line of research stems from linear logic and defines type systems based on restricted versions of the "!" modality controlling duplication. An instance of this is light linear logic for polynomial time computation [Girard98]. A second perspective relies on the idea of tracking the size increase between input and output, and together with a restricted use of recursion, to deduce from that time complexity bounds. This second approach is illustrated for instance by non-size-increasing types [Hofmann99]. However both approaches suffer from limitations. The first one, that of linear logic, has a limited intensional expressivity, that is to say some natural polynomial time programs are not typable. As to the second approach it is essentially linear, more precisely it does not allow for a non-linear use of functional arguments. In the present work we adress the problem of incorporating both approaches into a common type system. The source language we consider is a lambda-calculus with data-types and iteration, that is to say a variant of Godel's system T. Our goal is to design a system for this language allowing both to handle non-linear functional arguments and to keep a good intensional expressivity. We illustrate our methodology by choosing the system of elementary linear logic (ELL) and combining it with a system of linear size types. We discuss the expressivity of this new type system and prove that it gives a characterization of the complexity classes FPTIME and 2k-FEXPTIME, for $k \geqslant 0$.

1 Introduction

Controlling the time complexity of programs is a crucial aspect of program development. Complexity analysis can be performed on the overwhole final program and some automatic techniques have been devised for this purpose. However if the program does not meet our expected complexity bound it might not be easy to track which subprograms are responsible for the poor performance and how they should be rewritten in order to improve the global time bound. Can one instead investigate some methodologies to program while staying in a given complexity class? Can one carry such program construction without having to deal with explicit annotations for time bounds? These are some of the questions that have been explored by *implicit computational complexity*, a line of research which defines calculi and logical systems corresponding to various complexity classes, such as FP, FEXPTIME, FLOGSPACE ...

State of the Art. A first success in implicit complexity was the recursion-theoretic characterization of FP [9]. This work on safe recursion leads to languages for polynomial time [16], for oracle functionals or for probabilistic computation [13, 21]. Among the other different approaches of implicit complexity one can mention two important threads of work. The first one is issued from linear logic, which provides a decomposition of intuitionistic logic with a modality, !, accounting for duplication. By designing variants of linear logic with weak versions of the ! modality one obtains systems corresponding to different complexity classes, like light linear logic (LLL) for the class FP [15] and elementary linear logic (ELL) for the classes k-FEXPTIME, for $k \geqslant 0$. [15, 3, 14]. These logical systems can be seen as type systems for some variants of lambda-calculi. A key feature of these systems, and the main ingredient for proving their complexity properties, is that they induce a stratification of the typed program into levels. We will thus refer to them as *level-based systems*. Their advantage is that they deal with a higher-order language, and that they are also compatible with polymorphism. Unfortunately they have a critical drawback: only few and very specific programs are actually typable, because the restrictions imposed to recursion by typing are in fact very strong... A second thread of work relies on the idea of tracking the size increase between the input and the output of a program. This approach is well illustrated by Hofmann's Non-size-increasing (NSI) type system [17] : here the types carry information about the input/output size difference, and the recursion is restricted in such a way that typed programs admit polynomial time complexity. An important advantage with respect to LLL is that the system is algorithmically more expressive, that is to say that far more programs are typable. Of course such a simple system cannot be expected to recognize programs which are polynomial time for subtle reasons, but it enlightens interesting situations where the complexity can be deduced from this simple size analysis. A similar idea is also explored by the line of work on quasi-interpretations [10, 5], with a slightly different angle : here the kind of dependence between input and output size can be more general but the analysis is more of a semantic nature and in particular no type system is provided to derive quasi-interpretations. The type system $\mathsf{d}\ell\mathsf{T}$ of [4] can be thought of as playing this role of describing the dependence between input and output size, and it allows to derive sharp time complexity bounds, even though these are not limited to polynomial bounds. Altogether we will refer to these approaches as *size-based systems*. Unfortunately they also have a limitation: they do not deal with a full-fledge higher-order language, in the sense that functional arguments have to be used linearly, that is to say at most once.

Problematic. So on the one hand level-based systems manage higher-order but have a poor expressivity, and on the other hand sized-based systems have a good expressivity but do not deal with general higher-order... This is not a mere historical incident, in the sense that on both sides some attempts have been made to repair these defects but only with modest success: in [7] for instance LLL is extended to a language with recursive definitions, but the main expressivity problem remains; in [5] quasi-interpretations are defined for a higher-order language, but a linearity condition still has to be imposed on functional arguments. The goal of the present work is precisely to try to remedy to this problem by reconciliating the level-based and the size-based approaches. From a practical point of view we want to design a system which would bring together the advantages of the two approaches. From a fundamental point of view we want to understand how the levels and the input/output size dependencies are correlated, and for instance if one of these two characteristics subsumes the other one.

Methodology. One way to bridge these two approaches could be to start with a level-based system such as LLL, and try to extend it with more typing rules so as to integrate in it some size-based features. However a technical difficulty for that is that the complexity bounds for LLL and variants of this system are usually obtained by following specific term reduction strategies such as the *level-by-level* strategy. Enriching the system while keeping the validity of such reduction strategies turns out to be very intricate. For instance this has been done in [7] for dealing with recursive definitions with pattern-matching, but at the price of technical and cumbersome reasonings on the reduction sequences. Our methodology to overcome this difficulty in the present work will be to choose a variant of linear logic for which we can prove the complexity bound by using a measure which decreases for *any* reduction step. So in this case there is no need for specific reduction strategy, and the system is more robust to extensions. For that purpose we use elementary linear logic (ELL), and more precisely the elementary lambda-calculus studied in [20].

Our language. Let us recall that ELL is essentially obtained from linear logic by dropping the two axioms $!A \multimap A$ and $!A \multimap !!A$ for the ! functor (the co-unit and co-multiplication of the comonad). Basically, if we consider the family of types $W \multimap !^i W$ (where W is a type for binary words), the larger the integer i, the more computational power we get... This results in a system that can characterize the classes k-FEXPTIME, for $k \geqslant 0$ [3]. The paper [20] gives a reformulation of the principles of ELL in an extended lambda-calculus with constructions for !. It also incorporates other features (references and multithreading) which we will not be interested in here. Our idea will be to enrich the elementary lambda-calculus by a kind of *bootstrapping*, consisting in adding more terms to the "basic" type $W \multimap W$. For instance we can think of giving to this type enough terms for representing all polynomial time functions. The way we implement this idea is by using a second language. We believe that several equivalent choices could be made for this second language, and here we adopt for simplicity a variant of the language $\mathsf{d}\ell\mathsf{T}$ from [4], a descendant of previous work on linear dependent types [19]. This language is a linear version of system T, that is to say a lambda-calculus with recursion, with types annotated with size expressions. Actually the type system of our second language can be thought of as a linear cousin of sized types [18, 2] and we call it $\mathsf{s}\ell\mathsf{T}$. So on the whole our global language can be viewed as a kind of two-layer system, the lower one used for tuning first-order intensional expressivity, and the upper one for dealing with higher-order computation and non-linear use of functional arguments.

Roadmap of the paper. We will first define $\mathsf{s}\ell\mathsf{T}$ and investigate its properties (Sect. 2). Then we will recall the elementary lambda-calculus, define our enriched calculus, describe some examples of programs and study the reduction properties of this calculus (Sect. 3). After that we will establish the complexity results of our enriched calculus (Sect. 4).

2 Presentation of $\mathsf{s}\ell\mathsf{T}$ and Control of the Reduction Procedure

First, we present the calculus that we are using for polynomial time computation, relying on linear types with sizes. We adapt a linear version of Gödel's system T, called $\mathsf{d}\ell\mathsf{T}$ defined in [4]. The calculus that we define in this paper, called $\mathsf{s}\ell\mathsf{T}$ for sized linear system T, is $\mathsf{d}\ell\mathsf{T}$ without references and with some simplifications and modifications in indexes and types. Informally, $\mathsf{s}\ell\mathsf{T}$ is a linear λ-calculus enriched with constructors for base types such as booleans, integers and words, and it comes with a constructor for high-order primitive recursion. Types are enriched with a polynomial index describing the size of the value represented by a term, and this index imposes a restriction on recursions. With this, we are able to derive a weight on terms in order to control the number of reduction steps of a term.

2.1 Syntax of $\mathsf{s}\ell\mathsf{T}$ and Type System

Substitution. For an object t with a notion of free variable and substitution we write $t[t'/x]$ the term t in which free occurrences of x have been replaced by t'.

Terms. Terms and values of $\mathsf{s}\ell\mathsf{T}$ are defined by the following grammars :
$t := x \mid \lambda x.t \mid t\ t' \mid t \otimes t' \mid let\ x \otimes y = t\ in\ t' \mid \mathsf{zero} \mid succ(t) \mid ifn(t,t') \mid itern(V,t)$
$\mid \epsilon \mid s_0(t) \mid s_1(t) \mid ifw(t_0,t_1,t') \mid iterw(V_0,V_1,t) \mid \mathsf{tt} \mid \mathsf{ff} \mid if(t,t')$
$V := x \mid \lambda x.t \mid V \otimes V' \mid \mathsf{zero} \mid succ(V) \mid ifn(V,V') \mid itern(V,V')$
$\mid \epsilon \mid s_0(V) \mid s_1(V) \mid ifw(V_0,V_1,V') \mid iterw(V_0,V_1,V') \mid \mathsf{tt} \mid \mathsf{ff} \mid if(V,V')$

We define free variables and free occurrences as usual and we work up to α-renaming. In the following, we will often use the notation s_i to regroup the cases s_0 and s_1. Here, we choose the alphabet $\{0,1\}$ for simplification, but we could have taken any finite alphabet Σ and in this case, the constructors ifw and $iterw$ would need a term for each letter.

The definitions of the constructors will be more explicit with their reductions rules and their types. For intuition, the constructor $ifn(t,t')$ can be seen as $\lambda n.match\ n\ with\ succ(n') \mapsto t\ n' \mid 0 \mapsto t'$, and the constructor itern(V,t) is such that $itern(V,t)\ \underline{n} \to^* V^n\ t$, if $\underline{n}$ is the coding of the integer n, that is $succ^n(\mathsf{zero})$.

Reductions. Base reductions in $\mathsf{s}\ell\mathsf{T}$ are given by the rules described in Figure 1.

Note that in the $iterw$ rule, the order in which we apply the steps functions is the reverse of the one for iterators we see usually. In particular, it does not correspond to the reduction defined in [4]. This is not a

$$
\begin{array}{r|l}
(\lambda x.t)\ V \to t[V/x] & let\ x \otimes y = V \otimes V'\ in\ t \to t[V/x][V'/y] \\
ifn(V,V')\ \mathsf{zero} \to V' & ifn(V,V')\ succ(W) \to V\ W \\
itern(V,V')\ \mathsf{zero} \to V' & itern(V,V')\ succ(W) \to itern(V, V\ V')\ W \\
ifw(V_0,V_1,V')\ \epsilon \to V' & ifw(V_0,V_1,V')\ s_i(W) \to V_i\ W \\
iterw(V_0,V_1,V')\ \epsilon \to V' & iterw(V_0,V_1,V')\ s_i(W) \to iterw(V_0,V_1,V_i\ V')\ W \\
if(V,V')\ \mathsf{tt} \to V & if(V,V')\ \mathsf{ff} \to V'
\end{array}
$$

Fig. 1. Base rules for $\mathsf{s}\ell\mathsf{T}$

problem since we can compute the mirror of a word and the subject reduction is easier to prove with this definition.

Those base reductions can be applied in contexts C defined by the following grammar :
$C := [] \mid C\ t \mid V\ C \mid C \otimes t \mid t \otimes C \mid let\ x \otimes y = C\ in\ t \mid succ(C) \mid ifn(C,t) \mid ifn(t,C) \mid itern(V,C)$
$\mid s_i(C) \mid ifw(C,t,t') \mid ifw(t,C,t') \mid ifw(t,t',C) \mid iterw(V_0,V_1,C) \mid if(C,t) \mid if(t,C)$

Linear Types with Sizes. Base types are given by the following grammar :
$U := W^I \mid N^I \mid B$ $\qquad\qquad I, J, \ldots := a \mid n \in \mathbb{N}^* \mid I + J \mid I \cdot J$

I represents an index and a represents an index variable. We define for indexes the notions of free variables and free occurrences in the obvious way and we work up to renaming of variables. We also define the substitution of a free variable in an index in the obvious way.

For two indexes I and J, we say that $I \leqslant J$ if for any valuation ϕ mapping free variables of I and J to non-zero integers, we have $I_\phi \leqslant J_\phi$. I_ϕ is I where free variables have been replaced by their value in ϕ, thus I_ϕ is a non-zero integer. We now consider that if $I \leqslant J$ and $J \leqslant I$ then $I = J$ (ie we take the quotient set for the equivalence relation). Remark that by definition of indexes, we always have $1 \leqslant I$.

For two indexes I and J, we say that $I < J$ if for any valuation ϕ mapping free variables of I and J to non-zero integers, we have $I_\phi < J_\phi$. This is not equivalent to $I \leqslant J$ and $I \neq J$, as you can see with $a \leqslant a \cdot b$.

In this work, we only consider polynomial indexes, this is a huge restriction from usual linear dependent types, used for example in [4] or [12], in which they consider any set of functions described by some rewrite rules, but in this work we only want to use $\mathsf{s}\ell\mathsf{T}$ to characterize polynomial time computation.

Now, the types are given by the following grammar :
$D := U \mid D \multimap D' \mid D \otimes D'$

We define a subtyping order $\sqsubset$ on types given by the following rules :

- $B \sqsubset B$ and if $I \leqslant J$ then $N^I \sqsubset N^J$ and $W^I \sqsubset W^J$.
- $D_1 \multimap D_1' \sqsubset D_2 \multimap D_2'$ iff $D_2 \sqsubset D_1$ and $D_1' \sqsubset D_2'$
- $D_1 \otimes D_1' \sqsubset D_2 \otimes D_2'$ iff $D_1 \sqsubset D_2$ and $D_1' \sqsubset D_2'$

Typing Rules and Weight. *Variables contexts* are denoted Γ, with the shape $\Gamma = x_1 : D_1, \ldots, x_n : D_n$. We say that Γ is included in Γ' when for all variable that appears in Γ, then it also appears in Γ' with the same type. We say that $\Gamma \sqsubset \Gamma'$ when Γ and Γ' have exactly the same variables, and for $x : D$ in Γ and $x : D'$ in Γ' we have $D \sqsubset D'$.

Ground variables contexts, denoted $d\Gamma$, are variables contexts in which all types are base types.

We write $\Gamma = \Gamma', d\Gamma$ to denote the decomposition of Γ into a ground variable context $d\Gamma$ and a variable context Γ' in which types are non-base types.

For a variable context without base types, we note $\Gamma = \Gamma_1, \Gamma_2$ when Γ is the concatenation of Γ_1 and Γ_2, and Γ_1 and Γ_2 do not have any common variables.

We denote proofs as $\pi \lhd \Gamma \vdash t : D$ and we define an index $\omega(\pi)$ called the *weight* for such a proof. The idea is that the weight will be an upper-bound for the number of reduction steps of t. Note that since $\omega(\pi)$ is an index, this bound can depend of some index variables. The rules for those proofs are described by Figure 2 and Figure 3.

The typing rules for ifn differs from the one defined in [4], but since we do not consider type inference in this work, it allows us to simplify the rule to take something more intuitive. Remark that the if-rule is *multiplicative*, that means that the contexts are separated in the two premises. This gives us easier proofs for the following, but there is a way to keep the same contexts in the premises, see the appendix 6 for more details.

Note that in the rule for *itern* described in Figure 3, the index variable a must be a fresh variable.

The weight also differs from the one defined in [4], since here we focus on a measure for reductions of terms, whereas in this paper, the authors work on a measure for reductions in a CEK abstract machine.

$$\pi \lhd \frac{D \sqsubset D'}{\Gamma, x : D \vdash x : D'} \qquad \omega(\pi) = 1$$

$$\pi \lhd \frac{\sigma \lhd \Gamma, x : D \vdash t : D'}{\Gamma \vdash \lambda x.t : D \multimap D'} \qquad \omega(\pi) = 1 + \omega(\sigma)$$

$$\pi \lhd \frac{\sigma_1 \lhd \Gamma_1, d\Gamma \vdash t : D' \multimap D \qquad \sigma_2 \lhd \Gamma_2, d\Gamma \vdash t' : D'}{\Gamma_1, \Gamma_2, d\Gamma \vdash t\ t' : D} \qquad \omega(\pi) = \omega(\sigma_1) + \omega(\sigma_2)$$

$$\pi \lhd \frac{\sigma_1 \lhd \Gamma_1, d\Gamma \vdash t : D \qquad \sigma_2 \lhd \Gamma_2, d\Gamma \vdash t' : D'}{\Gamma_1, \Gamma_2, d\Gamma \vdash t \otimes t' : D \otimes D'} \qquad \omega(\pi) = \omega(\sigma_1) + \omega(\sigma_2) + 1$$

$$\pi \lhd \frac{\sigma_1 \lhd \Gamma_1, d\Gamma \vdash t : D \otimes D' \qquad \sigma_2 \lhd \Gamma_2, d\Gamma, x : D, y : D' \vdash t' : D''}{\Gamma_1, \Gamma_2, d\Gamma \vdash let\ x \otimes y = t\ in\ t' : D''} \qquad \omega(\pi) = \omega(\sigma_1) + \omega(\sigma_2)$$

Fig. 2. Part 1 of the type system for $\mathsf{s}\ell\mathsf{T}$

$$\pi \lhd \frac{}{\Gamma \vdash \mathsf{zero} : N^I} \qquad \omega(\pi) = 0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \vdash t : N^J \qquad J + 1 \leqslant I}{\Gamma \vdash succ(t) : N^I} \qquad \omega(\pi) = \omega(\sigma)$$

$$\pi \lhd \frac{\sigma_1 \lhd \Gamma_1, d\Gamma \vdash t : N^I \multimap D \qquad \sigma_2 \lhd \Gamma_2, d\Gamma \vdash t' : D}{\Gamma_1, \Gamma_2, d\Gamma \vdash ifn(t, t') : N^I \multimap D} \qquad \omega(\pi) = \omega(\sigma_1) + \omega(\sigma_2) + 1$$

$$\pi \lhd \frac{\begin{array}{ccc} D \sqsubset E & E \sqsubset E[a+1/a] & E[I/a] \sqsubset F \\ \sigma_1 \lhd d\Gamma \vdash V : D \multimap D[a+1/a] & & \sigma_2 \lhd \Gamma, d\Gamma \vdash t : D[1/a] \end{array}}{\Gamma, d\Gamma \vdash itern(V, t) : N^I \multimap F} \qquad \omega(\pi) = I + \omega(\sigma_2) + I \cdot \omega(\sigma_1)[I/a]$$

$$\pi \lhd \frac{}{\Gamma \vdash \mathsf{tt}(or\ \mathsf{ff}) : B} \qquad \omega(\pi) = 0$$

$$\pi \lhd \frac{\sigma_1 \lhd \Gamma_1, d\Gamma \vdash t : D \qquad \sigma_2 \lhd \Gamma_2, d\Gamma \vdash t' : D}{\Gamma_1, \Gamma_2, d\Gamma \vdash if(t, t') : B \multimap D} \qquad \omega(\pi) = \omega(\sigma_1) + \omega(\sigma_2) + 1$$

Fig. 3. Part 2 of the type system for $\mathsf{s}\ell\mathsf{T}$

2.2 Some Examples of Programs in sℓT

We give some examples of programs in sℓT to compute polynomials. We detail more precisely one possible term for addition and one for multiplication for unary integers with their weight. We also give some examples on how to work with binary integers with type W.

Length of a word. We can give a term *length* with $\cdot \vdash length : W^I \multimap N^I$ that computes the length of a word : $length = iterw(\lambda n.succ(n), \lambda n.succ(n), \mathsf{zero})$.

Reverse of a word, and mirror iterator. We can compute the reverse of a word $(a_0 a_1 \dots a_n \mapsto a_n \dots a_1 a_0)$ with the term $rev = iterw(\lambda w.s_0(w), \lambda w.s_1(w), \epsilon) : W^I \multimap W^I$. Recall that the iterator on words in sℓT works in the reverse order, that is why this term is not identity but is indeed the term computing the reverse of a word.

Now we define $ITERW(V_0, V_1, t) = \lambda w.(iterw(V_0, V_1, t)(rev\ w)$ that is the iterator on words with the right order $(ITERW(V_0, V_1, t)\ s_{i_1}(s_{i_2}(\dots s_{i_n}(\epsilon)\dots)) \to^* V_{i_1}(V_{i_2}(\dots V_{i_n}(t)\dots))$. The typing rule we can make for this constructor is exactly the same as the one for *iterw*.

Iterator with base type argument. We show that for integers we can construct a term $REC(V,t)$ such that $REC(V,t)\ \underline{n} \to^* V\ \underline{n-1}\ (V\ \underline{n-2}\ (\dots\ (V\ \mathsf{zero}\ t)\dots))$.

$$REC(V,t) = \lambda n.let\ x \otimes y = (itern(\lambda r \otimes n'.(V n' r) \otimes succ(n'), t \otimes \mathsf{zero})\ n)\ in\ x$$

We can give this constructor a typing rule close to the one for the iteration, with an additional argument in the step term of type N^a.

$$\dfrac{\begin{array}{ccc} D \sqsubset E & E \sqsubset E[a+1/a] & E[I/a] \sqsubset F \\ \multicolumn{2}{c}{d\Gamma \vdash V : N^a \multimap D \multimap D[a+1/a]} & \Gamma, d\Gamma \vdash t : D[1/a] \end{array}}{\Gamma, d\Gamma \vdash REC(V,t) : N^I \multimap F}$$

In the same way, we could define a similar constructor for words.

Addition for unary words. The addition can be written in sℓT :

$$\dfrac{\pi_{add}(I,J) \lhd \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{N^{I+a} \sqsubset N^{I+a}}{x : N^I, y : N^{I+a} \vdash y : N^{I+a}} \quad I+a+1 \leqslant I+a+1}{x : N^I, y : N^{I+a} \vdash succ(y) : N^{I+a+1}}}{x : N^I \vdash \lambda y.succ(y) : N^{I+a} \multimap N^{I+a}[a+1/a]} \quad \dfrac{N^I \sqsubset N^{I+1}}{x : N^I \vdash x : N^{I+a}[1/a]} \quad \begin{array}{c} N^{I+a} \sqsubset N^{I+a} \\ N^{I+a} \sqsubset N^{I+a}[a+1/a] \\ N^{I+a}[J/a] \sqsubset N^{I+J} \end{array}}{x : N^I \vdash itern(\lambda y.succ(y), x) : N^J \multimap N^{I+J}}}{\cdot \vdash \lambda x.itern(\lambda y.succ(y), x) : N^I \multimap N^J \multimap N^{I+J}}}{}$$

$add = \lambda x.itern(\lambda y.succ(y), x)$, $\pi_{add}(I,J) \lhd \cdot \vdash add : N^I \multimap N^J \multimap N^{I+J}$.

And the rules give us, for two integers n and m, $add\ \underline{n}\ \underline{m} \to itern(\lambda y.succ(y), \underline{n})\ \underline{m} \to^* (\lambda y.succ(y))^m\ \underline{n} \to^* \underline{n+m}$

The weight of this term is $\omega_{add}(I,J) = 1 + J + 1 + J \cdot (1+1)[J/a] = 3J + 2$

Multiplication for unary words. The multiplication can be written in sℓT :

$$\pi_{mult}(I,J) \lhd \cfrac{\cfrac{\cfrac{\cfrac{N^I \sqsubset N^I}{x : N^I \vdash x : N^I} \quad \cfrac{\pi_{add}(I, a\cdot I)}{x : N^I \vdash add : N^I \multimap N^{a\cdot I} \multimap N^{(a+1)\cdot I}}}{x : N^I \vdash add\ x : N^{a\cdot I} \multimap N^{(a+1)\cdot I}} \quad x : N^I \vdash \mathsf{zero} : N^I \quad \begin{array}{c} N^{a\cdot I} \sqsubset N^{a\cdot I} \\ N^{a\cdot I} \sqsubset N^{a\cdot I}[a+1/a] \\ N^{a\cdot I}[J/a] \sqsubset N^{I\cdot J} \end{array}}{x : N^I \vdash itern(add\ x, \mathsf{zero}) : N^J \multimap N^{I\cdot J}}}{\cdot \vdash \lambda x.itern(add\ x, \mathsf{zero}) : N^I \multimap N^J \multimap N^{I\cdot J}}$$

With $mult = \lambda x.itern(add\ x, \mathsf{zero}) : N^I \multimap N^J \multimap N^{I\cdot J}$.

And this term is indeed the multiplication : $mult\ \underline{n}\ \underline{m} \to^* \underline{nm}$.

The weight of this term is $\omega_{mult}(I,J) = 1 + J + 1 + J \cdot (1 + 3I \cdot a + 2)[J/a] = 2 + 4J + 3IJ^2$

To be rigorous, the sub-term $add\ x$ is not a value, so we cannot write $itern(add\ x, \mathsf{zero})$, but we can use the usual η-rule and write the term $itern(\lambda y.add\ x\ y, \mathsf{zero})$.

With the multiplication, the addition and the fact that variables of base types can be duplicated, we can now compute the polynomials in sℓT .

Addition on binary integers. Now, we define some terms working on integers written in binary, with type W^I. First, we define an addition on binary integers in sℓT with a control on the number of bits. More precisely, we give a term $Cadd : N^I \multimap W^{J_1} \multimap W^{J_2} \multimap W^I$ such that $Cadd\ n\ w_1\ w_2$ outputs the least significant n bits of the sum $w_1 + w_2$. For example, $Cadd$ 3 101 110 = 011, and $Cadd$ 5 101 110 = 01011. This will usually be used with a n greater than the expected number of bits, the idea being that those extra 0 can be useful for some other programs. If we want an exact addition, we can for example use this function with n being $length(w_1) + length(w_2)$, and then use a function to erase the extra zeros (we obtain then a type $W^I \multimap W^J \multimap W^{I+J}$). The term follows the usual idea for addition, we use a supplementary boolean to keep track of the carry. For simplification, we do not give an explicit term but we show that we have to use conditionals and work on each cases one by one.

$Cadd = \lambda n, w_1, w_2.\ let\ c' \otimes r' \otimes w_1' \otimes w_2' = itern(\lambda c \otimes r \otimes w \otimes w'.\ match\ c, w, w'\ with$

$(\mathsf{ff}, \epsilon, \epsilon) \mapsto \mathsf{ff} \otimes s_0(r) \otimes \epsilon \otimes \epsilon \mid \dots \mid (\mathsf{tt}, s_1(v), s_1(v')) \mapsto \mathsf{tt} \otimes s_1(r) \otimes v \otimes v', \mathsf{ff} \otimes \epsilon \otimes (rev\ w_1) \otimes (rev\ w_2))\ n\ in\ r.$

For the typing of this term, we use in the iteration the type $B \otimes W^a \otimes W^{J_1} \otimes W^{J_2}$, with c representing the carry, r the current result, and w, w' the binary integers that we read from right to left.

Unary integers to binary integers. We define a term $Cunarytobinary : N^I \multimap N^J \multimap W^I$ such that on the input n, n', this term computes the least n significant bit of the representation of n' in binary. As previously, we could define a term giving exactly the binary representation by taking an upper-bound of the size for the first argument and then erasing the extra zeros.

$$Cunarytobinary = \lambda n.itern(\lambda w.Cadd\ n\ w\ (s_1(\epsilon)), Cadd\ n\ \epsilon\ \epsilon)$$

Binary integers to unary integers. We would like a way to compute the unary integer for a given binary integer. However, this function is exponential in the size of its input, so it is impossible to write such a function in sℓT . Nevertheless, given an additional information bounding the size of this unary word, we can give a term $Cbinarytounary : N^I \multimap W^J \multimap N^I$ such that on an input n, w this term computes the minimum between n and the unary representation of w. First we describe a term $min : N^I \multimap N^J \multimap N^I$.

$min = \lambda n, n'.let\ r_0 \otimes n_0 = (itern(\lambda r_1 \otimes n_1.ifn(\lambda p.succ(r_1) \otimes p, r_1 \otimes \mathsf{zero})\ n_1, \mathsf{zero} \otimes n')\ n)\ in\ r_0$

In order to type this term, we use in the iteration the type $N^a \otimes N^J$. Remark that this term allows us to erase the index J. Now that we have this term, we can define the following term :

$Cbinarytounary =$
$\lambda n.iterw(\lambda n'.min\ n\ (mult\ n'\ \underline{2}), \lambda n'.min\ n\ succ(mult\ n'\ \underline{2}), \mathsf{zero})$

2.3 Properties of the Type System and Terms

In order to prove the subject reduction for $\mathsf{s}\ell\mathsf{T}$ and that the weight is a bound on the number of reduction steps of a term, we successively prove lemmas leading to usual substitution lemmas.

Values and Closed Normal Forms. First, we show that values are indeed linked to normal forms. In particular, this theorem shows that a value of type integer is indeed of the form $succ(succ(\ldots(succ(\mathsf{zero}))\ldots))$. This imposes that in this call-by-value calculus, when an argument is of type N, it is the encoding of an integer.

Theorem 1. *Let t be a term in $\mathsf{s}\ell\mathsf{T}$, if t is closed and has a typing derivation $\vdash t : D$ then : t is normal if and only if t is a value V.*

The proof of this theorem can be found in the appendix, section 6.1. This is an usual proof by induction for the two implications relying on the definition of contexts.

Index Variable Substitution, Weakening and Subtyping. We will give some intermediate lemmas in order to prove the main theorem and some intuitions for the proofs. More details can be found in the appendix.

Lemma 1 (Weakening). *Let Δ, Γ be disjoint typing contexts, and $\pi \lhd \Gamma \vdash t : D$ then we have a proof $\pi' \lhd \Gamma, \Delta \vdash t : D$ with $\omega(\pi) = \omega(\pi')$.*

Lemma 2. *Let I, J, K be indexes, a, b index variables with a not free in K.*
Then $I[J/a][K/b] = I[K/b][J[K/b]/a]$

Lemma 3 (Index substitution). *Let I be an index.*
1) Let J_1, J_2 be indexes such that $J_1 \leqslant J_2$ then $J_1[I/a] \leqslant J_2[I/a]$
1') Let J_1, J_2 be indexes such that $J_1 < J_2$ then $J_1[I/a] < J_2[I/a]$
2) Let D, D' be types such that $D \sqsubset D'$ then $D[I/a] \sqsubset D'[I/a]$
3) If $\pi \lhd \Gamma \vdash t : D$ then $\pi[I/a] \lhd \Gamma[I/a] \vdash t : D[I/a]$
4) $\omega(\pi[I/a]) = \omega(\pi)[I/a]$

Point 1 and 1' are by definition of $\leqslant$ and $<$, then point 2 is a direct induction on types using point 1 for base types. Points 3 and 4 are proved by induction on π. Point 1 is used in the successors rules, and point 2 is used in the axiom rule. The only interesting cases are iterations, and one can find the proof for the *itern* case in the appendix, section 6.1

Lemma 4 (Monotonic index substitution). *Take J_1, J_2 such that $J_1 \leqslant J_2$*
1) Let I be an index, then $I[J_1/a] \leqslant I[J_2/a]$
2) $\omega(\pi[J_1/a]) \leqslant \omega(\pi[J_2/a])$
3) Let E be a type.
If $E \sqsubset E[a+1/a]$ then $E[J_1/a] \sqsubset E[J_2/a]$ and if $E[a+1/a] \sqsubset E$ then $E[J_2/a] \sqsubset E[J_1/a]$

Point 1 can be proved by induction on indexes, and then point 2 is just a particular case of point 1, by lemma 3.4. Point 3 is proved by induction on E.

Lemma 5. *If $\pi \lhd \Gamma, d\Gamma \vdash V : U$ then we have a proof $\pi' \lhd d\Gamma \vdash V : U$ with $\omega(\pi) = \omega(\pi')$. Moreover, $\omega(\pi') \leqslant 1$.*

This is easily proved by looking which values can be typed with a base type, and observing that the axiom rule is only used with a base typed variable. The other rules that can be used are zero-like and *succ*-like rules, for which the weight does not increase. That is why the total weight is smaller than the weight for the axiom rule, ie 1.

Lemma 6 (Monotonic $d\Gamma$). *If $\pi \lhd \Gamma, d\Gamma \vdash t : D$ then for all subproof $\sigma \lhd \Gamma' \vdash t' : D'$ of π, $d\Gamma$ is included in the context Γ'.*

This can be proved directly by induction on π.

Lemma 7 (Subtyping). *If $\pi \lhd \Gamma \vdash t : D$ then for all Γ', D' such that $D \sqsubset D'$ and $\Gamma' \sqsubset \Gamma$, we have a proof $\pi' \lhd \Gamma' \vdash t : D'$ with $\omega(\pi') \leqslant \omega(\pi)$*

This can be proved by induction on π. The only interesting cases are for iterations, in which case the property directly follows from point 2 and 3 of lemma 4. This lemma shows that we do not need an explicit subtyping rule.

Term Substitution Lemmas. In order to prove the subject reduction of the calculus, we explicit what happens during a substitution of a value in a term. We need to work on two cases, first a substitution of variables with base types, that is to say duplicable variables, and then variables with a non-base type for which the type system imposes linearity.

Lemma 8 (Base value substitution). *If $\pi \lhd \Gamma, d\Gamma, x : U \vdash t : D$ and $\sigma \lhd d\Gamma \vdash V : U$ then we have a proof $\pi' \lhd \Gamma, d\Gamma \vdash t[V/x] : D$. The proof π' is π in which we replace the occurrences of axiom rules $\Gamma', x : U \vdash x : U'$ by the proof $\sigma' \lhd \Gamma' \vdash V : U'$ given by the weakening lemma ($d\Gamma$ is in Γ' by lemma 6) and the subtyping lemma. Moreover, $\omega(\pi') \leqslant \omega(\pi)$.*

This is proved by induction on π. Remark that x can appear several times in t since x is typed by a base type. The most interesting case is the axiom case for the variable x, that we develop here. The other cases are direct.

$$\pi \lhd \frac{U \sqsubset U'}{\Gamma, d\Gamma, x : U \vdash x : U'} \qquad\qquad \omega(\pi) = 1$$

We have $\sigma \lhd d\Gamma \vdash V : U$, by the lemma 5, we have $\omega(\sigma) \leqslant 1 = \omega(\pi)$. By the weakening and subtyping lemmas, we have $\pi' = \sigma' \lhd \Gamma, d\Gamma \vdash V : U'$ and $\omega(\pi') \leqslant \omega(\sigma) \leqslant \omega(\pi)$.

Lemma 9 (Non-base value substitution). *If $\pi \lhd \Gamma_1, d\Gamma, x : D' \vdash t : D$ with D' not a base type, and $\sigma \lhd \Gamma_2, d\Gamma \vdash V : D'$ then we have a proof $\pi' \lhd \Gamma_1, \Gamma_2, d\Gamma \vdash t[V/x] : D$. The proof π' is π in which we add Γ_2 in all contexts in the branch where x appears and we replace the occurrences of axiom rules $\Gamma', x : D' \vdash x : D''$ by the proof $\sigma' \lhd \Gamma', \Gamma_2 \vdash V : D''$ given by the weakening lemma and the subtyping lemma. Moreover, $\omega(\pi') \leqslant \omega(\pi) + \omega(\sigma)$.*

This is proved by induction on π. The bound on the weight holds for the axiom rule by the subtyping lemma. In multiplicative rules such as application and if, the property holds by the fact that x only appears in one of the premises, and so $\omega(\sigma)$ appears only once in the total weight. Finally, for the iteration rules, the property holds since x cannot appear in the step terms.

2.4 Main Theorem : The Weight Controls Reductions

In this section we express the subject-reduction of the calculus and the fact that the weight of a proof strictly decreases during a reduction.

Theorem 2. *Let $\tau \lhd \Gamma \vdash t_0 : D$, and $t_0 \to t_1$, then there is a proof $\tau' \lhd \Gamma \vdash t_1 : D$ such that $\omega(\tau') < \omega(\tau)$.*

The proof of this theorem can be found in the appendix, in section 6.2. The main difficulty is to prove this result for base reductions. Base reductions that induces a substitution, like the usual β reduction, are proved by using the substitution lemmas given previously. The others interesting cases are the rules for iterators. For such a rule, the subject reduction is given by a good use of the fresh variable given in the typing rule. This is a rather direct proof by using the previous lemmas on indexes and typing derivations.

As the indexes can only define polynomials, the weight of a sequent can only be a polynomial on the index variables. And so, in $\mathsf{s\ell T}$, we can only define terms that works in polynomial time on their inputs.

2.5 Polynomial Indexes and Degree

For the following section on the elementary affine logic, we need to define a notion of *degree* of indexes and explicit some properties of this notion. The indexes can be seen as multi-variables polynomials, and we can define the degree of an index I by induction on I :

- $\forall n \in \mathbb{N}^*, d(n) = 0$
- For an index variable a, $d(a) = 1$
- $d(I + J) = max(d(I), d(J))$
- $d(I \cdot J) = d(I) + d(J)$.

We have the following properties between indexes and degree :

Theorem 3 (Degree). *1) Let I be an index and $k \in \mathbb{N}^*$, $I[k/a] \leqslant k^{d(I)} \cdot I[1/a]$.*
2) If $I \leqslant J$ then $d(I) \leqslant d(J)$.

The first point is proved by induction on I. For the second point, the proof can be found in the appendix, section 6.1. If one thinks of indexes as polynomials, this second point is really intuitive.

By the point 2, we also obtain that if $I \leqslant J$ and $J \leqslant I$ then $d(I) = d(J)$, and so the degree can be defined up to equivalence. This justifies that this definition is correct in our set of indexes with a quotient by the equivalence relation. This definition of degree is primordial for the control of reductions in the enriched EAL calculus, that we present in the following section.

3 Enriched EAL-Calculus

We work on an elementary affine lambda calculus based on [20] without multithreading and side-effects, that we present here. In order to solve the problem of intensional expressivity of this calculus, we enrich it with constructors for integers, words and booleans, and some iterators on those types following the usual constraint on iteration in elementary affine logic (EAL). Then, using the fact that the proof of correctness in [20] is robust enough to support functions computable in polynomial time with type $N \multimap N$ (see Section 6.3 in the appendix), we enrich EAL with the polynomial time calculus defined previously. More precisely, we add the possibility to use first-order $\mathsf{s}\ell\mathsf{T}$ terms in this calculus in order to work on those base types, particularly we can then do controlled iterations for those types. We then adapt the measure used in [20] to our calculus to find an upper-bound on the number of reductions for a term.

3.1 A Classical EAL-Calculus

First, let us present a λ-calculus for the classical elementary affine logic. In this calculus, any sequence of reduction terminates in elementary time. The keystone of this proof is the use of the modality "!", called *bang*, inspired by linear logic. In order to have this bound, there are some restrictions in the calculus like linearity (or *affinity* if we allow weakening) and an important notion linked with the "!" is used, the *depth*. We follow the presentation from [20] and we encode the usual restrictions in a type system.

Terms and semantics. Terms are given by the following grammar :

$M := x \mid \lambda x.M \mid M\ M' \mid !M \mid let\ !x = M\ in\ M'$

The constructor $let\ !x = M\ in\ M'$ binds the variable x in M'. We define as usual the notion of free variables, free occurrences and substitution.

The semantic of this calculus is given by the two following rules :

$(\lambda x.M)\ M' \to M[M'/x]$ $\qquad$ $let\ !x = !M\ in\ M' \to M'[M/x]$.

Those rules can be applied in any contexts.

Type system. We add to this calculus a polymorphic type system that also restraints the possible term we can write.

Types are given by the following grammar :

$T := \alpha \mid T \multimap T' \mid\ !T \mid \forall\alpha.T$

Linear variables contexts are denoted Γ, with the shape $\Gamma = x_1 : T_1, \dots, x_n : T_n$. We write Γ_1, Γ_2 the disjoint union between Γ_1 and Γ_2.

Global variables contexts are denoted Δ, with the shape $\Delta = x_1 : T_1, \dots, x_n : T_n, y_1 : [T'_1], \dots y_n : [T'_m]$. We say that [T] is a *discharged type*, as we could see in light linear logic, see for example [15] and [22]. When we need to separate the discharged types from the others, we will write $\Delta = \Delta'', [\Delta']$". In this case, if $[\Delta'] = y_1 : [T'_1], \dots, y_m : [T'_m]$, then we note $\Delta = y_1 : T'_1, \dots, y_m : T'_m$.

Typing judgments have the shape $\Gamma \mid \Delta \vdash M : T$.

$$\dfrac{}{\Gamma, x : T \mid \Delta \vdash x : T}\ (\text{Lin Ax}) \qquad \dfrac{}{\Gamma \mid \Delta, x : T \vdash x : T}\ (\text{Glob Ax})$$

$$\dfrac{\Gamma, x : T \mid \Delta \vdash M : T'}{\Gamma \mid \Delta \vdash \lambda x.M : T \multimap T'}\ (\lambda\text{-Abs}) \qquad \dfrac{\Gamma \mid \Delta \vdash M : T' \multimap T \qquad \Gamma' \mid \Delta \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash M\ M' : T}\ (\text{App})$$

$$\dfrac{\varnothing \mid \Delta \vdash M : T}{\Gamma \mid \Delta', [\Delta] \vdash !M :\ !T}\ (!\ \text{Intro}) \qquad \dfrac{\Gamma' \mid \Delta \vdash M :\ !T \qquad \Gamma \mid \Delta, x : [T] \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash let\ !x = M\ in\ M' : T'}\ (!\ \text{Elim})$$

$$\dfrac{\Gamma \mid \Delta \vdash M : T \qquad \alpha \text{ fresh in } \Gamma, \Delta}{\Gamma \mid \Delta \vdash M : \forall\alpha.T}\ (\forall\ \text{Intro}) \qquad \dfrac{\Gamma \mid \Delta \vdash M : \forall\alpha.T}{\Gamma \mid \Delta \vdash M : T[T'/\alpha]}\ (\forall\ \text{Elim})$$

Fig. 4. Type system for the classical EAL

The rules are given in Figure 4. Observe that all the rules are multiplicative for Γ, and the "! Intro" rule erases linear contexts, non-discharged types and transforms discharged types into usual types. With this, we can see that some restrictions appears in a typed term. First, in $\lambda x.M$, x occurs at most once in M, and moreover, there is no "! Intro" rule behind the axiom rule for x. Then, in $let\ !x = M\ in\ M'$, x can be duplicated, but there is exactly one "! Intro" rule behind each axiom rule for x. For example, with this type system, we can not type terms like $\lambda x.!x$, $\lambda f, x.f\ (f\ x)$ or $let\ !x = M\ in\ x$.

With this type system, we obtain as a consequence of the results exposed in [20] that any sequence of reductions of a typed term terminates in elementary time. This proof relies on the notion of depth linked with the modality "!" and a measure on terms bounding the number of reduction for this term. We will adapt and work with those two notions in the following part on the enriched EAL calculus, but for now, let us present some terms and encoding in this calculus.

Examples of terms in EAL and Church integers. A useful term is the term proving the functoriality of the modality ! :

$fonct = \lambda f, x.let\ !g = f\ in\ let\ !y = x\ in\ !(g\ y) : \forall\alpha, \alpha'.!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'$.

$$\dfrac{\dfrac{\dfrac{}{f :!(\alpha \multimap \alpha') \mid \cdot \vdash f :!(\alpha \multimap \alpha')} \qquad \dfrac{\dfrac{}{x :!\alpha \mid g : [\alpha \multimap \alpha'] \vdash x :!\alpha} \qquad \dfrac{\dfrac{\begin{matrix}\cdot \mid g : \alpha \multimap \alpha', y : \alpha \vdash g : \alpha \multimap \alpha' \\ \cdot \mid g : \alpha \multimap \alpha', y : \alpha \vdash y : \alpha\end{matrix}}{\cdot \mid g : \alpha \multimap \alpha', y : \alpha \vdash g\ y : \alpha'}}{\cdot \mid g : [\alpha \multimap \alpha'], y : [\alpha] \vdash !(g\ y) :!\alpha'}}{x :!\alpha \mid g : [\alpha \multimap \alpha'] \vdash let\ !y = x\ in\ !(g\ y)}}{\dfrac{f :!(\alpha \multimap \alpha'), x :!\alpha \mid \cdot \vdash let\ !g = f\ in\ let\ !y = x\ in\ !(g\ y) :!\alpha'}{\cdot \mid \cdot \vdash \lambda f, x.let\ !g = f\ in\ let\ !y = x\ in\ !(g\ y) :\ !(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'}}}{\cdot \mid \cdot \vdash \lambda f, x.let\ !g = f\ in\ let\ !y = x\ in\ !(g\ y) : \forall\alpha, \alpha'.!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'}$$

Integers can be encoded in this calculus, using the type $N = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$. For example, 3 is described by the term

$\underline{3} = \lambda f.let\ !g = f\ in\ !(\lambda x.g\ (g\ (g\ x))) : N$.

With this encoding, addition and multiplication can be defined, with type $N \multimap N \multimap N$.

$add = \lambda n, m, f.let\ !f' = f\ in\ let\ !g = n\ !f'\ in\ let\ !h = m\ !f'\ in\ !(\lambda x.h\ (g\ x))$

$mult = \lambda n, m, f.let\ !g = f\ in\ n(m\ !f)$

And finally, one can also define an iterator using integers

$iter = \lambda f, x, n.fonct\ (n\ f)\ x : \forall\alpha.!(\alpha \multimap \alpha) \multimap !\alpha \multimap N \multimap !\alpha$ such that $iter\ !M\ !M'\ \underline{n} \rightarrow^* !(M^n\ M')$.

Intensional expressivity. Those examples show that this calculus suffers from limitation. First, we need to work with Church integers, because of a lack of data structure. Furthermore, we need to be careful with the modality, and this can be sometime a bit tricky, as one can remark with the addition. And finally if we want to do an iteration, we are forced to work with type with bangs. This implies that each time we need to use an iteration, we are forced to add a bang in the final type. However, it has been proved [6] that polynomial and exponential complexity classes can be characterized in this calculus, by fixing types. For example, with a type for words W and booleans B we have that $!W \multimap !!B$ characterizes polynomial time computation. But with this type, the restriction explained above imposes that one can only use once a term like $iter$, and so, some natural polynomial time programs cannot be typed with the type $!W \multimap !!B$. We say that this calculus has a limited intensional expressivity.

One goal of this paper is to try to solve this problem, and for that, we now present an enriched version of this EAL-calculus, using the language $\mathsf{s}\ell\mathsf{T}$ defined previously.

3.2 Notations

Applications. For an object with a notion of application M and an integer n, we write $M^n M'$ to denote n applications of M to M'. In particular, $M^0 M' = M'$

We also define for a word w, given objects M_a for all letter a, $M^w M'$. This is defined by induction on words with $M^\epsilon M' = M'$ and $M^{aw'} M' = M_a\ (M^{w'} M')$

Notations for vectors. In the following we will work with vectors of $\mathbb{N}^{n+1}$, for $n \in \mathbb{N}$. We introduce here some notations on those vectors.

We usually denote vectors by $\mu = (\mu(0), \ldots, \mu(n))$

When there is no ambiguity with the value of n, for $0 \leqslant k \leqslant n$, we note $\mathbb{1}_k$ the vector μ with $\mu(k) = 1$ and $\forall i, 0 \leqslant i \leqslant n, i \neq k, \mu(i) = 0$. We extend this notation for $k > n$. In this case, $\mathbb{1}_k$ is the zero-vector.

Let $\mu_0 \in \mathbb{N}^{n+1}$ and $\mu_1 \in \mathbb{N}^{m+1}$. We denote $\mu = (\mu_0, \mu_1) \in \mathbb{N}^{m+n+2}$ the vector with $\forall i, 0 \leqslant i \leqslant n, \mu(i) = \mu_0(i)$ and $\forall i, 0 \leqslant i \leqslant m, \mu(i + n + 1) = \mu_1(i)$.

Let $\mu_0, \mu_1 \in \mathbb{N}^{n+1}$. We write $\mu_0 \leqslant \mu_1$ when $\forall i, 0 \leqslant i \leqslant n, \mu_0(i) \leqslant \mu_1(i)$. And we write $\mu_0 < \mu_1$ when $\mu_0 \leqslant \mu_1$ and $\mu_0 \neq \mu_1$.

Let $\mu_0, \mu_1 \in \mathbb{N}^{n+1}$. We write $\mu_0 \leqslant_{lex} \mu_1$ for the lexicographic order on vectors.

For $k \in \mathbb{N}$, when there is no ambiguity with the value of n, we write $\tilde{k}$ the vector μ such that $\forall i, 0 \leqslant i \leqslant n, \mu(i) = k$.

3.3 Syntax and Type System

Terms. Terms are defined by the following grammar :

$M := x \mid \lambda x.M \mid M\ M' \mid !M \mid let\ !x = M\ in\ M' \mid M \otimes M'$

$\mid let\ x \otimes y = M\ in\ M' \mid \mathsf{zero} \mid succ(M) \mid ifn(M, M') \mid iter^!_N(M, M') \mid \mathsf{tt} \mid \mathsf{ff} \mid if(M, M')$

$\mid \epsilon \mid s_0(M) \mid s_1(M) \mid ifw(M_0, M_1, M) \mid iter^!_W(M_0, M_1, M) \mid [\lambda x_n \ldots x_1.t](M_1, \ldots, M_n)$

Note that the t used in $[\lambda x_n \dots x_1.t](M_1, \dots, M_n)$ refers to terms defined in $\mathsf{s}\ell\mathsf{T}$. This notation means that we call the function t defined in $\mathsf{s}\ell\mathsf{T}$ with arguments $M_1, \dots, M_n$. Moreover, n can be any integer, even 0.

Constructors for iterations directly follow from the ones we can define usually in EAL for Church integers or Church words, as we could see in the previous section on classical EAL.

Once again, we often write s_i to denote s_0 or s_1, and the choice of the alphabet $\{0, 1\}$ is arbitrary, we could have used any finite alphabet.

In the following, we note $\underline{v}$ for base type values, defined by the following grammar

$\underline{v} := \mathsf{zero} \mid succ(\underline{v}) \mid \epsilon \mid s_i(\underline{v}) \mid \mathsf{tt} \mid \mathsf{ff}$

In particular, if n is an integer and w is a binary word, we note $\underline{n}$ for the base value $succ^n(\mathsf{zero})$, and $\underline{w} = \underline{w_1 \cdots w_n}$ for the base value $s_{w_1}(\dots s_{w_n}(\epsilon)\dots)$.

We define the size $|\underline{v}|$ of $\underline{v}$ by $|\mathsf{zero}| = |\epsilon| = |\mathsf{tt}| = |\mathsf{ff}| = 1$ and $|succ(\underline{v})| = |s_i(\underline{v})| = 1 + |\underline{v}|$.

As usual, we work up to α-isomorphism and we do not explicit the renaming of variables.

Reductions. Base reductions are defined by the rules given in Figure 5. Note that for some of these rules, for example the last one, $\underline{v}$ can denote either the $\mathsf{s}\ell\mathsf{T}$ term or the enriched EAL term.

$$
\begin{array}{c|c}
(\lambda x.M)\ M' \to M[M'/x] & let\ !x = !M\ in\ M' \to M'[M/x] \\
let\ x \otimes y = M \otimes M'\ in\ N \to N[M/x][M'/y] & ifn(M, M')\ \mathsf{zero} \to M' \\
ifn(M, M')\ succ(N) \to M\ N & iter^!_N(!M, !M')\ \underline{n} \to !(M^n\ M'),\ n \text{ integer} \\
ifw(M_0, M_1, M)\ \epsilon \to M & ifw(M_0, M_1, M)\ s_i(N) \to M_i\ N \\
iter^!_W(!M_0, !M_1, !M')\ \underline{w} \to !(M^w\ M'),\ w \text{ binary word} & if(M, M')\ \mathsf{tt} \to M \\
if(M, M')\ \mathsf{ff} \to M' & \text{if } t \to t' \text{ in } \mathsf{s}\ell\mathsf{T}\ ,\ [t]() \to [t']()
\end{array}
$$

$$
[\lambda x_n \dots x_1.t](M_1, \dots, M_{n-1}, \underline{v}) \to [\lambda x_{n-1} \dots x_1.t[\underline{v}/x_n]](M_1, \dots, M_{n-1})
$$

$$
[\underline{v}]() \to \underline{v}
$$

Fig. 5. Base rules for enriched EAL-calculus

Those reductions can be extended to any contexts, and so we write $M \to M'$ if there is a context C and a base reduction $M_0 \to M'_0$ such that $M = C(M_0)$ and $M' = C(M'_0)$.

Types. Types are usual types for intuitionistic linear logic enriched with some base types for booleans, integers and words. Base types are given by the following grammar :

$A := B \mid N \mid W$

Types are given by the following grammar :

$T := A \mid T \multimap T' \mid !T \mid T \otimes T'$

Type System and Measure. *Linear variables contexts* are denoted Γ and *global variables contexts* are denoted Δ. They are defined in the same way as in the previous part on the classical EAL-calculus.

Typing judgments have the usual shape of dual contexts judgments $\pi \lhd \Gamma \mid \Delta \vdash M : T$. For such a proof π, and $i \in \mathbb{N}$, we define a *weight* $\omega_i(\pi) \in \mathbb{N}$.

For all $k, n \in \mathbb{N}$, we note $\mu^k_n(\pi) = (\omega_k(\pi), \dots, \omega_n(\pi))$, with the convention that if $k > n$, then $\mu^k_n(\pi)$ is the null-vector. We write $\mu_n(\pi)$ to denote the vector $\mu^0_n(\pi)$. In the definitions given in the type system, instead of defining $\omega_i(\pi)$ for all i, we define $\mu_n(\pi)$ for all n, from which one can recover the weights. We will often call $\mu_n(\pi)$ the *measure* of the proof π.

The *depth* of a proof (or a typed term) is the greatest integer i such that $\omega_i(\pi) \neq 0$. It is always defined for any proof.

The idea behind the definition of measure is to show that with a reduction step, this measure strictly decreases for the lexicographic order and we can control the growing of the weights.

$$\pi \lhd \frac{}{\Gamma, x : T \mid \Delta \vdash x : T} \qquad \mu_n(\pi) = \mathbb{1}_0$$

$$\pi \lhd \frac{}{\Gamma \mid \Delta, x : T \vdash x : T} \qquad \mu_n(\pi) = \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma, x : T \mid \Delta \vdash M : T'}{\Gamma \mid \Delta \vdash \lambda x.M : T \multimap T'} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : T' \multimap T \qquad \tau \lhd \Gamma' \mid \Delta \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash M\ M' : T} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \varnothing \mid \Delta \vdash M : T}{\Gamma \mid \Delta', [\Delta] \vdash !M : \ !T} \qquad \mu_n(\pi) = (1, \mu_{n-1}(\sigma))$$

$$\pi \lhd \frac{\sigma \lhd \Gamma' \mid \Delta \vdash M : \ !T \qquad \tau \lhd \Gamma \mid \Delta, x : [T] \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash let\ !x = M\ in\ M' : T'} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \tau \lhd \Gamma' \mid \Delta \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash M \otimes M' : T \otimes T'} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma' \mid \Delta \vdash M : T \otimes T' \qquad \tau \lhd \Gamma, x : T, y : T' \mid \Delta \vdash M' : T''}{\Gamma, \Gamma' \mid \Delta \vdash let\ x \otimes y = M\ in\ M' : T''} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

Fig. 6. Type system and measure for the classical EAL

$$\pi \lhd \frac{}{\Gamma \mid \Delta \vdash \mathsf{zero} : N} \qquad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : N}{\Gamma \mid \Delta \vdash succ(M) : N} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mathbb{1}_1$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : N \multimap T \qquad \tau \lhd \Gamma' \mid \Delta \vdash M' : T}{\Gamma, \Gamma' \mid \Delta \vdash ifn(M, M') : N \multimap T} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \ !(T \multimap T) \qquad \tau \lhd \Gamma' \mid \Delta \vdash M' : \ !T}{\Gamma, \Gamma' \mid \Delta \vdash iter^!_N(M, M') : N \multimap !T} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

Fig. 7. Type system and measure for constructors on integers

$$\pi \lhd \frac{\forall i, (1 \leqslant i \leqslant k), \sigma_i \lhd \Gamma_i \mid \Delta \vdash \ M_i : A_i \qquad \tau \lhd x_1 : A_1^{a_1}, \ldots, x_k : A_k^{a_k} \vdash_{\mathsf{s}\ell\mathsf{T}} t : A^I}{\Gamma, \Gamma_1, \ldots, \Gamma_k \mid \Delta \vdash [\lambda x_k \ldots x_1.t](M_1, \ldots, M_k) : A}$$

$$\mu_n(\pi) = \sum_{i=1}^{k} \mu_n(\sigma_i) + k(d(\omega(\tau) + I) + 1) \cdot \mathbb{1}_0 + ((\omega(\tau) + I)[1/b_1] \cdots [1/b_l] + 1) \cdot \mathbb{1}_1$$

where $\{b_1, \ldots, b_l\} = FV(\omega(\tau)) \cup FV(I)$.

Fig. 8. Typing rule and measure for the $\mathsf{s}\ell\mathsf{T}$ call

The rules are given by the Figure 6, Figure 7 and Figure 8.

The rules given in figure 6 represent the classic rules for EAL. Note that as we could see in the classical EAL calculus, those rules impose some restrictions in the use of variables. Indeed, for the terms $\lambda x.M$ and $let\ x \otimes y = M'\ in\ M$, the newly defined variables are used at most once in M, and if they appear, then we did not cross a !. For the term $let\ !x = M\ in\ M'$, the variable x can be used as many times as we want in M', but we need to cross exactly one time a ! to use the variable.

Remark that the constructors for base types values such as zero and $succ$ given in Figure 7 influence the weight only in position 1 and not 0 like the others constructors.

For the rule given by Figure 8, some explanations are necessary. The premise for t is a proof in $\mathsf{s}\ell\mathsf{T}$. In this proof, we add on each base types A_i an index, more precisely an index variable a_i. There is here an abuse of notation, since in $\mathsf{s}\ell\mathsf{T}$ there is no indexes on the boolean type B. So when $A_i = B$, we just do not put any index on the type B. The same goes for the type A, if A is the boolean type B, then there is no index I, and we just replace in the measure I by 1. The typing in $\mathsf{s}\ell\mathsf{T}$ give us a weight $\omega(\tau)$ and a size for the output I. The degree of those indexes influences the weight at position 0, and their values when all free variables are replaced by 1 influence the weight at position 1. Having the degree at position 0 will allow us the replacement of the arguments x_i by their values given by M_i, and the measure at position 1 will allow us to bound the number of reductions in $\mathsf{s}\ell\mathsf{T}$ and the size of the output.

Remark that when $k = 0$, the term $[t]()$ influences only the weight at position 1, as constructors for base types.

We include a weakening in this rule, in order to conserve the weakening property of the calculus for the case $k = 0$.

The terms from $\mathsf{s}\ell\mathsf{T}$ that can be called are only first-order functions, so they take base type inputs and output a base type value.

3.4 Examples

We give some examples of terms in our enriched EAL calculus, first some terms we can usually see for the elementary affine logic, and then we give the term for computing tower of exponentials.

Some General Results and Notations on EAL. To begin with, we prove some formulas on the elementary affine logic that we will often use in examples without explicitly reminding them.

- We have a proof of the formula $!T \multimap !T \otimes !T$ given by the term $\lambda x.let\ !x' = x\ in\ !x' \otimes !x'$.
- The types $!(T \otimes T')$ and $!T \otimes !T'$ are equivalent, using the terms $\lambda c.let\ !c' = c\ in\ !(let\ x \otimes y = c'\ in\ x) \otimes !(let\ x \otimes y = c'\ in\ y)$ and $\lambda c.let\ x \otimes y = c\ in\ let\ !x' = x\ in\ let\ !y' = y\ in\ !(x' \otimes y')$. This proof is specific to affine logic since it relies on weakening.
- For base types A we have the coercion $A \multimap !A$. For example, for words, this is given by the term $coerc_w = iter^!_W(!(\lambda w'.s_0(w')), !(\lambda w'.s_1(w')), !\epsilon)$, with $coerc_w\ \underline{w} \to^* \ !\underline{w}$.
- We write $\lambda x \otimes y.M$ for the term $\lambda c.let\ x \otimes y = c\ in\ M$.

Polynomials and Tower of Exponentials in EAL. Recall that we defined polynomials in $\mathsf{s}\ell\mathsf{T}$. With this we can define polynomials in EAL with type $N \multimap N$ using the $\mathsf{s}\ell\mathsf{T}$ call. Moreover, using the iteration in EAL, we can define a tower of exponential.

We can compute the function $k \mapsto 2^{2^k}$ in EAL with type $N \multimap !N$:

$$\dfrac{\dfrac{\dfrac{\dfrac{n : N \mid \cdot \vdash n : N \quad x_1 : N^{a_1} \vdash_{\mathsf{s}\ell\mathsf{T}} mult\ x_1\ x_1 : N^{a_1 \cdot a_1}}{n : N \mid \cdot \vdash [\lambda x_1.mult\ x_1\ x_1](n) : N \multimap N}}{\cdot \mid \cdot \vdash \lambda n.[\lambda x_1.mult\ x_1\ x_1](n) : N \multimap N}}{\cdot \mid \cdot \vdash !(\lambda n.[\lambda x_1.mult\ x_1\ x_1](n)) : !(N \multimap N)} \qquad \cdot \mid \cdot \vdash !\underline{2} : !N}{\cdot \mid \cdot \vdash exp = iter^!_N(!\lambda n.[\lambda x_1.mult\ x_1\ x_1](n), !\underline{2}) : N \multimap !N}$$

And the reduction rules give us : $iter^!_N(!\lambda n.[\lambda x_1.mult\ x_1\ x_1](n), !\underline{2})\ \underline{k} \rightarrow^*$ $!((\lambda n.[\lambda x_1.mult\ x_1\ x_1](n))^k\ \underline{2}) \rightarrow^* !(2^{2^k})$.

For an example of measure, for the subproof $\pi \lhd \cdot \mid \cdot \vdash \lambda n.[\lambda x_1.mult\ x_1\ x_1](n) : N \multimap N$, we have $depth(\pi) = 1$ and as the weight for $\sigma \lhd x_1 : N^{a_1} \vdash_{\mathsf{s}\ell\mathsf{T}} mult\ x_1\ x_1 : N^{a_1 \cdot a_1}$ is $\omega(\sigma) = 4 + a_1 + 3a_1^3$, we can deduce

$\mu(\pi) = (1 + 1 + 1 \cdot (d(\omega(\sigma) + a_1 \cdot a_1) + 1), 1 + (\omega(\sigma) + a_1 \cdot a_1)[1/a_1]) = (2 + 3 + 1, 1 + 4 + 1 + 3 + 1) = (6, 10)$

If we define, $2^x_0 = x$ and $2^x_{k+1} = 2^{2^x_k}$, with the use of polynomials, we can represent the function $n \mapsto 2^{P(n)}_{2k}$ for all $k \geqslant 0$ and polynomial P with a term of type $N \multimap !^k N$.

Satisfiability of a Propositional Formula. We can give a term of type $N \otimes W \multimap !B$ such that, given a formula on conjunctive normal form encoded in the type $N \otimes W$, we can check the satisfiability of this formula. The modality in front of the output $!B$ shows that we used a non-polynomial computation, or more precisely an iteration in EAL, in order to define the term, as expected of a term for satisfiability.

We encode formula in conjunctive normal form in the type $N \otimes W$, representing the number of distinct variables in the formula and the encoding of the formula by a word on the alphabet $\Sigma = \{0, 1, \#, |\}$. A literal is represented by the number of the corresponding variable written in binary and the first bit determines if the literal is positive or negative (0 meaning negative). The # indicates the beginning of the representation of a variable, and the | indicates the beginning of the representation of a clause.

For example, the formula $(x_1 \vee x_0 \vee x_2) \wedge (x_3 \vee \overline{x_0} \vee \overline{x_1}) \wedge (\overline{x_2} \vee x_0 \vee \overline{x_3})$ is represented by $\underline{4} \otimes \underline{|\#11\#10\#110|\#111\#00\#01|\#010\#10\#011}$

Intermediate terms in $\mathsf{s}\ell\mathsf{T}$. We recall that we have already defined previously a term $Cbinarytounary$: $N^I \multimap W^J \multimap N^I$ that can be extended for the alphabet Σ. In the following, we may not always consider that the alphabet now contains # and | when we do not use them in the term. In this case, it means that the terms for those letters are not useful and so we could for example put the identity. We can also define a term $occ_a : W^I \multimap N^I$ that gives the number of occurrences of $a \in \Sigma$ in a word. We define a term that gives the n^{th} bit (from right) of a binary word as a boolean :

$n^{th} = \lambda w, n.ifw(\lambda w'.\mathsf{ff}, \lambda w'.\mathsf{tt}, \mathsf{ff})\ ((itern(pred, rev\ w))\ n) : W^I \multimap N^I \multimap B$

with $pred : W^I \multimap W^I = ifw(\lambda w.w, \lambda w.w, \epsilon)$.

We can define a term of type $W^I \multimap W^I \otimes W^I$ that separates a word $w = w_0 a w_1$ in $w_0 \otimes w_1$ such that w_1 does not contain any a. This function will allow us to extract the last clause/literal of a word representing a formula.

$Extract_a = \lambda w.let\ b' \otimes w'_0 \otimes w'_1 = ITERW(V_0, V_1, V_\#, V_|, V_\epsilon)\ w\ in\ w'_0 \otimes w'_1$

with $V_a = \lambda b \otimes w_0 \otimes w_1.if(\mathsf{tt} \otimes s_a(w_0) \otimes w_1, \mathsf{tt} \otimes w_0 \otimes w_1)\ b$

$\forall c \neq a, V_c = \lambda b \otimes w_0 \otimes w_1.if(\mathsf{tt} \otimes s_c(w_0) \otimes w_1, \mathsf{ff} \otimes w_0 \otimes s_c(w_1))\ b$

$V_\epsilon = ff \otimes \epsilon \otimes \epsilon$

For the intuition on this term, the boolean b' used in the iteration is a boolean that indicates if we have already read the letter "a" previously.

A valuation is represented by a binary word with a length equal to the number of variable, such that the n^{th} bit of the word represents the boolean associated to the n^{th} variable.

We define a term $ClausetoBool : N^I \multimap W^J \multimap W^K \multimap B$ such that, given the number of variable, a valuation and a word representing a clause, this term outputs the truth value of this clause using the valuation.

$ClausetoBool = \lambda n, w_v, w_c.\ let\ w \otimes b = itern(\lambda w' \otimes b'.let\ w_0 \otimes w_1 = Extract_\#\ w'\ in\ w_0 \otimes (or\ b'\ (LittoBool\ n\ w_v\ w_1)), w_c \otimes \mathsf{ff})\ (occ_\# w_c)\ in\ b$

With $LittoBool : N^I \multimap W^J \multimap W^K \multimap B$ converting a literal into the boolean given by the valuation.

$LittoBool = \lambda n, w_v, w_l.ifw$
$(\lambda w'.n^{th}\ w_v\ (Cbinarytounary\ n\ w'), \lambda w'.not\ (n^{th}\ w_v\ (Cbinarytounary\ n\ w')), \mathsf{ff})\ w_l$.

With this we can check if a clause is true given a certain valuation. We can define in the same way a term $FormulatoBool : N^I \multimap W^J \multimap W^K \multimap B$.

Testing all different valuations. Now all we have to do is to test this term on all possible valuations. If n is the number of variables, all possible valuations are described by all the binary integer from 0 to $2^n - 1$. Then we only need to use the iterator in $\mathsf{s\ell T}$ with base type-inputs in order to check if one valuation satisfy the formula. Formally, this is given by the term :

$$SAT = \lambda n \otimes w.let\ !r = iter^!_N(!(\lambda n_0 \otimes n_1.succ(n_0) \otimes [double](n_1)), !(\underline{0} \otimes \underline{1}))\ n\ in\ let\ !w_f = coerc\ w\ in\ !(let\ n \otimes exp = r\ in\ [\lambda n, exp, w_f.REC(\lambda val, b.or\ b\ (FormulatoBool\ n\ (Cunarytobinary\ n\ val)\ w_f), \mathsf{ff})\ exp](n, exp, w_f)).$$

The first line computes 2^n and also do the coercion of n. This technique is important as it shows that the linearity of EAL for base variables is not too constraining for the iteration. If you have to use it multiple time, you can just do a *simultaneous* iteration using the tensor type. In the last line the term is a big "or" on the term $FormulatoBool$ applied to different valuations. We recall that the constructor REC is the iterator with base type arguments that has been defined in the previous section on $\mathsf{s\ell T}$.

And with that we have $SAT : N \otimes W \multimap !B$.

Solving QBF_k. Now we consider the following problem, with k being a fixed non-negative integer : Suppose given a formula with the form

$Q_k x_n, x_{n-1}, \ldots, x_{i_{k-1}+1}.\ Q_{k-1} x_{i_{k-1}}, x_{i_{k-1}-1}, \ldots, x_{i_{k-2}+1}.Q_{k-2} \ldots, Q_1 x_{i_1}, x_{i_1-1}, \ldots x_0.\phi$

The formula ϕ is a propositional formula in conjunctive normal form on the variables from x_0 to x_n, and $Q_i \in \{\forall, \exists\}$ are alternating quantifiers. That means that if Q_1 is $\forall$ then Q_2 must be $\exists$ and then Q_3 must be $\forall$ and so on. Here the variables are ordered for simplification. It can always be done by renaming. And now we have to answer if this formula is true. This can be solved in our enriched EAL calculus.

First, let us talk about the encoding of such a formula. With those ordered variable, a representation of such a formula can be a term of type $N_k \otimes N_{k-1} \otimes \ldots \otimes N_1 \otimes B \otimes W$. For all i with $1 \leqslant i \leqslant k$, N_i represents the number of variables between the quantifiers Q_i and Q_{i-1}. The boolean represents the quantifier Q_k, with the convention $\forall = \mathsf{tt}$. And finally, the formula ϕ is encoded in a word as previously. This is not a canonic representation of a formula, but for any good encoding of a QBF_k formula we should be able to extract those informations with a $\mathsf{s\ell T}$ term, so for simplification, we directly take this encoding.

For example, with $k = 2$, the formula $\forall x_3, x_2.\exists x_1, x_0.(x_1 \vee x_0 \vee x_2) \wedge (x_3 \vee \overline{x_0} \vee \overline{x_1}) \wedge (\overline{x_2} \vee x_0 \vee \overline{x_3})$ is represented by $\underline{2} \otimes \underline{2} \otimes \mathsf{tt} \otimes \underline{|\#11\#10\#110|\#111\#00\#01|\#010\#10\#011}$.

Defining a $\mathsf{s\ell T}$ term for QBF_k. Now we define by induction on k a $\mathsf{s\ell T}$ term called qbf_k for k a non-negative integer. We give to this term a type :

$qbf_k : W^{K_1} \otimes N^{I_k} \otimes N^{J_k} \otimes \ldots \otimes N^{I_1} \otimes N^{J_1} \otimes B \otimes W^{K_2} \multimap B$.

One can see a similitude with the representation of a QBF_k formula. But we add some arguments. First, the argument w_v of type W^{K_1} is a valuation on free variables of the QBF_k formula. Then we are given for each quantifiers two integers n_i and exp_i of type N^{I_i} and N^{J_i}, with n_i being the number of variables between the quantifiers Q_i and Q_{i-1}, and $exp_i = 2^{n_i}$. Finally, the boolean represents the quantifier Q_k and W^{K_2} is a formula on variables from x_0 to $x_{n_1+\ldots+n_k+length(w_v)-1}$.

qbf_0 has already been defined. Indeed, we have $qbf_0 =$

$\lambda w_v \otimes q \otimes w_f.FormulatoBool\ (length\ w_v)\ w_v\ w_f : W^{K_1} \otimes B \otimes W^{K_2} \multimap B$.

Now, let us give the term for qbf_1. One can observe that it is close to the $\mathsf{s\ell T}$ term used for SAT. To begin with, we define a term $andor : B \multimap (B \multimap B) = if(and, or)$. We also write $conc : W^I \multimap W^J \multimap W^{I+J}$ the term for concatenation of words. With that we define :

$qbf_1 = \lambda w_v \otimes n_1 \otimes exp_1 \otimes q \otimes w_f.$
$REC(\lambda val, b.(andor\ q)\ b\ (qbf_0\ (conc\ w_v\ (Cunarytobinary\ n_1\ val)) \otimes (not\ q) \otimes w_f), q)\ exp_1$

So, contrary to SAT, we do not always do a big "or" on the results of qbf_0 but we do either a big "and" if the quantifier Q_k is $\forall$, either a big "or" if the quantifier is $\exists$. And when we call qbf_0, we have to update the current valuation w_v and we have to alternate the quantifier.

Now with this intuition, we give the general term for qbf_{k+1} :

$qbf_{k+1} = \lambda w_v \otimes exp_{k+1} \otimes n_{k+1} \otimes exp_k \otimes n_k \ldots \otimes exp_1 \otimes n_1 \otimes q \otimes w_f.$
$REC(\lambda val, b.(andor\ q)\ b\ (qbf_k\ (conc\ w_v\ (Cunarytobinary\ n_{k+1}\ val)) \otimes exp_k \otimes n_k \ldots \otimes exp_1 \otimes n_1 \otimes (not\ q) \otimes w_f, q)\ exp_{k+1}.$

And with this term, given a QBF_k formula in the enriched EAL calculus of type $N_k \otimes N_{k-1} \otimes \ldots \otimes N_1 \otimes B \otimes W$, we can define the following term :

$ealqbf_k = \lambda n_k \otimes \ldots \otimes n_1 \otimes q \otimes w.let\ !r = (!\epsilon) \otimes exp\ n_k \otimes coerc\ n_k \otimes \ldots \otimes exp\ n_1 \otimes coerc\ n_1 \otimes coerc\ q \otimes coerc\ w_f\ in\ !([qbf_k](r))$

With some abuse of notations since we consider the equivalence between $!(T \otimes T')$ and $(!T) \otimes (!T')$ and also between $T \otimes T' \multimap T''$ and $T \multimap T' \multimap T''$. Moreover, we duplicate here the variables n_i but as explained previously for SAT, one can compute the coercion and the exponential simultaneously without duplication. It is just better for the intuition written that way.

And so, we obtain a term solving QBF_k with type $N_k \otimes N_{k-1} \otimes \ldots \otimes N_1 \otimes B \otimes W \multimap !B$.

Solving the SUBSET SUM Problem. We give here another example of a NP-Complete problem. Given a goal integer $k \in \mathbb{N}$ and a set S of integers, is there a subset $S' \subset S$ such that $\sum_{n \in S'} n = k$?

We explain how we could solve this problem in our calculus. We represents the SUBSET SUM problem by two words, k written as a binary integer and a word of the form $|\underline{n_1}|\underline{n_2}| \ldots |\underline{n_m}$, with the integers written in binary, representing the set S. In order to solve this problem, we can first define a $\mathsf{s\ell T}$ term $equal : W^I \multimap W^J \multimap B$ that verifies if two binary integers are equal. Note that this is not exactly the equality on words because of the possible extra zeros at the beginning. Then, we can define a term $subsetsum : W^I \multimap W^J \multimap W^{I \cdot J}$ such that, given the word w_S representing the set S and a binary word w_{sub} with a length equal to the cardinality of S, this term computes the sum of all the elements of the subset represented by w_{sub}, since this word can be seen as a function from S to $\{0, 1\}$.

$$subsetsum = \lambda w_S.ITERW(\lambda w \otimes w_r.let\ w_0 \otimes w_1 = Extract_|\ w\ in\ w_0 \otimes w_r, \lambda w \otimes w_r.let\ w_0 \otimes w_1 = Extract_|\ w\ in\ w_0 \otimes (Binaryadd\ w_r\ w_1), w_S \otimes s_0(\epsilon))$$

We obtain a type $W^{I \cdot J}$ for the output because we iterate at most J times a function for binary addition which can be given a type $W^{a \cdot I} \multimap W^I \multimap W^{(a+1) \cdot I}$. Note that to define this function, we use $extract_|$ defined previously. Then, we can solve the SUBSET SUM problem in the same way as SAT with the term :

$SolvSubsetSum = \lambda k \otimes w_S.let\ !r = (exp\ [occ_|](w_S)) \otimes (coerc\ w_S) \otimes (coerc\ k)\ in$
$![\lambda n, w, k.REC(\lambda val, b.or\ b\ (equal\ k\ (subsetsum\ w\ (Cunarytobinary\ (occ_|\ w)\ val))), \mathsf{ff})\ n](r)$

With again some abuse of notation and non-linearity only to clarify things. And so, we obtain a term of type $W \otimes W \multimap !B$. We could also construct a term that gives us the subset corresponding to the goal, by changing the type in the iteration REC from $N^a \multimap B \multimap B$ to $N^a \multimap (B \otimes W^I) \multimap (B \otimes W^I)$, W^I being the type of the argument w.

3.5 Properties of the Type System and the Measure

We give some properties of the type system and the measure, and we give the intuition or the way to prove those properties. The final goal is to prove the main theorem that will follow this section and bound the number of reduction steps of a typed term.

Weakening and Monotonic Contexts. To begin with, we define a *!-free* path. In a derivation tree π, a !-free path is a path starting from the root that does not cross a !-rule (the rule introducing a $!M$ term). Those kind of paths are important for the substitution lemmas since the !-rule can erase contexts and remove discharged types. In the same way, we define a !-free branch as a !-free path that leads to a leaf.

Lemma 10 (Monotonic Δ). *Suppose $\pi \lhd \Gamma \mid \Delta \vdash M : T$, then for any sequent $\Gamma' \mid \Delta' \vdash M' : T'$ that appears in a !-free path of π, we have Δ included in Δ'.*

This is a direct proof by looking at the typing rules of the system. We can use this lemma when we know that the linear context has not been erased, or when the non-discharged types have not been erased.

Lemma 11 (Weakening). *If Γ and Γ' are disjoint linear variables contexts, Δ and Δ' are disjoint general variables contexts and $\pi \lhd \Gamma \mid \Delta \vdash M : T$ then we have a proof $\pi' \lhd \Gamma, \Gamma' \mid \Delta, \Delta' \vdash M : T$ with $\mu_n(\pi') = \mu_n(\pi)$ for all n.*

Substitution Lemmas.

Lemma 12 (Linear Substitution). *If $\pi \lhd \Gamma_1, x : T' \mid \Delta \vdash M : T$ and $\sigma \lhd \Gamma_2 \mid \Delta \vdash M' : T'$ then we have a proof $\pi' \lhd \Gamma_1, \Gamma_2 \mid \Delta \vdash M[M'/x] : T$. The proof π' is π in which we add Γ_2 in all branches where x appears, and we replace the occurrences of axiom rules $\Gamma', x : T' \mid \Delta' \vdash x : T'$ by the proof $\sigma' \lhd \Gamma', \Gamma_2 \mid \Delta' \vdash M' : T'$ given by the weakening lemma, since Δ is in Δ' by lemma 10. Moreover, for all n, $\mu_n(\pi') \leqslant \mu_n(\pi) + \mu_n(\sigma)$.*

This is proved by induction on π. The proof is rather direct since it comes from the fact that rules are multiplicative for Γ, and so x only appears in one of the premises for each rule. Moreover, in the !-rule, since $x : T'$ is erased, it means that x does not appear in the term M, thus the proof follows from the fact that $M[M'/x] = M$.

Lemma 13 (General substitution). *If $\pi \lhd \Gamma \mid \Delta, x : T' \vdash M : T$ and $\sigma \lhd \varnothing \mid \Delta \vdash M' : T'$ and the number of occurrences of x in M is less than K, then we have a proof $\pi' \lhd \Gamma \mid \Delta \vdash M[M'/x] : T$. The proof π' is π in which we replace the occurrences of axiom rules $\Gamma' \mid \Delta', x : T' \vdash x : T'$ by the proof $\sigma' \lhd \Gamma' \mid \Delta' \vdash M' : T'$ given by the weakening lemma, since Δ is in Δ' by lemma 10. Moreover, for all n, $\mu_n(\pi') \leqslant \mu_n(\pi) + K \cdot \mu_n(\sigma)$.*

This lemma is proved by induction on π. For rules with more than one premise, for example the *app* rule with 2 premises, we use the fact that if we can bound the number of occurrences of x in $M = M_0\ M_1$ by K, then we can find K_0, K_1 such that $K_0 + K_1 = K$ and K_i bounds the number of occurrences of x in M_i.

Lemma 14 (Discharged substitution lemma). *If $\pi \lhd \Gamma \mid \Delta', [\Delta], x : [T'] \vdash M : T$ and $\sigma \lhd \varnothing \mid \Delta \vdash M' : T'$ then we have a proof $\pi' \lhd \Gamma \mid \Delta', [\Delta] \vdash M[M'/x] : T$. The proof π' is π in which we replace the occurrences of axiom rules $\Gamma' \mid \Delta'', x : T' \vdash x : T'$ by the proof $\sigma' \lhd \Gamma' \mid \Delta'' \vdash M' : T'$ given by the weakening lemma. Moreover, for all n, $\mu_n(\pi') \leqslant (\omega_0(\pi), (\mu_n^1(\pi) + \omega_1(\pi) \cdot \mu_{n-1}(\sigma)))$.*

First we precise why we can use the weakening lemma to replace the axiom rules. By the lemma 10, in a branch that leads to an axiom rule for x, before crossing a !, we have a !-free path and so Δ grows. Then in order to apply the axiom rule, we need to cross a ! since x has a discharged type at the root of π. After crossing this !, we cannot cross again a ! in a branch that lead to an axiom rule for x since crossing a ! would erase x. So we can again apply the lemma 10 and so Δ is indeed included in Δ''.

Lemma 14 is proved by induction on π. All cases are direct except the ! rule, that can be found in the appendix.

3.6 Main Theorem : Defining a Measure to Bound the Number of Reductions

In this section, we show that we can bound the number of reduction steps of a typed term using the measure. This is done by showing that a reduction preserves some properties on the measure, and then give an explicit integer bound that will strictly decrease after a reduction. This proof uses the same logic as the one from [20]. The relation $\mathcal{R}$ defined in the following is a generalization of the usual requirements exposed in elementary linear logic in order to control reductions.

Definition of t_α and the Relation $\mathcal{R}$.

Definition of t_α. We define a family of tower functions $t_\alpha(x_1, \ldots, x_n)$ on vectors of integers by induction on n, where we assume $\alpha \geqslant 1$ and $x_i \geqslant 2$ for all i.

$t_\alpha() = 0 \qquad t_\alpha(x_1, \ldots, x_{n-1}, x_n) = (\alpha \cdot x_n)^{2^{t_\alpha(x_1, \ldots, x_{n-1})}}$ for $n \geqslant 1$

Definition of $\mathcal{R}$. We define a relation on vectors that we note $\mathcal{R}$. Intuitively, we want $\mathcal{R}(\mu, \mu')$ to express the fact that a proof of measure μ has been reduced to a proof of measure μ'.

Let $\mu, \mu' \in \mathbb{N}^{n+1}$. We have $\mathcal{R}(\mu, \mu')$ if and only if :

1. $\mu \geqslant \tilde{2}$ and $\mu' \geqslant \tilde{2}$.
2. $\mu' <_{lex} \mu$. And so we write $\mu = (\omega_0, \ldots, \omega_n)$ and $\mu' = (\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0}, \ldots, \omega'_n)$. With $\omega_{i_0} > \omega'_{i_0}$.
3. There exists $d \in \mathbb{N}, 1 \leqslant d \leqslant (\omega_{i_0} - \omega'_{i_0})$ such that $\forall j > i_0,\ \omega'_j \leqslant \omega_j \cdot (\omega_{i_0+1})^{d-1}$

The first condition with $\tilde{2}$, that can also be seen in the definition of t_α, makes calculation easier, since with this condition, exponentials and multiplications conserve the strict order between integers. This does not harm the proof, since we can simply add $\tilde{2}$ to each vector we will consider.

Some Properties on t_α and $\mathcal{R}$. We give some properties on t_α and $\mathcal{R}$. The proofs are usually calculation.

Lemma 15. *If $\mu \leqslant \mu'$ then $t_\alpha(\mu) \leqslant t_\alpha(\mu')$*

This is just a consequence of the fact that the exponentiation is monotonic.

Lemma 16 (Shift). *Let $k \in \mathbb{N}^*$. Let $\mu = (\omega_0, \ldots, k \cdot \omega_{i-1}, \omega_i, \ldots \omega_n)$ and $\mu' = (\omega_0, \ldots, \omega_{i-1}, k \cdot \omega_i, \ldots \omega_n)$. Then $t_\alpha(\mu') \leqslant t_\alpha(\mu)$.*

The proof of this lemma can be found in 6.6. This is just a calculation.

Lemma 17. *If $\tilde{2} \leqslant \mu' < \mu$ then $\mathcal{R}(\mu, \mu')$.*

Take $d = 1$ and the proof is direct.

Lemma 18. *If $\mathcal{R}(\mu, \mu')$ then for all μ_0, we have $\mathcal{R}(\mu + \mu_0, \mu' + \mu_0)$.*

Point 1 and 2 in the definition of $\mathcal{R}(\mu + \mu_0, \mu' + \mu_0)$ are given by the hypothesis $\mathcal{R}(\mu, \mu')$. We keep the notations $\omega_j, \omega'_j, i_0, d$.

$1 \leqslant d \leqslant \omega_{i_0} - \omega'_{i_0}$ so $1 \leqslant d \leqslant (\omega_{i_0} + \mu_0(i_0)) - (\omega'_{i_0} + \mu_0(i_0))$. Let $j > i_0$, we have :
$\omega'_j + \mu_0(j) \leqslant \omega_j \cdot (\omega_{i_0+1})^{d-1} + \mu_0(j) \leqslant (\omega_j + \mu_0(j)) \cdot (\omega_{i_0+1} + \mu_0(i_0+1))^{d-1}$ since $\omega_{i_0+1} \geqslant 1$.

This concludes the proof of lemma 18.

Finally, the theorem that connects those two definitions :

Theorem 4. *Let $\mu, \mu' \in \mathbb{N}^{n+1}$ and $\alpha \geqslant n$, $\alpha \geqslant 1$. If $\mathcal{R}(\mu, \mu')$ then $t_\alpha(\mu') < t_\alpha(\mu)$*

The proof of this theorem can be found in the appendix, section 6.6. Again, this is just a calculation using the previous lemmas. This theorem shows that if we want to ensure that a certain integer defined with t_α strictly decreases for a reduction, it is sufficient to work with the relation $\mathcal{R}$.

Reductions and Relations. We state the subject reduction of the calculus and we show that the measure allows us to construct a bound on the number of reductions.

Theorem 5. *Let $\tau \lhd \Gamma \mid \Delta \vdash M_0 : T$ and $M_0 \to M_1$. Let α be an integer equal or greater than the depth of τ. Then there is a proof $\tau' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$. Moreover, the depth of τ' is smaller than the depth of τ.*

The proof of this theorem can be found in the appendix, in section 6.2. This proof uses the substitution lemma for reduction in which substitution appears, and for the others constructors, one can see that the measure given in the type system for this calculus is following this idea of the relation $\mathcal{R}$. For example, in the reduction
$[\lambda x_n \dots x_1.t](M_1, \dots, M_{n-1}, \underline{v}) \to [\lambda x_{n-1} \dots x_1.t[\underline{v}/x_n]](M_1, \dots, M_{n-1})$, the degree that appears at position 0 is here to compensate the growing of the measure at position 1. Indeed, at position 1, in the type system, we consider that all variables are sent to the integer 0. However, in this reduction rule, the variable x_n is sent to an actual integer generally greater than 0. Thus, the weight in $\mathsf{s}\ell\mathsf{T}$ of the term t increases, and so the total measure μ increases at position 1. But as expressed by the relation $\mathcal{R}$, we only need to control this growing. This is in fact what we show in the proof.

Now using the previous results, we can easily conclude our bound on the number of reductions.

Theorem 6. *Let $\pi \lhd \Gamma \mid \Delta \vdash M : T$. Denote $\alpha = max(depth(\pi), 1)$, then $t_\alpha(\mu_\alpha(\pi) + \tilde{2})$ is a bound on the number of reductions from M.*

For a proof π', we define an integer $t_\alpha(\mu_\alpha(\pi') + \tilde{2})$. If we prove that this integer strictly decreases for a reduction, we obtain theorem 6.

By the theorem 5, for $M \to^* M' \to M''$, α is an upper bound on the depth of the typing derivation of M'. We note τ the typing of M' and τ' the typing of M''. We have $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$ again by theorem 5. And finally, by the theorem 4, we have $t_\alpha(\mu_\alpha(\tau') + \tilde{2}) < t_\alpha(\mu_\alpha(\tau) + \tilde{2})$. This concludes the proof.

4 Complexity Results : Characterization of $2k - EXP$

Now that we have proved the precedent theorem, we have obtained a bound on the number of reduction steps from a term. More precisely, this bound shows that between two consecutive weights ω_{i+1} and ω_i, there is a difference of 2 in the height of the tower of exponentials. This will allow us to give a characterization of the classes $2k - EXP$ for $k \geqslant 0$, and each modality "!" in the type of a term will induce a difference of 2 in the height of the tower of exponential.

4.1 Restricted Reductions and Values

First, we show that the precedent bound on the number of reductions in Theorem 6 is imprecise. Indeed, if we restrict the possible reductions, we obtain a more precise bound.

Restricted Reductions : Reductions up to a Certain Depth. For $i \in \mathbb{N}$, we define the i-reductions, that we note $\to_i$:

- $\forall i \geqslant 1, [t]() \to_i [t']()$ if $t \to t'$ in $\mathsf{s}\ell\mathsf{T}$.
- $\forall i \geqslant 1, [\underline{v}]() \to_i \underline{v}$
- For the other base reductions $M \to M'$, we have $\forall i \in \mathbb{N}, M \to_i M'$
- For all $i \in \mathbb{N}$, if $M \to_i M'$ then $!M \to_{i+1} !M'$
- For all others constructors, the index i stays the same. For example for the application, we have for all $i \in \mathbb{N}$, if $M \to_i M'$ then $M\ N \to_i M'\ N$.

Now, we can find a more precise measure to bound the number of i-reductions. The proof is very similar to the proof of theorem 5 and 6, and one can find in section 6.7 some remarks explicitly describing the important points for lemma 19.

Lemma 19. *Let $i \in \mathbb{N}$, $\tau \lhd \Gamma \mid \Delta \vdash M_0 : T$ and $M_0 \to_i M_1$. Then there is a proof $\tau' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}(\mu_i(\tau) + \tilde{2}, \mu_i(\tau') + \tilde{2})$*

From that, we can conclude the following theorem :

Theorem 7. *Let $\pi \lhd \Gamma \mid \Delta \vdash M : T$ and $\alpha = max(i, 1)$. Then $t_\alpha(\mu_i(\pi) + \tilde{2})$ is a bound on the number of i-reductions from M.*

Values Associated to Restricted Reductions. We can now give the form of closed normal terms for i-reductions. For that, we define for all $i \in \mathbb{N}$, closed i-values V^i by the following grammar :

$V^0 := M$

$\forall i \geqslant 1, V^i := \lambda x.M \mid !V^{i-1} \mid V_0^i \otimes V_1^i \mid \mathsf{zero} \mid succ(V^i) \mid ifn(V_0^i, V_1^i) \mid iter_N^!(V_0^i, V_1^i)$
$\mid \mathsf{tt} \mid \mathsf{ff} \mid if(V_0^i, V_1^i) \mid \epsilon \mid s_i(V^i) \mid ifw(V_0^i, V_1^i, V_2^i) \mid iter_W^!(V_0^i, V_1^i, V_2^i)$.

And now we can prove the following lemma

Lemma 20. *Let M be a term. If M is closed and has a typing derivation then, for all $i \in \mathbb{N}$, if M is normal for i-reductions then M is a i-value V^i.*

We prove this by induction on the term M. Some cases can be found in the appendix, section 6.8.

From the previous results, we now have that, from a typed term M, we can reach the normal form for i-reductions for M in less than $t_i(\mu_i(\pi) + \tilde{2})$ reductions, and this form is a i-value.

4.2 A Characterization of 2k-EXP

Now, we show that the type $!W \multimap !^{k+1}B$ characterizes the class $2k - EXP$ for $k \geqslant 0$.
We recall that 2_k^x is defined by $2_0^x = x$ and $2_{k+1}^x = 2^{2_k^x}$. The class $k - EXP$ is the class of problem solvable by a Turing machine that works in time $2_k^{p(n)}$ on an entry of size n.

First we show that the number of reductions for such a term is bounded by a tower of exponentials of height $2k$.

Lemma 21. *Let $\pi \lhd \cdot \mid \cdot \vdash t :!W \multimap !^{k+1}B$. Let w be a word of size $|w|$. We can compute the result of $t\ !\underline{w}$ in less than a $2k$-exponential tower in the size of w.*

Observe that the result of this computation is of type $!^{k+1}B$, and a $(k+2)$-value of type $!^{k+1}B$ is exactly of the form $!^{k+1}\mathsf{tt}$ or $!^{k+1}\mathsf{ff}$. So it is enough to only consider $(k+2)$-reductions to compute the result, by lemma 20.

The measure μ_n of $t\ !\underline{w}$ is $\mu_n = \mu_n(\pi) + 2 \cdot \mathbb{1}_0 + |\underline{w}| \cdot \mathbb{1}_2$. By theorem 7, we can bound the number of reductions from $t\ !\underline{w}$ by $t_{k+2}(\mu_{k+2} + \tilde{2})$.

By definition, in $t_{k+2}(\mu_{k+2} + \tilde{2})$, we can see that the weight at position 2, where the size of w appears, is at height $2k$. This concludes the proof of lemma 21.
Now we have to prove that we can simulate a Turing-machine in our calculus. This proof is usual in implicit complexity [6, 3]. The first thing we prove is the existence of a term in $\mathsf{s}\ell\mathsf{T}$ to simulate n steps of a deterministic Turing-machine on a word w. We give here the intuition of the encoding, and a more detailed explanation on how to work with this encoding can be found in the appendix, section 6.9.

Suppose given two variables $w : W^{a_w}$ and $n : N^{a_n}$, we note $Conf_b$ the type $W^{a_w+b} \otimes B \otimes W^{a_w+b} \otimes B^q$, with q an integer and B^q being q tensors of booleans. This type represents a configuration on a Turing machine after b steps, with B^q coding the state, and then $\underline{w_0} \otimes b \otimes \underline{w_1}$ represents the tape, with b being the position of the head, w_0 represents the reverse of the word before b, and w_1 represents the word after b. We can then define multiple term in $\mathsf{s}\ell\mathsf{T}$ with this encoding. First we have a term $init$ such that $w : W^{a_w}, n : N^{a_n} \vdash init : Conf_1$ and $init$ computes the initial configuration of the Turing machine. Then, we have a term $step$ with $\cdot \vdash step : Conf_b \multimap Conf_b + 1$ that computes the result of the transition function from a configuration to the next one, and finally we have a term $final$ with $\cdot \vdash final : Conf_b \multimap B$ verifying if the final configuration is accepted or not.

Now that we have that, if we can compute an integer n bounding the number of steps of a Turing-machine on an entry w, then we can effectively simulate the Turing-machine in our calculus using a $\mathsf{s}\ell\mathsf{T}$ call. As explained in the example in the previous section on enriched EAL, we can compute tower of exponentials if we have enough modalities !. This shows that, by using a ! modality, we can increase the integer n we can compute and thus increase the working time of the Turing-machine we want to simulate. A more detailed explanation on how to simulate a Turing machine is given in the appendix, section 6.9. With this, using the lemma 21, we have the following theorem

Theorem 8. *Terms of type $!W \multimap !^{k+1}B$ characterize the class $2k$-EXP.*

This theorem can be expanded for the classes $2k$-FEXP, that is the class of function from words to words that can be computed by a one-tape Turing machine running with a time at most $2_{2k}^{P(|w|)}$ on a word w. For a more precise definitions of such classes, see [6]. This characterization uses the same proof by replacing $!W \multimap !^{k+1}B$ by $!W \multimap !^{k+1}W$.

We could also characterize the class $2k$-EXP, in the type $W \multimap !^{k}B$. Indeed, we used here the type $!W \multimap !^{k+1}B$ because it is easier to work with a duplicable variable in input, but in fact, for the case $k = 0$ as we can do everything in $\mathsf{s\ell T}$, the input is already duplicable. And then, for $k \neq 0$, we showed previously in the examples for SAT or SUBSET SUM that even if the input is not duplicable, you can do simultaneous iteration in ELL and it works as if the variable was duplicable. We think that in our calculus, the base type variables could always be duplicable, not only in $\mathsf{s\ell T}$, and this should not harm our complexity bound. However, in order to simplify the proof, we did not consider this in this paper.

In elementary linear logic, we can characterize k-EXP with the type $!W \multimap !^{k+1}B$. The difference between the two calculus can be explained by the fact that in classical ELL, in the type $N \multimap N$ we only have polynomials of degree 1 (polynomials in general have the type $!N \multimap !N$), whereas in our case, polynomials have the type $N \multimap N$.

5 Conclusion

We showed a way to enrich a calculus from ELL with data structures and iterations controlled by $\mathsf{s\ell T}$, and we showed that this new calculus still has the elementary time reduction procedure. Moreover, with precise types we can characterize more precise complexity classes. This proof relies on the definition of a measure and restricted reductions. It differs from the standard proofs one can find for precise correction proofs in calculus inspired by linear logic. Some classical methods for this kind of proof rely on proof-nets with particular reduction strategies, or if the model is a calculus, one can show how the variables are duplicated, and again explicit a specific reduction strategy. For example, some work had already been done in order to enrich light linear logic with a more natural way of programming [7]. The proof of correction relies on a very specific strategy reduction, and as a consequence, it is hard to add a new enrichment in the calculus, since this proof is hard to adapt. This proof is pretty robust, it does not rely on a particular reduction strategy and thus an improvement of the calculus could be done without too much difficulty. Indeed, if we want to add more possibilities in the enriched calculus we defined, by verifying that the measure is still defined in such a way that reductions satisfy the relation $\mathcal{R}$, and by adapting i-reductions and i-values, then we can show that this additional possibility does not break the proof of correction. An example of enrichment would be a way to duplicate base variable for the EAL-calculus by allowing the use of first-order functions in $\mathsf{s\ell T}$ that output a tensor of base values. Another possibility would be to add list in our base values.

For further works, we could be interested in working with smaller complexity classes, and for example try to characterize the polynomial class with a type $!W \multimap !!B$. For that, we could wonder what happens when the computational power of the calculus used to enrich EAL decreases. For example, we could replace indexes in $\mathsf{s\ell T}$ describing polynomials by an other class of indexes, such as indexes describing affine functions. Another approach would be to replace $\mathsf{s\ell T}$ by the calculus for non-size increasing function [17]. With the polynomial bound on this calculus [1] we could adapt our correction proof to this calculus, and we can then wonder what is the expressiveness of this calculus.

There are also some questions left open for the enriched EAL-calculus, we could for example work on type inference, using previous works on linear dependent types [11],[4], and on ELL [8]. Maybe it is also possible to increase the power of $\mathsf{s\ell T}$ in our calculus and not limit its use to first order-function. Furthermore, we could also try to add multithreading and side-effects in this calculus following the proof given in [20].

6 Appendix

Type system for words in sℓT $\pi \lhd \dfrac{}{\Gamma \vdash \epsilon : W^I}\ \omega(\pi) = 0$

$$\pi \lhd \frac{\sigma \lhd \Gamma \vdash t : W^J \qquad J + 1 \leqslant I}{\Gamma \vdash s_i(t) : W^I}\ \omega(\pi) = \omega(\sigma)$$

$$\pi \lhd \frac{\begin{array}{l}\sigma_1 \lhd \Gamma_1, d\Gamma \vdash t_1 : W^I \multimap D \\ \sigma_0 \lhd \Gamma_0, d\Gamma \vdash t_0 : W^I \multimap D \qquad \sigma \lhd \Gamma, d\Gamma \vdash t' : D\end{array}}{\Gamma_0, \Gamma_1, \Gamma, d\Gamma \vdash ifw(t_0, t_1, t') : W^I \multimap D}\ \omega(\pi) = \omega(\sigma_1) + \omega(\sigma_0) + \omega(\sigma) + 1$$

$$\pi \lhd \frac{\begin{array}{l}D \sqsubset E \qquad E \sqsubset E[a+1/a] \\ \sigma_1 \lhd d\Gamma \vdash V_1 : D \multimap D[a+1/a] \qquad E[I/a] \sqsubset F \\ \sigma_0 \lhd d\Gamma \vdash V_0 : D \multimap D[a+1/a] \qquad \sigma \lhd \Gamma, d\Gamma \vdash t : D[1/a]\end{array}}{\Gamma, d\Gamma \vdash iterw(V_0, V_1, t) : N^I \multimap F}\ \omega(\pi) = I + \omega(\sigma) + I \cdot (\omega(\sigma_1) + \omega(\sigma_0))[I/a]$$

A non multiplicative rule for if In sℓT , the rule for if is multiplicative, this is not intuitive since in a computation, only one of the two term in the if is important, the other one will be erased. And so, for example, the term $if(x, x)$ should be considered linear in x. There is a way to avoid this problem, we give the method with an example. Take D a non base type and f a variable of type D. Take t and t' terms such that t contains at most one occurrence of the variable f and t' contains at most one occurrence of the variable f, suppose given two proofs $\sigma \lhd f : D \vdash t : E$ and $\sigma' \lhd f : D \vdash t' : E$. We cannot have the proof $f : D \vdash if(t, t') : B \multimap D$, however, we have the following proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\sigma \lhd f' : D \vdash t[f'/f] : E}{\cdot \vdash \lambda f'.t[f'/f] : D \multimap E} \qquad \dfrac{\sigma' \lhd f' : D \vdash t'[f'/f] : E}{\cdot \vdash \lambda f'.t'[f'/f] : D \multimap E}}{\cdot \vdash if(\lambda f'.t[f'/f], \lambda f'.t'[f'/f]) : B \multimap D \multimap E} \qquad b : B \vdash b : B}{b : B \vdash (if(\lambda f'.t[f'/f], \lambda f'.t'[f'/f]))\ b : D \multimap E} \qquad f : D \vdash f : D}{f : D, b : B \vdash (if(\lambda f'.t[f'/f], \lambda f'.t'[f'/f]))\ b\ f : E}}{f : D \vdash \lambda b.(if(\lambda f'.t[f'/f], \lambda f'.t'[f'/f]))\ b\ f : B \multimap E}$$

So if we forget about the renaming of variable, we could define $IF(t, t') = \lambda b.if(\lambda f.t, \lambda f.t')\ b\ f$.

States A state is a tensor of boolean for which we can have a match case. More precisely, for $n \in \mathbb{N}^*$, we define by induction the type $B^n = B \otimes B^{n-1}$ with $B^1 = B$. B^n describes states of size n. In the following, we will ignore the term for the associativity of the tensor. In order to precise the decomposition, we will note $let\ x_D \otimes y_{D'} = t\ in\ t'$ to explicit the decomposition when it is ambiguous.

There are 2^n base states of size n, given by the 2^n possibilities of associating n times $\mathtt{tt}$ or $\mathtt{ff}$. Moreover, there is a constructor to do a match-case on those states, $case_n(t_{0^n}, \ldots, t_{1^n})$. We will consider in order to simplify the notations that those indexes are the integers from 0 to $2^n - 1$ written in binary, with 1 referring to $\mathtt{tt}$. We define it by induction, and give the typing.

For $n = 1$, $case_1(t_0, t_1) = if(t_1, t_0)$ and for $n \geqslant 0$:

$$case_{n+1}(t_{0^{n+1}}, \ldots, t_{1^{n+1}}) = \lambda s.let\ s'_{B^n} \otimes x_B = s\ in\ case_n(t'_{0^n}, \ldots, t'_{1^n})\ s'$$

with, for all boolean word i, $t'_i = if(t_{i1}, t_{i0})\ x$.

With this definition, by noting $i = b_1 \cdots b_n$ the state and the boolean word, we have $case_n(t_{0^n}, \dots, t_{1^n})\ (b_1 \cdots b_n) \to^* case_{n-1}(t_{0^{n-1}b_n}, \dots t_{1^{n-1}b_n})\ (b_1 \cdots b_{n-1}) \to^* t_i$

Moreover, we can deduce this rule :

$$\frac{\forall i, 0 \leqslant i \leqslant 2^n - 1, \Gamma_i, d\Gamma \vdash t_i : D}{\Gamma_0, \dots, \Gamma_{2^n-1}, d\Gamma \vdash case_n(t_0, \dots, t_{2^n-1}) : B^n \multimap D}$$

6.1 Intermediate lemmas in sℓT

Proof of theorem 1 We prove theorem 1 saying that a closed and typed term is normal if and only if it is a value.

First, we prove by induction on values V that if V is closed and has a typing derivation then V is normal. We treat only some cases and the others are easily deducible from those cases.

- If $V = \lambda x.t$ then V is normal since in the definition of contexts for reductions, we cannot reduce under a λ-abstraction.
- If $V = V_0 \otimes V_1$. V is closed so are V_0 and V_1. Moreover V has a typing derivation, so it must finish with the introduction of tensor rule, and we deduce that V_0 and V_1 have also a typing derivation. So by induction hypothesis, V_0 and V_1 are normal. Then V has no base reduction possible, and no contexts reductions since V_0 and V_1 are normals, so V is normal.
- If $V = \mathsf{zero}$ then V is normal

Now for the other implication,, we prove that if a closed typed term is normal then it is a value. We prove that by induction on terms, again we only detail some interesting cases.

- If $t = t_0\ t_1$. Suppose, by absurd, that t is a closed typed normal term. Since t has a typing derivation, we know that t_0 and t_1 are also closed typed terms. By definition of contexts in which we can apply reductions, t_0 is normal, and so by induction hypothesis, t_0 is a value. Again, by definition of contexts, t_1 is normal, and so by induction hypothesis, t_1 is a value. So t_0 is a value with an arrow type $D \multimap D'$. By looking at the definition of values, either t_0 is a λ-abstraction, either it is one of the functional constructor like ifn. If t_0 is a λ-abstraction, as t_1 is a value, we could apply the usual β-rule, so this is not possible because t is normal. If t_0 is $ifn(V, V')$, as t_1 is a value of type N, it is the encoding of an integer, and so t is not normal since we could apply one of the ifn rule. All the other cases works in the same way, and we deduce that t cannot be normal.
- If $t = let\ x \otimes y = t_0\ in\ t_1$. Suppose that t is a closed typed normal term. Since t has a typing derivation, we know that t_0 has also a typing derivation, and t_0 is closed. By definition of contexts, t_0 is normal and so by induction hypothesis, t_0 is a value. t_0 has a tensor type $D \otimes D'$, and t_0 is a value, by definition of values, t_0 is of the form $V \otimes V'$, this is absurd since in this case t would not be normal. And so, we deduce that t cannot be a normal term.

Proof of lemma 3 We recall point 3 and point 4 of this lemma, and we show how to do the *itern* case for those points.

Let I be an index.

3) If $\pi \lhd \Gamma \vdash t : D$ then $\pi[I/a] \lhd \Gamma[I/a] \vdash t : D[I/a]$

4) $\omega(\pi[I/a]) = \omega(\pi)[I/a]$

Suppose that we have the following proof :

$$\pi \lhd \frac{\dfrac{D \sqsubset E \qquad E \sqsubset E[b+1/b]}{\sigma_1 \lhd d\Gamma \vdash V : D \multimap D[b+1/b]} \qquad \dfrac{E[J/b] \sqsubset F}{\sigma_2 \lhd \Gamma, d\Gamma \vdash t : D[1/b]}}{\Gamma, d\Gamma \vdash itern(V,t) : N^J \multimap F} \quad \omega(\pi) = J + \omega(\sigma_2) + J \cdot \omega(\sigma_1)[J/b]$$

We want to prove that $\pi[I/a] \lhd \Gamma[I/a] \vdash iter n(V, t) : N^{J[I/a]} \multimap F[I/a]$
By induction hypothesis and point 2 of lemma 3 we have

$$\begin{array}{ccc} D[I/a] \sqsubset E[I/a] & E[I/a] \sqsubset E[b+1/b][I/a] & E[J/b][I/a] \sqsubset F[I/a] \\ \multicolumn{2}{c}{\sigma_1[I/a] \lhd d\Gamma[I/a] \vdash V : D[I/a] \multimap D[b+1/b][I/a]} & \sigma_2[I/a] \lhd \Gamma[I/a], d\Gamma[I/a] \vdash t : D[1/b][I/a] \end{array}$$

By using the fact that b must be a fresh variable in Γ, $d\Gamma$, J and F, we can suppose, by renaming, that b is not free in I. Then, by lemma 2, we obtain a proof :

$$\pi[I/a] \lhd \frac{\begin{array}{ccc} D[I/a] \sqsubset E[I/a] & E[I/a] \sqsubset E[I/a][b+1/b] & E[I/a][J[I/a]/b] \sqsubset F[I/a] \\ \multicolumn{2}{c}{\sigma_1[I/a] \lhd d\Gamma[I/a] \vdash V : D[I/a] \multimap D[I/a][b+1/b]} & \sigma_2[I/a] \lhd \Gamma[I/a], d\Gamma[I/a] \vdash t : D[I/a][1/b] \end{array}}{\Gamma[I/a], d\Gamma[I/a] \vdash iter n(V, t) : N^{J[I/a]} \multimap F[I/a]}$$

With weight $\omega(\pi[I/a]) = J[I/a] + \omega(\sigma_2)[I/a] + J[I/a] \cdot \omega(\sigma_1)[I/a][J[I/a]/b]$
And so again by lemma 2, $\omega(\pi[I/a]) = \omega(\pi)[I/a]$.

Proof of theorem 3 We want to prove that for two indexes I and J, $I \leqslant J$ implies $d(I) \leqslant d(J)$. First, we prove the following lemma :

Lemma 22. *Let I be an index with at most one free variable x, then $x^{d(I)} \leqslant I \leqslant I[1/x] \cdot x^{d(I)}$.*

This is proved directly by induction on indexes, and it uses the fact that the constant integers in indexes are non-zero, the image of a variable in a valuation is non-zero and an index is always positive.

Now, we prove our theorem by contraposition. Given I, J such that $d(I) > d(J)$, we construct two new indexes called I' and J' that are I and J in which we replaced all variables by a new fresh variable x. The degree stays the same, and we have, by the lemma 22 :

$$x^{d(J)+1} \leqslant x^{d(I)} \leqslant I' \text{ and } J' \leqslant x^{d(J)} \cdot J'[1/x].$$

If we replace x by $k = (J'[1/x] + 1)$ (which is a non-zero integer), we obtain

$$I'[k/x] \geqslant k^{d(J)+1} \text{ and } J'[k/x] \leqslant k^{d(J)} \cdot (k-1).$$

And so we have $I'[k/x] > J'[k/x]$. We deduce that we have a valuation ϕ that send all free variables of I and J to k such that $I_\phi > J_\phi$, so we do not have $I \leqslant J$. By contraposition, we obtain the point 2 of the theorem 3.

6.2 Main theorem in sℓT

We want to prove theorem 2, first let us recall the statement of this theorem :
Let $\tau \lhd \Gamma \vdash t_0 : D$, and $t_0 \to t_1$, then there is a proof $\tau' \lhd \Gamma \vdash t_1 : D$ such that $\omega(\tau') < \omega(\tau)$.

We first consider the base-reduction case. Some cases are trivial and we will not develop them, indeed the if-rules can be proved only by using the weakening lemma.

- If $t_0 = (\lambda x.t)V$, and $t_1 = t[V/x]$ such that V is a base-typed term. We have a proof :

$$\tau \lhd \frac{\dfrac{\pi \lhd \Gamma_1, d\Gamma, x : U \vdash t : D}{\Gamma_1, d\Gamma \vdash \lambda x.t : U \multimap D} \qquad \sigma \lhd \Gamma_2, d\Gamma \vdash V : U}{\Gamma_1, \Gamma_2, d\Gamma \vdash (\lambda x.t)V : D}$$

with $\omega(\tau) = \omega(\sigma) + 1 + \omega(\pi)$.
Then by lemma 5, we have a proof $\sigma' \lhd d\Gamma \vdash V : U$ with $\omega(\sigma) = \omega(\sigma')$. Then by using the base value substitution lemma with π and σ', we obtain a proof $\pi' \lhd \Gamma_1, d\Gamma \vdash t[V/x] : D$ with $\omega(\pi') \leqslant \omega(\pi)$. Finally, by using the weakening lemma, we obtain a proof
$\tau' \lhd \Gamma_1, \Gamma_2, d\Gamma \vdash t[V/x] : D$ with $\omega(\tau') = \omega(\pi') \leqslant \omega(\pi) < \omega(\tau)$.

– If $t_0 = (\lambda x.t)V$, and $t_1 = t[V/x]$ such that V is a non-base-typed term. We have a proof :

$$\tau \lhd \dfrac{\dfrac{\pi \lhd \Gamma_1, d\Gamma, x : D' \vdash t : D}{\Gamma_1, d\Gamma \vdash \lambda x.t : D' \multimap D} \qquad \sigma \lhd \Gamma_2, d\Gamma \vdash V : D'}{\Gamma_1, \Gamma_2, d\Gamma \vdash (\lambda x.t)V : D}$$

With $\omega(\tau) = \omega(\sigma) + 1 + \omega(\pi)$.
Then by using the non-base value substitution lemma with π and σ, we obtain a proof $\pi' \lhd \Gamma_1, \Gamma_2, d\Gamma, \vdash t[V/x] : D$ with $\omega(\pi') \leqslant \omega(\pi) + \omega(\sigma)$. And so we have a proof $\tau' = \pi' \lhd \Gamma_1, \Gamma_2, d\Gamma \vdash t[V/x] : D$ with $\omega(\tau') = \omega(\pi') \leqslant \omega(\pi) + \omega(\sigma) < \omega(\tau)$.

– If $t_0 = let\ x \otimes y = V_0 \otimes V_1\ in\ t$ and $t_1 = t[V_0/x][V_1/y]$. We have a proof :

$$\tau \lhd \dfrac{\dfrac{\sigma_0 \lhd \Gamma_0, d\Gamma \vdash V_0 : D_0 \qquad \sigma_1 \lhd \Gamma_1, d\Gamma \vdash V_1 : D_1}{\Gamma_0, \Gamma_1, d\Gamma \vdash V_0 \otimes V_1 : D_0 \otimes D_1} \qquad \pi \lhd \Gamma, x : D_0, y : D_1, d\Gamma \vdash t : D}{\Gamma, \Gamma_0, \Gamma_1, d\Gamma \vdash let\ x \otimes y = V_0 \otimes V_1\ in\ t : D}$$

With $\omega(\tau) = \omega(\pi) + 1 + \omega(\sigma_0) + \omega(\sigma_1)$.
By considering two times the substitution lemmas, either in the base type case or the non-base type case, we obtain a proof $\tau' \lhd \Gamma, \Gamma_1, \Gamma_2, d\Gamma \vdash t[V_0/x][V_1/y]$ such that
$\omega(\tau') \leqslant \omega(\pi) + \omega(\sigma_0) + \omega(\sigma_1) < \omega(\tau)$. We do not detail explicitly how to use the substitution lemmas since it is the same as the previous cases.

– If $t = itern(V, V')\mathsf{zero}$ and $t_1 = V'$. We have a proof :

$$\tau \lhd \dfrac{\dfrac{\begin{array}{c} D \sqsubset E \qquad E \sqsubset E[a+1/a] \qquad E[I/a] \sqsubset F \\ \sigma_1 \lhd d\Gamma \vdash V : D \multimap D[a+1/a] \qquad \sigma_2 \lhd \Gamma, d\Gamma \vdash V' : D[1/a] \end{array}}{\Gamma, d\Gamma \vdash itern(V, V') : N^I \multimap F} \qquad \dfrac{}{\Gamma', d\Gamma \vdash \mathsf{zero} : N^I}}{\Gamma, \Gamma', d\Gamma \vdash itern(V, V')\mathsf{zero} : F}$$

With $\omega(\tau) = I + \omega(\sigma_2) + I \cdot \omega(\sigma_1)[I/a] \geqslant 1 + \omega(\sigma_2)$
We have $D[1/a] \sqsubset E[1/a] \sqsubset E[I/a] \sqsubset F$ by the index substitution lemma, and the monotonic index substitution lemma since $1 \leqslant I$. And so, by the subtyping and weakening lemmas, we have a proof $\tau' = \sigma_2' \lhd \Gamma, \Gamma', d\Gamma \vdash V' : F$ with $\omega(\tau') \leqslant \omega(\sigma_2) < \omega(\tau)$
The proof for the rule $iterw$ with ϵ follows the same logic.

– If $t = itern(V, V')\ succ(W)$ and $t_1 = itern(V, V\ V')W$. We have a proof :

$$\tau \lhd \dfrac{\dfrac{\begin{array}{c} D \sqsubset E \qquad E \sqsubset E[a+1/a] \qquad E[I/a] \sqsubset F \\ \sigma_1 \lhd d\Gamma \vdash V : D \multimap D[a+1/a] \qquad \sigma_2 \lhd \Gamma, d\Gamma \vdash V' : D[1/a] \end{array}}{\Gamma, d\Gamma \vdash itern(V, V') : N^I \multimap F} \qquad \dfrac{\pi \lhd \Gamma', d\Gamma \vdash W : N^J \qquad J + 1 \leqslant I}{\Gamma', d\Gamma \vdash succ(W) : N^I}}{\Gamma, \Gamma', d\Gamma \vdash itern(V, V')\ succ(W) : F}$$

With $\omega(\tau) = \omega(\pi) + I + \omega(\sigma_2) + I \cdot \omega(\sigma_1)[I/a]$
We can construct a proof for t_1 :

$$\tau' \lhd \dfrac{\dfrac{\begin{array}{c} \dfrac{\sigma_1[1/a] \lhd d\Gamma \vdash V : D[1/a] \multimap D[a+1/a][1/a] \qquad \sigma_2 \lhd \Gamma, d\Gamma \vdash V' : D[1/a]}{\Gamma, d\Gamma \vdash V\ V' : D[a+1/a][1/a]} \\ D[a+1/a] \sqsubset E[a+1/a] \\ E[a+1/a] \sqsubset E[a+1/a][a+1/a] \\ E[a+1/a][J/a] = E[J+1/a] \sqsubset E[I/a] \sqsubset F \\ \sigma_1[a+1/a] \lhd d\Gamma \vdash V : D[a+1/a] \multimap D[a+1/a][a+1/a] \end{array}}{\Gamma, d\Gamma \vdash itern(V, V\ V') : N^J \multimap F} \qquad \pi \lhd \Gamma', d\Gamma \vdash W : N^J}{\Gamma, \Gamma', d\Gamma \vdash itern(V, V\ V')W : F}$$

With $\omega(\tau') = \omega(\pi) + J + \omega(\sigma_2) + \omega(\sigma_1)[1/a] + J \cdot \omega(\sigma_1)[a+1/a][J/a]$
And we have $\omega(\tau') \leqslant \omega(\pi) + J + \omega(\sigma_2) + (J+1) \cdot \omega(\sigma_1)[J+1/a]$ so, since $J + 1 \leqslant I$, we have $\omega(\tau') < \omega(\pi) + I + \omega(\sigma_2) + I \cdot \omega(\sigma_1)[I/a] = \omega(\tau)$
The rules for $iterw$ in the cases s_0 and s_1 are follow the same logic.

Now we need to verify that a reduction under context strictly decreases the weight. This can be proved directly by structural induction on contexts.

6.3 Adding polynomial time functions in EAL

Here we explain very informally how we can add polynomial time functions in the calculus defined in [20], keeping the same kind of proof relying on the measure.

Suppose given a function f from integers to integers. We define a new constructor f in the classical EAL-calculus, and a new reduction rule $f\ \underline{n} \rightarrow \underline{f(n)}$, saying that f applied to the encoding of the integer n is reduced to the encoding of the integer $f(n)$. We add a cost to this reduction, depending on the integer n, that we call $C_f(n)$. We give a typing rule for this constructor, f has type $N \multimap N$.

If this function f is a polynomial time computable function, we can bound the cost function $C_f(n)$ by a polynomial function $(n+2)^d$ for a certain d, and we can also bound the size of $f(n)$ by the cost, and so $f(n) \leqslant (n+2)^d$. Now if we look at the reduction rule, if we call $\mu(f)$ the measure for f, we go from $\mu(f) + (1, n+1)$ to $(0, (n+2)^d)$, if we want to take in consideration the cost, we can add it in the measure, and suppose that in the right part of the reduction we have the measure $(0, 2(n+2)^d)$. Now, see that if $\mu(f) = (d, 1)$, this reduction follows the relation $\mathcal{R}$ defined in section 3, and with that we can deduce that this construction works with the measure.

6.4 Substitution lemmas in enriched EAL

$$\pi \lhd \frac{\tau \lhd \varnothing \mid \Delta, x : T' \vdash M : T}{\Gamma \mid \Delta', [\Delta], x : [T'] \vdash !M :\ !T} \qquad \mu_n(\pi) = (1, \mu_{n-1}(\tau))$$

We have $\tau \lhd \varnothing \mid \Delta, x : T' \vdash M : T$, and $\sigma \lhd \varnothing \mid \Delta \vdash M' : T'$. Moreover, the number of occurrences of the axiom rule for x in τ is bounded by $\omega_0(\tau)$. Indeed the axiom rule has a weight 1 at position 0, and the only constructor that can shift this weight is the !, but as x is not erased, we know that in a branch of τ that leads to an axiom rule, we do not cross a !.

So by using the general substitution lemma, we obtain a proof
$\tau' \lhd \varnothing \mid \Delta \vdash M[M'/x] : T$ with $\mu_{n-1}(\tau') \leqslant \mu_{n-1}(\tau) + \omega_0(\tau) \cdot \mu_{n-1}(\sigma)$. We can now build the proof

$$\pi' \lhd \frac{\tau' \lhd \varnothing \mid \Delta \vdash M[M'/x] : T}{\Gamma \mid \Delta', [\Delta] \vdash !M[M'/x] :\ !T} \qquad \mu_n(\pi') = (1, \mu_{n-1}(\tau'))$$

With $\mu_n(\pi') \leqslant (1, (\mu_{n-1}(\tau) + \omega_0(\tau) \cdot \mu_{n-1}(\sigma))) \leqslant (1, (\mu_n^1(\pi) + \omega_1(\pi) \cdot \mu_{n-1}(\sigma)))$
And so lemma 14 is verified.

6.5 Type system for words and boolean in EAL

$$\pi \lhd \frac{}{\Gamma \mid \Delta \vdash \epsilon : W} \quad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : W}{\Gamma \mid \Delta \vdash s_i(M) : W} \quad \mu_n(\pi) = \mu_n(\sigma) + \mathbb{1}_1$$

$$\pi \lhd \frac{\begin{array}{l}\sigma_1 \lhd \Gamma_1 \mid \Delta \vdash M_1 : W \multimap T \\ \sigma_0 \lhd \Gamma_0 \mid \Delta \vdash M_0 : W \multimap T \qquad \sigma \lhd \Gamma \mid \Delta \vdash M' : T\end{array}}{\Gamma_0, \Gamma_1, \Gamma \mid \Delta \vdash ifw(M_0, M_1, M') : W \multimap T} \quad \mu_n(\pi) = \mu_n(\sigma_0) + \mu_n(\sigma_1) + \mu_n(\sigma) + \mathbb{1}_0$$

$$\pi \lhd \frac{\begin{array}{l}\sigma_1 \lhd \Gamma_1 \mid \Delta \vdash M_1 :\ !(T \multimap T) \\ \sigma_0 \lhd \Gamma_0 \mid \Delta \vdash M_0 :\ !(T \multimap T) \qquad \sigma \lhd \Gamma \mid \Delta \vdash M :\ !T\end{array}}{\Gamma_0, \Gamma_1, \Gamma \mid \Delta \vdash iter^!_W(M_0, M_1, M) : W \multimap !T} \quad \mu_n(\pi) = \mu_n(\sigma_0) + \mu_n(\sigma_1) + \mu_n(\sigma) + \mathbb{1}_0$$

$$\pi \lhd \frac{}{\Gamma \mid \Delta \vdash \mathsf{tt} : B} \quad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi \lhd \frac{}{\Gamma \mid \Delta \vdash \mathsf{ff} : B} \qquad \mu_n(\pi) = \mathbb{1}_1$$

$$\pi \lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \tau \lhd \Gamma' \mid \Delta \vdash M' : T}{\Gamma, \Gamma' \mid \Delta \vdash if(M, M') : B \multimap T} \ \mu_n(\pi) = \mu_n(\sigma) + \mu_n(\tau) + \mathbb{1}_0$$

6.6 Lemmas for t_α and $\mathcal{R}$

Shift Lemma We want to prove the shift lemma, lemma 16. First we recall the statement of this lemma :
Let $k \in \mathbb{N}^*$. Let $\mu = (\omega_0, \ldots, k \cdot \omega_{i-1}, \omega_i, \ldots \omega_n)$ and $\mu' = (\omega_0, \ldots, \omega_{i-1}, k \cdot \omega_i, \ldots \omega_n)$.
Then $t_\alpha(\mu') \leqslant t_\alpha(\mu)$.

Let us define $\mu_0 = (\omega_0, \ldots, \omega_{i-2})$.

$$k \geqslant 1 \text{ so } k \leqslant 2^{2^{k-1}-1}, \text{ then}$$

$$k \cdot \omega_i \leqslant \omega_i \cdot 2^{2^{k-1}-1} \leqslant (\omega_i)^{2^{k-1}} \text{ since } w_i \geqslant 2. \text{ So,}$$

$$\alpha \cdot k \cdot \omega_i \leqslant (\alpha \cdot \omega_i)^{2^{(\alpha \cdot (k-1) \cdot \omega_{i-1})^{2^{t_\alpha(\mu_0)}}}}$$

$$(\alpha \cdot k \cdot \omega_i)^{2^{(\alpha \cdot \omega_{i-1})^{2^{t_\alpha(\mu_0)}}}} \leqslant (\alpha \cdot \omega_i)^{2^{(\alpha \cdot (k-1) \cdot \omega_{i-1})^{2^{t_\alpha(\mu_0)}}} \cdot 2^{(\alpha \cdot \omega_{i-1})^{2^{t_\alpha(\mu_0)}}}} \text{ and so,}$$

$$t_\alpha(\mu_0, (\omega_{i-1}, k \cdot \omega_i)) \leqslant (\alpha \cdot \omega_i)^{2^{(\alpha \cdot k \cdot \omega_{i-1})^{2^{t_\alpha(\mu_0)}}}} = t_\alpha(\mu_0, (k \cdot \omega_{i-1}, \omega_i))$$

We can now obtain $t_\alpha(\mu') \leqslant t_\alpha(\mu)$ by monotonicity of exponential.

This concludes the proof of lemma 16.

Link between t_α and $\mathcal{R}$ We want to prove Theorem 4. First let us recall the statement of this theorem :
Let $\mu, \mu' \in \mathbb{N}^{n+1}$ and $\alpha \geqslant n$, $\alpha \geqslant 1$. If $\mathcal{R}(\mu, \mu')$ then $t_\alpha(\mu') < t_\alpha(\mu)$

Suppose $\mathcal{R}(\mu, \mu')$. Using the notations from the definition of $\mathcal{R}$, we have
$\mu \geqslant (\omega_0, \ldots, \omega'_{i_0} + d, \omega_{i_0+1}, \ldots, \omega_n)$ and we have
$\mu' \leqslant (\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0}, \omega_{i_0+1} \cdot (\omega_{i_0+1})^{d-1}, \ldots, \omega_n \cdot (\omega_{i_0+1})^{d-1})$.

Let us call $\mu_0 = (\omega_0, \ldots, \omega_{i_0-1})$.

$$\alpha \cdot d \geqslant 1 \text{ so } \alpha \cdot d < 2^{\alpha \cdot d} \text{ so,}$$

$$\text{as } \omega_{i_0+1} \geqslant 2, \text{ we have } (\omega_{i_0+1})^{\alpha \cdot d} < (\omega_{i_0+1})^{2^{\alpha \cdot d}} \text{ so}$$

$$\alpha \cdot (\omega_{i_0+1})^{\alpha \cdot d} < (\alpha \cdot \omega_{i_0+1})^{2^{(\alpha \cdot d)^{2^{t_\alpha(\mu_0)}}}} \text{ and so}$$

$$(\alpha \cdot (\omega_{i_0+1})^{\alpha \cdot d})^{2^{(\alpha \cdot \omega'_{i_0})^{2^{t_\alpha(\mu_0)}}}} < (\alpha \cdot \omega_{i_0+1})^{2^{(\alpha \cdot (d + \omega'_{i_0}))^{2^{t_\alpha(\mu_0)}}}}$$

So we obtain
$t_\alpha(\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0}, (\omega_{i_0+1})^{\alpha \cdot d}) < t_\alpha(\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0} + d, \omega_{i_0+1})$.

By lemma 15, since $\omega_{i_0+1} \cdot (\omega_{i_0+1})^{(n-i_0)(d-1)} \leqslant (\omega_{i_0+1})^{\alpha \cdot d}$, and by monotonicity of the exponential, we obtain
$t_\alpha(\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0}, \omega_{i_0+1} \cdot (\omega_{i_0+1})^{(n-i_0)(d-1)}, \ldots, \omega_n) < t_\alpha(\omega_0, \ldots, \omega'_{i_0} + d, \omega_{i_0+1}, \ldots, \omega_n)$.

Using several times the shift lemma, we obtain
$t_\alpha(\omega_0, \ldots, \omega_{i_0-1}, \omega'_{i_0}, \omega_{i_0+1} \cdot (\omega_{i_0+1})^{d-1}, \ldots, \omega_n \cdot (\omega_{i_0+1})^{d-1}) < t_\alpha(\omega_0, \ldots, \omega'_{i_0} + d, \omega_{i_0+1}, \ldots, \omega_n)$.

Again by lemma 15, we obtain $t_\alpha(\mu') < t_\alpha(\mu)$

This finishes the proof of theorem 4.

6.7 Main theorem for the enriched EAL calculus

In this section, we prove the main theorem for the enriched EAL calculus :
Let $\tau \lhd \Gamma \mid \Delta \vdash M_0 : T$ and $M_0 \to M_1$. Let α be an integer equal or greater than the depth of τ. Then there is a proof $\tau' \lhd \Gamma \mid \Delta \vdash M_1 : T$ such that $\mathcal{R}(\mu_\alpha(\tau)+\tilde{2}, \mu_\alpha(\tau')+\tilde{2})$. Moreover, the depth of τ' is smaller than the depth of τ.

We prove this by first considering the base reductions. The case for the if-constructors are direct, it is a simple consequence of lemma 17. We detail the other cases :

- If $M_0 = (\lambda x.M)M'$ and $M_1 = M[M'/x]$, we have a proof :

$$\tau\lhd\ \dfrac{\dfrac{\pi \lhd \Gamma_1, x : T' \mid \Delta \vdash M : T}{\Gamma_1 \mid \Delta \vdash \lambda x.M : T' \multimap T} \qquad \sigma \lhd \Gamma_2 \mid \Delta \vdash M' : T'}{\Gamma_1, \Gamma_2 \mid \Delta \vdash (\lambda x.M)M' : T}$$

  $\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\sigma) + \mu_n(\pi) + 2 \cdot \mathbb{1}_0$.
  The proof $\tau' \lhd \Gamma_1, \Gamma_2 \mid \Delta \vdash M[M'/x] : T$ is given by the linear substitution lemma. As a consequence, we have $\forall n \in \mathbb{N}, \mu_n(\tau') \leqslant \mu_n(\pi) + \mu_n(\sigma)$.
  So we have $\forall n \in N, \mu_n(\tau') < \mu_n(\tau)$. As a consequence, it is still true for $n = \alpha \geqslant depth(\tau)$ and the depth of τ' is smaller than the depth of τ, moreover, by the lemma 17, we have
  $\mathcal{R}(\mu_\alpha(\tau)+\tilde{2}, \mu_\alpha(\tau')+\tilde{2})$
- If $M_0 = let\ !x =!M'\ in\ M$ and $M_1 = M[M'/x]$ then we have a proof :

$$\tau\lhd\ \dfrac{\dfrac{\sigma \lhd \varnothing \mid \Delta \vdash M' : T'}{\Gamma_1 \mid \Delta', [\Delta] \vdash !M' : !T'} \qquad \pi \lhd \Gamma_2 \mid \Delta', [\Delta], x : [T'] \vdash M : T}{\Gamma_1, \Gamma_2 \mid \Delta', [\Delta] \vdash let\ !x =!M'\ in\ M : T}$$

  $\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\pi) + (2, \mu_{n-1}(\sigma))$.
  By the discharged substitution lemma, we obtain a proof $\pi' \lhd \Gamma_2 \mid \Delta', [\Delta] \vdash M[M'/x] : T$, with
  $\forall n \in \mathbb{N}, \mu_n(\pi') \leqslant (\omega_0(\pi), (\mu^1_n(\pi) + \omega_1(\pi) \cdot \mu_{n-1}(\sigma)))$. We can now use the weakening lemma to obtain $\tau' \lhd \Gamma_1, \Gamma_2 \mid \Delta', [\Delta] \vdash M[M'/x] : T$. By the precedent upper-bound, we obtain
  $depth(\tau') \leqslant depth(\tau)$. Moreover, $\omega_0(\tau) - \omega_0(\tau') \geqslant 2$, and so for $\alpha \geqslant depth(\tau) \geqslant 0$, we have
  $\mu_\alpha(\tau') <_{lex} \mu_\alpha(\tau)$. Finally, for $\alpha \geqslant j > 0$, we have

$$\begin{aligned} \omega_j(\tau') + 2 &\leqslant \omega_j(\pi) + \omega_1(\pi) \cdot \omega_{j-1}(\sigma) + 2 \\ \omega_j(\tau') + 2 &\leqslant (\omega_j(\pi) + \omega_{j-1}(\sigma) + 2) \cdot (\omega_1(\pi) + \omega_0(\sigma) + 2) \\ \omega_j(\tau') + 2 &\leqslant (\omega_j(\tau) + 2) \cdot (\omega_1(\tau) + 2) \end{aligned}$$

  And so we obtain $\mathcal{R}(\mu_\alpha(\tau)+\tilde{2}, \mu_\alpha(\tau')+\tilde{2})$.
- If $M_0 = let\ x \otimes y = M \otimes M'\ in\ N$ and $M_1 = N[M/x][M'/y]$, we have a proof :

$$\tau\lhd\ \dfrac{\dfrac{\sigma \lhd \Gamma \mid \Delta \vdash M : T \qquad \sigma' \lhd \Gamma' \mid \Delta \vdash M' : T'}{\Gamma, \Gamma' \mid \Delta \vdash M \otimes M' : T \otimes T'} \qquad \pi \lhd \Gamma'', x : T, y : T' \mid \Delta \vdash N : T''}{\Gamma, \Gamma', \Gamma'' \mid \Delta \vdash let\ x \otimes y = M \otimes M'\ in\ N : T''}$$

  And $\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\pi) + \mu_n(\sigma) + \mu_n(\sigma') + 2 \cdot \mathbb{1}_0$
  Using two times the linear substitution lemma, we obtain a proof
  $\tau' \lhd \Gamma, \Gamma', \Gamma'' \mid \Delta \vdash N[M/x][M'/y] : T''$ with $\forall n \in \mathbb{N}, \mu_n(\tau') \leqslant \mu_n(\pi) + \mu_n(\sigma) + \mu_n(\sigma') < \mu_n(\tau)$. And so $depth(\tau') \leqslant depth(\tau)$ and for $\alpha \geqslant depth(\tau)$, $\mu_\alpha(\tau') < \mu_\alpha(\tau)$. By lemma 17, we have $\mathcal{R}(\mu_\alpha(\tau)+\tilde{2}, \mu_\alpha(\tau')+\tilde{2})$
- If $M_0 = iter^!_N(!M, !M')\ \underline{k}$ and $M_1 =!(M^k M')$, then we have a proof :

$$\tau\lhd\ \dfrac{\dfrac{\dfrac{\sigma_1 \lhd \varnothing \mid \Delta \vdash M : T \multimap T}{\Gamma_1 \mid \Delta', [\Delta] \vdash !M : !(T \multimap T)} \qquad \dfrac{\sigma_2 \lhd \varnothing \mid \Delta \vdash M' : T}{\Gamma_2 \mid \Delta', [\Delta] \vdash !M' : !T}}{\Gamma_1, \Gamma_2 \mid \Delta', [\Delta] \vdash iter^!_N(!M, !M') : N \multimap !T} \qquad \sigma \lhd \Gamma_3 \mid \Delta', [\Delta] \vdash \underline{k} : N}{\Gamma_1, \Gamma_2, \Gamma_3 \mid \Delta', [\Delta] \vdash iter^!_N(!M, !M')\underline{k} : !T}$$

And $\forall n \in \mathbb{N}, \mu_n(\tau) = \mu_n(\sigma) + (4, \mu_{n-1}(\sigma_1) + \mu_{n-1}(\sigma_2))$
Note that $\forall n \in \mathbb{N}, \mu_n(\sigma) = (k+1) \cdot \mathbb{1}_1$.
We can construct the proof τ' :

$$\tau' \lhd \dfrac{\dfrac{\sigma_1 \lhd \varnothing \mid \Delta \vdash M : T \multimap T \qquad \dfrac{\dfrac{\sigma_1 \lhd \varnothing \mid \Delta \vdash M : T \multimap T \qquad \sigma_2 \lhd \varnothing \mid \Delta \vdash M' : T}{\vdots}}{\varnothing \mid \Delta \vdash M^{k-1} M' : T}}{\varnothing \mid \Delta \vdash M^k M' : T}}{\Gamma_1, \Gamma_2, \Gamma_3 \mid \Delta', [\Delta] \vdash !(M^k M') : !T}$$

And $\forall n \in \mathbb{N}, \mu_n(\tau') = k \cdot \mathbb{1}_1 + (1, k \cdot \mu_{n-1}(\sigma_1) + \mu_{n-1}(\sigma_2))$.
We can see that $depth(\tau') \leqslant depth(\tau)$. Furthermore, we have $\omega_0(\tau) - \omega_0(\tau') \geqslant 2$, so for $\alpha \geqslant depth(\tau) \geqslant 0$, we have $\mu_\alpha(\tau') <_{lex} \mu_\alpha(\tau)$.
Moreover, $k \cdot (1 + \omega_0(\sigma_1)) + \omega_0(\sigma_2) + 2 \leqslant (k + 1 + \omega_0(\sigma_1) + \omega_0(\sigma_2) + 2)^2$
this means $\omega_1(\tau') + 2 \leqslant (\omega_1(\tau) + 2)^2$
And for $1 < j \leqslant \alpha$,
$\omega_j(\tau') + 2 = k \cdot \omega_{j-1}(\sigma_1) + \omega_{j-1}(\sigma_2) + 2 \leqslant (\omega_{j-1}(\sigma_1) + \omega_{j-1}(\sigma_2) + 2)(k + 1 + \omega_0(\sigma_1) + \omega_0(\sigma_2) + 2) = (\omega_j(\tau) + 2)(\omega_1(\tau) + 2)$.
We can conclude $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$
The proof for the rule $iter^!_W$ follows the same logic.

– If $M_0 = [\lambda x_k \ldots x_1.t](M'_1, \ldots, M'_{k-1}, \underline{v})$ and $M_1 = [\lambda x_{k-1} \ldots x_1.t[\underline{v}/x_k]](M'_1, \ldots, M'_{k-1})$, then we have a proof

$$\tau \lhd \dfrac{\begin{array}{c}\forall 1 \leqslant i \leqslant (k-1) \\ \sigma_i \lhd \Gamma_i \mid \Delta \vdash M'_i : A_i\end{array} \qquad \sigma \lhd \Gamma_k \mid \Delta \vdash \underline{v} : A_k \qquad \pi \lhd x_k : A_k^{a_k}, \ldots, x_1 : A_1^{a_1} \vdash_{\mathsf{s}\ell\mathsf{T}} t : A^I}{\Gamma, \Gamma_1, \ldots \Gamma_k \mid \Delta \vdash [\lambda x_k \ldots x_1.t](M'_1, \ldots, M'_{k-1}, \underline{v}) : A}$$

Note that the proof σ induces that $\underline{v}$ is either an actual integer $\underline{m}$, an actual word $\underline{w}$ or an actual boolean tt or ff. Moreover, $\forall n \in \mathbb{N}, \mu_n(\sigma) = |\underline{v}| \cdot \mathbb{1}_1$.
$\forall n \in \mathbb{N}, \mu_n(\tau) = \sum_{i=1}^{k-1} \mu_n(\sigma_i) + |v| \cdot \mathbb{1}_1 + k(d(\omega(\pi) + I) + 1) \cdot \mathbb{1}_0 + ((\omega(\pi) + I)[1/b_1] \cdots [1/b_l] + 1) \cdot \mathbb{1}_1$
With $\{b_1, \ldots, b_l\} = FV(I) \cup FV(\omega(\tau))$.
From the proof π, we can construct by lemma 3 a proof
$\pi[|\underline{v}|/a_k] \lhd x_k : A_k^{|\underline{v}|}, x_{k-1} : A_{k-1}^{a_{k-1}}, \ldots, x_1 : A_1^{a_1} \vdash t : A^{I[|\underline{v}|/a_k]}$. Note that if A_k is the boolean type B, we do not need this substitution. But the following proof is still correct for this case, it would be as if a_k is not a free variable, and so the substitution does nothing. For the other imprecise case when A is the boolean type B, just consider that $I = 1$ and so there is no substitution in I.
Furthermore, we can construct a proof $\sigma' \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} \underline{v} : A_k^{|\underline{v}|}$.
By the base-value substitution lemma in $\mathsf{s}\ell\mathsf{T}$, we have a proof
$\pi' \lhd x_{k-1} : A_{k-1}^{a_{k-1}}, \ldots, x_1 : A_1^{a_1} \vdash t[\underline{v}/x_k] : A^{I[|\underline{v}|/a_k]}$ and we have $\omega(\pi') \leqslant \omega(\pi)[|\underline{v}|/a_k]$.
We can now construct the proof τ' :

$$\tau' \lhd \dfrac{\begin{array}{c}\forall 1 \leqslant i \leqslant (k-1) \\ \sigma_i \lhd \Gamma_i \mid \Delta \vdash M'_i : A_i\end{array} \qquad \pi' \lhd x_{k-1} : A_{k-1}^{a_{k-1}}, \ldots, x_1 : A_1^{a_1} \vdash_{\mathsf{s}\ell\mathsf{T}} t[\underline{v}/x_k] : A^{I[|\underline{v}|/a_k]}}{\Gamma, \Gamma_k, \Gamma_1, \ldots \Gamma_{k-1} \mid \Delta \vdash [\lambda x_{k-1} \ldots x_1.t[\underline{v}/x_k]](M'_1, \ldots, M'_{k-1}) : A}$$

Let us denote $\{b'_1, \ldots, b'_{l'}\} = FV(I) \cup FV(\omega(\tau)) \cup FV(\omega(\tau'))$.
$\forall n \in \mathbb{N}, \mu_n(\tau') = \sum_{i=1}^{k-1} \mu_n(\sigma_i) + (k-1)(d(\omega(\pi') + I[|\underline{v}|/a_k]) + 1) \cdot \mathbb{1}_0 +$
$((\omega(\pi') + I[|\underline{v}|/a_k])[1/b'_1] \cdots [1/b'_{l'}] + 1) \cdot \mathbb{1}_1$.
With this, we can first see that $depth(\tau') \leqslant depth(\tau)$.

Moreover, by theorem 3, since $\omega(\pi') + I[|\underline{v}|/a_k] \leqslant (I + \omega(\pi))[|\underline{v}|/a_k]$, we have
$d(\omega(\pi') + I[|\underline{v}|/a_k]) \leqslant d((I + \omega(\pi))[|\underline{v}|/a_k]) \leqslant d(I + \omega(\pi))$.
By the theorem 3, $(I + \omega(\pi))[|\underline{v}|/a_k] \leqslant |\underline{v}|^{d(I+\omega(\pi))} \cdot (I + \omega(\pi))[1/a_k]$
By the lemma 3, $(I+\omega(\pi))[|\underline{v}|/a_k][1/b'_1,\ldots,b'_{l'}] \leqslant |\underline{v}|^{d(I+\omega(\pi))} \cdot (I+\omega(\pi))[1/b'_1,\ldots,b'_{l'}]$ (the substitution for a_k is either one of the b' by definition, either irrelevant if a_k does not appear in the indexes).
Now from those results, we have $\forall n \in \mathbb{N}$,
$\mu_n(\tau') \leqslant \sum_{i=1}^{k-1} \mu_n(\sigma_i) + (k-1)(d(\omega(\pi) + I) + 1) \cdot \mathbb{1}_0 + (|\underline{v}|^{d(I+\omega(\pi))} \cdot (I + \omega(\pi))[1/b'_1,\ldots,b'_{l'}] + 1)\mathbb{1}_1$.
Now we can prove $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$:
By the precedent bound we have $\omega_0(\tau) - \omega_0(\tau') \geqslant d(\omega(\pi) + I) + 1$.

$$\omega_1(\tau') + 2 \leqslant \sum_{i=1}^{k-1} \omega_1(\sigma_i) + |\underline{v}|^{d(I+\omega(\pi))} \cdot (I + \omega(\pi))[1/b'_1,\ldots,b'_{l'}] + 3$$
$$\omega_1(\tau') + 2 \leqslant (\sum_{i=1}^{k-1} \omega_1(\sigma_i) + |\underline{v}| + (\omega(\pi) + I)[1/b'_1,\ldots,b'_{l'}] + 3)^{d(\omega(\pi)+I)+1}$$
$$\omega_1(\tau') + 2 \leqslant (\omega_1(\tau) + 2) \cdot (\omega_1(\tau) + 2)^{d(\omega(\pi)+I)}$$

And for $1 < j \leqslant \alpha$,
$$\omega_j(\tau') + 2 \leqslant \sum_{i=1}^{k-1} \omega_j(\sigma_i) + 2 = \omega_j(\tau) + 2 \leqslant (\omega_j(\tau) + 2)(\omega_1(\tau) + 2)^{d(\omega(\pi)+I)}$$

This proves $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$.

- If $M_0 = [t_0]()$ and $M_1 = [t_1]()$ with $t_0 \to t_1$ in $\mathsf{s}\ell\mathsf{T}$. We have a proof :

$$\tau \lhd \frac{\pi \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} t_0 : A^I}{\Gamma \mid \Delta \vdash [t_0]() : A}$$

  $\forall n \in \mathbb{N}, \mu_n(\tau) = (1 + (\omega(\pi) + I)[1/b_1,\ldots,b_l]) \cdot \mathbb{1}_1$ with $\{b_1,\ldots,b_l\} = FV(I) \cup FV(\omega(\pi))$
  By the theorem 2, the main theorem of $\mathsf{s}\ell\mathsf{T}$, we have a proof $\pi' \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} t_1 : A^I$ with $\omega(\pi') < \omega(\pi)$. So we can construct

$$\tau' \lhd \frac{\pi' \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} t_1 : A^I}{\Gamma \mid \Delta \vdash [t_1]() : A}$$

  Let us denote $\{b'_1,\ldots,b'_{l'}\}$ all the free variables in I, $\omega(\pi)$ and $\omega(\pi')$.
  $\forall n \in \mathbb{N}, \mu_n(\tau') = (1 + (\omega(\pi') + I)[1/b'_1,\ldots,b'_{l'}]) \cdot \mathbb{1}_1$.
  We directly see that the depth does not increase. Remark that the depth of τ is greater than 1 in this case
  We have by lemma 3, $(\omega(\pi') + I)[1/b'_1,\ldots,b'_{l'}] < (\omega(\pi) + I)[1/b'_1,\ldots,b'_{l'}]$.
  And so, for $\alpha \geqslant depth(\tau) \geqslant 1, \mu_\alpha(\tau') < \mu_\alpha(\tau)$, and so we have $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$.
  Remark that as opposed to all the precedent cases, $\mu_0(\tau)$ and $\mu_0(\tau')$ are equal, and so we need to look at position 1 to see that the measure strictly decreases. This remark is primordial in the proof of another lemma, lemma 19.
- If $M_0 = [\underline{v}]()$ and $M_1 = \underline{v}$. The fact that M_0 can be typed by τ indicates that $\underline{v}$ is either an actual integer, a word or a boolean. With this remark, the typing τ' of M_1 is just the usual typing for those values. Moreover, we know the weight in $\mathsf{s}\ell\mathsf{T}$ and the measure in EAL for the typing proof of a value, in $\mathsf{s}\ell\mathsf{T}$ the weight is 0 and in EAL the measure is $|\underline{v}| \cdot \mathbb{1}_1$. Furthermore, if $\pi \lhd \cdot \vdash_{\mathsf{s}\ell\mathsf{T}} \underline{v} : A^I$, then we know that $|\underline{v}| \leqslant I$. With this, we have $\mu_n(\tau) = (1 + I[1/FV(I)]) \cdot \mathbb{1}_1$ and $\mu_n(\tau') = |\underline{v}| \cdot \mathbb{1}_1$. By the lemma 3, we have $|\underline{v}| \leqslant I[1/FV(I)]$ and so for $n \geqslant 1$, $\mu_n(\tau') < \mu_n(\tau)$.
  This gives us $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$. And the fact that the depth does not increase is direct.
  Remark that as the precedent case, we need to look at position 1 to see that the measure strictly decreases.

Now we need to work on the reductions under a context. For this we work by induction on contexts, and what we have done previously is the base case. For any inductive case of context except the ! case, the subject reduction and the fact the the depth does not increase are direct. Then, the relation between the measures is a consequence of lemma 18

When the context has the form $C = !C'$, the notion of depth is crucial for this case. Indeed suppose $M \to M'$, $M_0 = !M$ and $M_1 = !M'$. With the proof τ for M_0, we obtain a proof π for M, this gives us by induction hypothesis a proof π' for M', and this gives us a proof τ' for M_1.

Moreover, $\forall n \in \mathbb{N}, \mu_n(\tau) = (1, \mu_{n-1}(\pi))$ and $\mu_n(\tau') = (1, \mu_{n-1}(\pi'))$.

As $depth(\pi') \leqslant depth(\pi)$ we have $depth(\tau') = depth(\pi') + 1 \leqslant depth(\pi) + 1 = depth(\tau)$. And for $\alpha \geqslant depth(\tau)$, then $(\alpha - 1) \geqslant depth(\pi)$. And with $\mathcal{R}(\mu_{\alpha-1}(\pi) + \tilde{2}, \mu_{\alpha-1}(\pi') + \tilde{2})$ given by the induction hypothesis, we can easily deduce $\mathcal{R}(\mu_\alpha(\tau) + \tilde{2}, \mu_\alpha(\tau') + \tilde{2})$.

Remark that this proof shows that if we had $\mathcal{R}(\mu_n(\pi) + \tilde{2}, \mu_n(\pi') + \tilde{2})$ we obtain $\mathcal{R}(\mu_{n+1}(\tau) + \tilde{2}, \mu_{n+1}(\tau') + \tilde{2})$. This remark is important for the proof of lemma 19.

This concludes the proof of theorem 5.

6.8 i-values and i-normal forms

We want to prove lemma 20, saying that
Let M be a term. If M is closed and has a typing derivation then,
for all $i \in \mathbb{N}$, if M is normal for i-reductions then M is a i-value V^i.

We prove that by induction on terms. Note that the case $i = 0$ is always direct since i-values are all terms, so we only need to work for $i \in \mathbb{N}^*$. We detail here some cases :

- Let $i \in \mathbb{N}^*$. If $M = M_0\ M_1$, M closed and typed. Suppose, by contradiction, that M is normal for i-reductions. Then, M_0 and M_1 are also normals for i-reductions. And they are also closed and typed. So by induction hypothesis, M_0 and M_1 are i-values. By typing, we have a proof $\cdot \mid \cdot \vdash M_0 : T \multimap T'$. And so we have different possible case for M_0 :
  - $M_0 = \lambda x.M'$, in this case we have $M \to_i M'[M_1/x]$, and this contradicts the fact that M is normal for i-reductions.
  - $M_0 = ifn(V_0^i, V_1^i)$. In this case, M_1 has type N. But the only i-values of type N are actual integers $\underline{n}$ since $i \geqslant 1$, and so we can reduce M by the ifn-rule. This contradicts our hypothesis.
  - For $iter_N^!$, if, ifw and $iter_W^!$, we have a similar proof as the one for the ifn case.

  Finally, we have indeed a contradiction. So M is not normal for i-reductions.
- If $M = !M'$, M closed and typed. Then M' is also closed and typed. Let $i \in \mathbb{N}^*$. Suppose that M is normal for i-reductions, then by definition, M' is normal for $(i-1)$-reductions. So, by induction hypothesis, M' is a $(i-1)$-value, and so M is indeed a i-value.
- If $M = let\ !x = M_0\ in\ M_1$, M closed and typed, then M_0 is also closed and typed. Let $i \in \mathbb{N}^*$, and suppose that M is normal for i-reductions. Then by definition M_0 is also normal for i-reductions, and so by induction hypothesis, M_0 is a i-value. Furthermore, by typing, M_0 had a type of the form $!T$, and since $i \geqslant 1$ and M_0 is a i-value, we obtain that M_0 has the form $M_0 = !M'$, and so M can be reduced.
- If $M = [\lambda x_k \ldots x_1.t](M_1, \ldots, M_k)$. M closed and typed. If $k > 0$, then as previously, if we suppose M normal, we obtain that M_k must be of the form $\underline{v}$, and this is absurd since for $i \geqslant 1$, M would be reducible. If $k = 0$, then if we suppose M normal, it means that t is normal, closed, and typed, and so by theorem 1, t is of a form $\underline{v}$, and so M is not normal.

This concludes the proof.

6.9 Complexity Results

We recall the notations. We are given two variables $w : W^{a_w}$ and $n : N^{a_n}$. n is the number of steps of the Turing machine and w is the input. $Conf_b$ is the type $W^{a_w+b} \otimes B \otimes W^{a_w+b} \otimes B^q$. This type represents a configuration on a Turing machine after b steps, with B^q coding the state, and then $\underline{w_0} \otimes b \otimes \underline{w_1}$ represents the tape, with b being the head, w_0 represents the reverse of the word before b, and w_1 represents the word after b.

We give her some terms used in the simulation of a Turing-machine :

- Given an initial state s of size q, we can code this state in a term $s : B^q$. Then we can construct the term $init$ with type
  $w : W^{a_w}, n : N^{a_n} \vdash init : Conf_1$. For this, pose
  $init = \epsilon \otimes (ifw(\lambda w'.\mathsf{ff} \otimes w', \lambda w'.\mathsf{tt} \otimes w', \mathsf{ff} \otimes \epsilon)\ w) \otimes s$.
- Given for each state s of size q and boolean b a transition function
  $\delta(b, s) \subset \{left, right, stay\} \times \{0,1\} \times \{0,1\}^q$, we can construct a term $step$ with type $\cdot \vdash step : Conf_b \multimap Conf_{b+1}$. For this, pose
  $step = \lambda c.let\ x \otimes b \otimes y \otimes s = c\ in\ (case_{q+1}(t_{0^{q+1}}, \ldots, t_{1^{q+1}}))\ (b \otimes s)$.
  $b \otimes s$ can be seen as a binary word of size $q+1$, and we define $t_{b\otimes s}$ according to $\delta(b, s)$. For example, if $\delta(b, s) = (left, b', s')$, we define
  $t_{b\otimes s} = (ifw(\lambda w.w \otimes \mathsf{ff}, \lambda w.w \otimes \mathsf{tt}, \epsilon \otimes \mathsf{ff})\ x) \otimes s_{b'}(y) \otimes s'$. Remark that the duplicated variables are x and y, which are base type variables.
- Given for states of size q a function $accept : B^q \to B$, we can construct a term $final$ with $\cdot \vdash final : Conf_b \multimap B$, defined by
  $final = \lambda c.let\ x \otimes b \otimes y \otimes s = c\ in\ case_q(t_{0^q}, t_{1^q})\ s$, with $t_s = accept(s)$.

Now, suppose given a one-tape deterministic Turing machine TM on binary words such that for words w, TM works in time $2_{2k}^{P(|w|)}$. TM has an infinite tape, this means that on an input w, the Turing-machine can read outside the bound of w and in this case, it reads a 0. We can compute a term in EAL t_{TM} such that $\cdot \mid \cdot \vdash t_{TM} :!W \multimap !^{k+1}B$ and on an input $!w$, the term reduces to the term $!^{k+1}b$ with $b = \mathsf{tt}$ if w is accepted by TM, and $b = \mathsf{ff}$ otherwise.

For this, we show how to decompose the work in order to construct this term.

1. We duplicate the word given in input.
2. With one of those words, we compute the length of the word, using the term
   $\cdot \mid \cdot \vdash \lambda w.[\lambda x_1.length\ x_1](w) : W \multimap N$
3. Now that we have the size, we can compute $2_{2k}^{P(|w|)}$ following the results from section 3.4. So we obtain $!w \otimes !^{k+1}n$ with n representing $2_{2k}^{P(|w|)}$. By using the coercion, we obtain $!^{k+1}w \otimes !^{k+1}n$
4. Now in $\mathsf{s}\ell\mathsf{T}$, given a word $w : W^{a_w}$ and an integer $n : N^{a_n}$, we can code the input w in the type $Conf_1$ using the term $init$, by using for states a size q such that all states of TM can be coded in binary words of size q. Moreover, given the term $\cdot \vdash step : Conf_b \multimap Conf_{b+1}$ that represents one step of the Turing machine TM, we can use the constructor $itern$ to create a term of type $N^{a_n} \multimap Conf_{a_n}$
5. Given this term, we can apply it to n to simulate n steps of the Turing-machine TM. And by using the term $final$ we can extract the result of the computation
6. This simulation in $\mathsf{s}\ell\mathsf{T}$ allows us to construct a term in EAL that simulates the Turing-machine TM with type $(!^{k+1}W \otimes !^{k+1}N) \multimap !^{k+1}B$.

So in conclusion, we can effectively simulate TM in a term of type $!W \multimap !^{k+1}B$.

Bibliography

[1] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *ACM Transactions on Computational Logic (TOCL)*, 3(3):383–401, 2002.

[2] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *PACMPL*, 1(ICFP):43:1–43:29, 2017.

[3] Patrick Baillot. On the expressivity of elementary linear logic: Characterizing ptime and an exponential time hierarchy. *Information and Computation*, 241:3–31, 2015.

[4] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit computational complexity of subrecursive definitions and applications to cryptographic proofs (long version). ENS Lyon, 2015.

[5] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Information and Computation*, 248:56–81, 2016.

[6] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing polynomial and exponential complexity classes in elementary lambda-calculus. In *IFIP International Conference on Theoretical Computer Science*, pages 151–163. Springer, 2014.

[7] Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *European Symposium on Programming*, pages 104–124. Springer, 2010.

[8] Patrick Baillot and Kazushige Terui. A feasible algorithm for typing in elementary affine logic. In *TLCA*, volume 5, pages 55–70. Springer, 2005.

[9] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293. ACM, 1992.

[10] Guillaume Bonfante, J-Y Marion, and J-Y Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

[11] Ugo Dal Lago and Barbara Petit. The geometry of types. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Proceedings*, pages 167–178. ACM, 2013.

[12] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. *Science of Computer Programming*, 84:77–100, 2014.

[13] Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In *International Workshop on Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2011.

[14] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123–137, 2003.

[15] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[16] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *International Workshop on Computer Science Logic*, pages 275–294. Springer, 1997.

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

[18] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423. ACM Press, 1996.

[19] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8(4), 2011.

[20] Antoine Madet and Roberto M Amadio. An elementary affine λ-calculus with multithreading and side effects. In *International Conference on Typed Lambda Calculi and Applications*, pages 138–152. Springer, 2011.

[21] John Mitchell, Mark Mitchell, and Andre Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 725–733. IEEE, 1998.

[22] Kazushige Terui. Light affine lambda calculus and polytime strong normalization. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 209–220. IEEE, 2001.