Combining Linear Logic and Size Types for Implicit Complexity

Patrick Baillot
Univ Lyon, CNRS, ENS de Lyon, Université Claude-Bernard Lyon 1, LIP
F-69342, Lyon Cedex 07, France

Alexis Ghyselen
ENS Paris-Saclay, 94230 Cachan, 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 restricted versions of its !-modality controlling duplication. A second approach relies on the idea of tracking the size increase between input and output, and together with a restricted recursion scheme, to deduce time complexity bounds. However both approaches suffer from limitations : either a limited intensional expressivity, or linearity restrictions. In the present work we incorporate both approaches into a common type system, in order to overcome their respective constraints. Our system is based on elementary linear logic combined with linear size types, called sEAL, and leads to characterizations of the complexity classes FPTIME and 2k-FEXPTIME, for k >= 0.

2012 ACM Subject Classification Theory of computation → Lambda calculus, Theory of computation → Linear logic, Theory of computation → Turing machines, Software and its engineering → Functional languages

Keywords and phrases Implicit computational complexity, λ-calculus, linear logic, type systems, polynomial time complexity, size types

Digital Object Identifier 10.4230/LIPIcs.CSL.2018.9

Related Version Combining Linear Logic and Size Types for Implicit Complexity (Long Version), Patrick Baillot, Ghyselen Alexis, `https://hal.archives-ouvertes.fr/hal-01687224`, 2018

1 Introduction

Controlling the time complexity of programs is a crucial aspect of program development. Complexity analysis can be performed on the overall 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 ...

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 [18], for oracle functionals or for probabilistic computation [13, 25]. 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 $k \geq 0$. [15, 2, 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 from a programming point of view 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 [19] : 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. This has triggered a fertile research line on type-based complexity analysis using ideas of amortized cost analysis [20, 17, 16]. Some aspects of higher-order have been adressed [22] but note that this approach deals with complexity analysis and not with the characterization of complexity classes. In particular it does not suggest disciplines to program within a given complexity class. A similar idea is also explored by the line of work on quasi-interpretations [10, 4], 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 [3] can be thought of as playing this role of describing the dependence between input and output size, and it allows to derive time complexity bounds, even though these are not limited to polynomial bounds. Altogether we will refer to these approaches as *size-based systems*. However they also have a limitation: characterizations of complexity classes have not been obtained for full-fledged higher-order languages, but only for linear higher-order languages, that is to say languages in which functional arguments have to be used at most once (as in [19, 4]).

Problematic and methodology. 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 characterize complexity classes within a general higher-order language... On both sides some attempts have been made to repair these shortcomings but only with limited success: in [6] for instance LLL is extended to a language with recursive definitions, but the main expressivity problem remains; in [4] quasi-interpretations are defined for a higher-order language, but with a linearity condition on functional arguments. The goal of the present work is precisely to improve this situation 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.

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 [6] 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 [24].

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 $\mathsf{W} \multimap !^i\mathsf{W}$ (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 $k \geq 0$ [2]. The paper [24] 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 $\mathsf{W} \multimap \mathsf{W}$. 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 [3], a descendant of previous work on linear dependent types [23]. 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 [21, 1] 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. We will call it sEAL, for *sized Elementary affine logic* typed λ-calculus. We do not include polymorphism in sEAL for the simplicity of exposition, but we are convinced that our results could be adapted to the polymorphic extension.

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

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

We present $\mathsf{s}\ell\mathsf{T}$ which is a linear λ-calculus with constructors for base types and 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.

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

Figure 1 Base rules for sℓT.

2.1 Syntax of sℓT and Type System

▶ **Definition 1** (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 sℓT are defined by the following grammars :

$t := x \mid \lambda x.t \mid t\ t' \mid t\otimes t' \mid \texttt{let}\ x\otimes y = t\ \texttt{in}\ t' \mid \texttt{zero} \mid \texttt{succ}(t) \mid \texttt{ifn}(t,t') \mid \texttt{itern}(V,t) \mid \epsilon \mid \texttt{s}_0(t) \mid \texttt{s}_1(t) \mid \texttt{ifw}(t_0,t_1,t') \mid \texttt{iterw}(V_0,V_1,t) \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{if}(t,t')$

$V := x \mid \lambda x.t \mid V\otimes V' \mid \texttt{zero} \mid \texttt{succ}(V) \mid \texttt{ifn}(V,V') \mid \texttt{itern}(V,V') \mid \epsilon \mid \texttt{s}_0(V) \mid \texttt{s}_1(V) \mid \texttt{ifw}(V_0,V_1,V') \mid \texttt{iterw}(V_0,V_1,V') \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{if}(V,V')$

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 $\texttt{s}_i$ to regroup the cases $\texttt{s}_0$ and $\texttt{s}_1$. Here, we choose the alphabet $\{0,1\}$ for simplification, but we could have taken any finite alphabet Σ and in this case, the constructors $\texttt{ifw}$ and $\texttt{iterw}$ 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 $\texttt{ifn}(t,t')$ can be seen as $\lambda n.\texttt{match}\ n\ \texttt{with}\ \texttt{succ}(n') \mapsto t\ n' \mid 0 \mapsto t'$, and the constructor $\texttt{itern}(V,t)$ is such that $\texttt{itern}(V,t)\ \underline{n} \to^* V^n\ t$, if $\underline{n}$ is the coding of the integer n, that is $\texttt{succ}^n(\texttt{zero})$.

Reductions. Base reductions in sℓT are given by the rules described in Figure 1.

Note that in the $\texttt{iterw}$ 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 [3]. This is not a 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 : $C := [] \mid C\ t \mid V\ C \mid C\otimes t \mid t\otimes C \mid \texttt{let}\ x\otimes y = C\ \texttt{in}\ t \mid \texttt{succ}(C) \mid \texttt{ifn}(C,t) \mid \texttt{ifn}(t,C) \mid \texttt{itern}(V,C) \mid \texttt{s}_i(C) \mid \texttt{ifw}(C,t,t') \mid \texttt{ifw}(t,C,t') \mid \texttt{ifw}(t,t',C) \mid \texttt{iterw}(V_0,V_1,C) \mid \texttt{if}(C,t) \mid \texttt{if}(t,C)$.

Linear Types with Sizes. Base types are given by the following grammar :

$U := \mathsf{W}^I \mid \mathsf{N}^I \mid \mathsf{B}$ $\qquad$ $I,J,\cdots := a \mid n\in\mathbb{N}^* \mid I+J \mid I\cdot J$

$\mathbb{N}^*$ is the set of non-zero integers. I represents an *index* and a represents an *index variable*. We define for indexes the notions of free variables and free occurrences in the usual way and we work up to renaming of variables. We also define the substitution of a free variable in an index in the usual way. Then, we can generalize substitution to types, with for example $\mathsf{N}^I[J/a] = \mathsf{N}^{I[J/a]}$.

The intended meaning is that closed values of type N^I (resp. W^I) will be integers (resp. words) of size (resp. length) at most I.

▶ **Definition 2** (Order on Indexes). For two indexes I and J, we say that $I \le J$ if for any valuation ϕ mapping free variables of I and J to non-zero integers, we have $I_\phi \le J_\phi$. 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 $I \leq J$ and $J \leq I$ then $I = J$ (ie we take the quotient set for the equivalence relation). Remark that by definition of indexes, we always have $1 \leq I$. 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 $I \leq J$ and $I \neq J$, as we can see with $a \leq a \cdot b$.

Here we only consider polynomial indexes. This is a severe restriction w.r.t. linear dependent types, used for example in [12, 3], in which indexes can use any set of functions described by some rewrite rules. But in the present setting this is sufficient because we only want $\mathsf{s}\ell\mathsf{T}$ to characterize polynomial time computation.

▶ **Definition 3.** Types are given by the grammar $D, E, \cdots := U \mid D \multimap D' \mid D \otimes D'$

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

- $\mathsf{B} \sqsubset \mathsf{B}$ and if $I \leq J$ then $\mathsf{N}^I \sqsubset \mathsf{N}^J$ and $\mathsf{W}^I \sqsubset \mathsf{W}^J$.
- $D_1 \multimap D_1' \sqsubset D_2 \multimap D_2'$ 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 $D_1' \sqsubset D_2'$.

▶ **Definition 4** (Contexts). *Variables contexts* are denoted Γ, with the shape $\Gamma = x_1 : D_1, \ldots, x_n : D_n$. 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. The rules for words and booleans can be found in the appendix 6.1, they can be deduced from the rules for integers. Observe that this system enforces a linear usage of variables of non-base types (see e.g. the rule for application in Fig. 2). Note that in the rule for `itern` described in Figure 2, the index variable a must be a fresh variable.

Example in $\mathsf{s}\ell\mathsf{T}$. We sketch here the multiplication in $\mathsf{s}\ell\mathsf{T}$, other examples can be found in the appendix 6.3. The multiplication can be written $mult = \lambda x.\texttt{itern}(\lambda y.add\ x\ y, \texttt{zero}) : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I \cdot J}$, if we are given the term $add : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I+J}$.

$$\dfrac{\dfrac{x : \mathsf{N}^I, y : \mathsf{N}^{I \cdot a} \vdash add\ x\ y : \mathsf{N}^{I \cdot a + I}}{x : \mathsf{N}^I \vdash \lambda y.add\ x\ y : \mathsf{N}^{I \cdot a} \multimap \mathsf{N}^{I \cdot a}[a+1/a]} \quad \dfrac{}{x : \mathsf{N}^I \vdash \texttt{zero} : \mathsf{N}^I}}{x : \mathsf{N}^I \vdash \texttt{itern}(\lambda y.add\ x\ y, \texttt{zero}) : \mathsf{N}^J \multimap \mathsf{N}^{I \cdot J}}$$

2.2 Subject Reduction and Upper Bound

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 give some important intermediate lemmas. Other lemmas can be found in the appendix 6.2, and more details are available in [7], as for other sections in this paper. 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 $\texttt{succ}(\texttt{succ}(\ldots(\texttt{succ}(\texttt{zero}))\ldots))$. This imposes that in this call-by-value calculus, when an argument is of type N, it is the encoding of an integer.

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

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

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

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

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

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

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

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

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

Figure 2 Type system for $\mathsf{s}\ell\mathsf{T}$.

▶ **Theorem 5.** *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.*

Another important lemma is the one for subtyping.

▶ **Lemma 6** (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 $\omega(\pi') \leq \omega(\pi)$*

This lemma shows that we do not need an explicit rule for subtyping and subtyping does not harm the upper bound derived from typing. Moreover, this lemma is important in order to substitute variables, since the axiom rule allows subtyping.

We can now express the subject-reduction of the calculus and the fact that the weight of a proof strictly decreases during a reduction.

▶ **Theorem 7.** *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 [7]. The main difficulty is to prove the statement for base reductions. Base reductions that induce a substitution, like the usual β reduction, are proved by a substitution lemma. The other 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.

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

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.

▶ **Definition 8.** 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, $d(a) = 1$
- $d(I + J) = max(d(I), d(J))$
- $d(I \cdot J) = d(I) + d(J)$.

This definition of degree is primordial for the control of reductions in sEAL, that we present in the following section.

3 Elementary Affine Logic and Sizes

We work on an elementary affine lambda calculus based on [24] 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 [24] is robust enough to support functions computable in polynomial time with type $\mathsf{N} \multimap \mathsf{N}$ (see Section 6.4 in the appendix), we enrich EAL with the polynomial time calculus defined previously. We call this new language sEAL (EAL with sizes). 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 [24] to sEAL to find an upper-bound on the number of reductions for a term.

3.1 An EAL-Calculus

First, let us present a λ-calculus for the 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 [24] and we encode the usual restrictions in a type system.

Syntax. Terms are given by the grammar: $M := x \mid \lambda x.M \mid M\ M' \mid !M \mid \texttt{let}\ !x = M\ \texttt{in}\ M'$

The constructor $\texttt{let}\ !x = M\ \texttt{in}\ M'$ 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

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

Those rules can be applied in any contexts.

Type System. We add to this calculus a polymorphic type system that also restrains the possible terms we can write. Types are given by the grammar $T := \alpha \mid T \multimap T' \mid !T \mid \forall \alpha.T$

▶ **Definition 9** (Contexts). *Linear variables contexts* are denoted Γ, with the shape $\Gamma = x_1 : T_1, \ldots, x_n : T_n$. We write Γ_1, Γ_2 the disjoint union between Γ_1 and Γ_2. *Global variables contexts* are denoted Δ, with the shape $\Delta = x_1 : T_1, \ldots, x_n : T_n, y_1 : [T'_1], \ldots y_n : [T'_m]$. We say that [T] is a *discharged type*, as we could see in light linear logic [15, 26]. When we need to separate the discharged types from the others, we will write $\Delta = \Delta'', [\Delta']$. In this case, if $[\Delta'] = y_1 : [T'_1], \ldots, y_m : [T'_m]$, then we note $\Delta' = y_1 : T'_1, \ldots, y_m : T'_m$.

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

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

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

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

Figure 3 Type system for the EAL-calculus.

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

The rules are given in Figure 3. 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 $\texttt{let}\ !x = M\ \texttt{in}\ M'$, 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 $\texttt{let}\ !x = M\ \texttt{in}\ x$.

With this type system, we obtain as a consequence of the results exposed in [24] 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 those two notions in the following part on sEAL, but for now, let us present some terms and encoding in this EAL-calculus.

Examples of Terms in EAL and Church Integers. First, a useful term proving the functoriality of $! : fonct = \lambda f, x.\texttt{let}\ !g = f\ \texttt{in}\ \texttt{let}\ !y = x\ \texttt{in}\ !(g\ y) : \forall \alpha, \alpha'.!(\alpha \multimap \alpha') \multimap !\alpha \multimap !\alpha'$.

Integers can be encoded in this calculus, using the type $\mathsf{N} = \forall \alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$. For example, 3 is described by the term $\underline{3} = \lambda f.\texttt{let}\ !g = f\ \texttt{in}\ !(\lambda x.g\ (g\ (g\ x))) : \mathsf{N}$.

With this encoding, addition and multiplication can be defined, with type $\mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$.
$add = \lambda n, m, f.\texttt{let}\ !f' = f\ \texttt{in}\ \texttt{let}\ !g = n\ !f'\ \texttt{in}\ \texttt{let}\ !h = m\ !f'\ \texttt{in}\ !(\lambda x.h\ (g\ x))$
$mult = \lambda n, m, f.\texttt{let}\ !g = f\ \texttt{in}\ n(m\ !g)$

And finally, one can also define an iterator using integers.
$iter = \lambda f, x, n.fonct\ (n\ f)\ x : \forall \alpha.!(\alpha \multimap \alpha) \multimap !\alpha \multimap \mathsf{N} \multimap !\alpha$ with $iter\ !M\ !M'\ \underline{n} \to^* !(M^n\ M')$.

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 sometimes 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 types with bangs. This implies that each time we need to use an iteration, we are forced to add a bang in the final type. Typically this prevents from iterating a function which has itself been defined by iteration. It has been proved [5] 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 $!\mathsf{W} \multimap !!\mathsf{B}$ characterizes polynomial time computation. However, because of the restrictions mentioned above some natural polynomial time programs cannot be typed with the type $!\mathsf{W} \multimap !!\mathsf{B}$. We say that this calculus has a limited intensional expressivity. One goal of this paper is to try to lessen this problem, and for that, we now present an enriched version of this calculus, sEAL, using the language $\mathsf{s}\ell\mathsf{T}$.

3.2 Syntax and Type System for sEAL

Notation. Let us first give some notations on terms and vectors.

▶ **Definition 10** (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 $M^{aw'} M' = M_a\ (M^{w'} M')$

▶ **Definition 11** (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 $\mu = (\mu(0), \dots, \mu(n))$. When there is no ambiguity with the value of n, for $0 \leq k \leq n$, we note $\mathbb{1}_k$ the vector μ with $\mu(k) = 1$ and $\forall i, 0 \leq i \leq n, i \neq k, \mu(i) = 0$. 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 $\forall i, 0 \leq i \leq n, \mu(i) = \mu_0(i)$ and $\forall i, 0 \leq i \leq m, \mu(i+n+1) = \mu_1(i)$. Let $\mu_0, \mu_1 \in \mathbb{N}^{n+1}$. We write $\mu_0 \leq \mu_1$ when $\forall i, 0 \leq i \leq n, \mu_0(i) \leq \mu_1(i)$. And we write $\mu_0 < \mu_1$ when $\mu_0 \leq \mu_1$ and $\mu_0 \neq \mu_1$. We also write $\mu_0 \leq_{lex} \mu_1$ 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 $\forall i, 0 \leq i \leq n, \mu(i) = k$.

Terms and Reductions. Terms of sEAL are defined by the following grammar :

$M := x \mid \lambda x.M \mid M\ M' \mid !M \mid \mathtt{let}\ !x = M\ \mathtt{in}\ M' \mid M \otimes M' \mid \mathtt{let}\ x \otimes y = M\ \mathtt{in}\ M'$
$\mid \mathtt{zero} \mid \mathtt{succ}(M) \mid \mathtt{ifn}(M, M') \mid \mathtt{iter}^!_N(M, M') \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(M, M') \mid \epsilon \mid \mathtt{s}_0(M) \mid \mathtt{s}_1(M)$
$\mid \mathtt{ifw}(M_0, M_1, M) \mid \mathtt{iter}^!_W(M_0, M_1, M) \mid [\lambda x_n \dots x_1.t](M_1, \dots, M_n)$

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 EAL. Once again, we often write $\mathtt{s}_i$ to denote $\mathtt{s}_0$ or $\mathtt{s}_1$, and the choice of the alphabet $\{0, 1\}$ is arbitrary, we could have used any finite alphabet. As usual, we work up to α-isomorphism and we do not explicit the renaming of variables.

▶ **Definition 12** (Base type values). We note $\underline{v}$ for base type values, defined by the grammar $\underline{v} := \mathtt{zero} \mid \mathtt{succ}(\underline{v}) \mid \epsilon \mid \mathtt{s}_i(\underline{v}) \mid \mathtt{tt} \mid \mathtt{ff}$.

In particular, if n is an integer and w is a binary word, we note $\underline{n}$ for the base value $\mathtt{succ}^n(\mathtt{zero})$, and $\underline{w} = \underline{w_1 \cdots w_n}$ for the base value $\mathtt{s}_{w_1}(\dots \mathtt{s}_{w_n}(\epsilon) \dots)$. We define the *size* $|\underline{v}|$ of $\underline{v}$ by $|\mathtt{zero}| = |\epsilon| = |\mathtt{tt}| = |\mathtt{ff}| = 1$ and $|\mathtt{succ}(\underline{v})| = |\mathtt{s}_i(\underline{v})| = 1 + |\underline{v}|$.

Base reductions are defined by the rules given in Figure 4. 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 sEAL term.

Those reductions can be extended to any contexts, and so we have $M \to M'$ if there is a context C and a base reduction $M_0 \to M'_0$ such that $M = C(M_0)$ and $M' = C(M'_0)$. However, the scope of those contexts does not allow context reduction in $\mathsf{s}\ell\mathsf{T}$. For reduction in $\mathsf{s}\ell\mathsf{T}$, we use the last reduction rule.

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 grammar : $A := \mathsf{B} \mid \mathsf{N} \mid \mathsf{W}$. Types are given by the grammar : $T := A \mid T \multimap T' \mid !T \mid T \otimes T'$

▶ **Definition 13** (Contexts and Type System). *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 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* $\omega_i(\pi) \in \mathbb{N}$.

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

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

Figure 4 Base rules for sEAL.

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

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

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

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

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

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

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

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

Figure 5 Type and measure for generic constructors in sEAL.

▶ **Definition 14** (Measure and Depth). 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. The rules are given on Figures 5, 6 and 7, and the rules for words and booleans can be found in the appendix 6.5.

The rules given in figure 5 represent the usual constructors in EAL. Those rules impose some restrictions in the use of variables similar to the one described in the previous section on classical EAL. Remark that the constructors for base types values such as `zero` and `succ` given in Figure 6 influence the weight only in position 1 and not 0 like the others constructors.

For the rule given by Figure 7, some explanations are necessary. The premise for t is a proof τ in $\mathsf{s\ell T}$. 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 $\mathsf{s\ell T}$ there is no indexes on the

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

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

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

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

Figure 6 Type and measure for constructors on integers in sEAL.

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

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

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

Figure 7 Typing rule and measure for the $\mathsf{s}\ell\mathsf{T}$ call in sEAL.

boolean type B. So when $A_i = \mathsf{B}$, 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 previous section gives us a weight $\omega(\tau)$ for this proof in $\mathsf{s}\ell\mathsf{T}$. Let us now comment on the definition of $\mu_n(\pi)$. The degrees of $\omega(\tau)$ and I influence 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. Furthermore, when $k = 0$, the term $[t]()$ influences only the weight at position 1, as constructors for base types.

3.3 Example: Testing Satisfiability of a Propositional Formula

Some examples of sEAL terms, like towers of exponentials, can be found in the appendix 6.6. We sketch here the construction of a term for deciding the SAT problem. Some other examples are given in the appendix, like testing the satisfiability of quantified boolean formulas (QBF_k) and deciding the subset-sum problem.

The term for SAT has type $\mathsf{N} \otimes \mathsf{W} \multimap !\mathsf{B}$ and given a formula on conjunctive normal form encoded in the type $\mathsf{N} \otimes \mathsf{W}$, it checks its satisfiability. The modality in front of the output $!\mathsf{B}$ shows that we used a non-polynomial computation, or more precisely an iteration in EAL, as expected of a term for satisfiability.

We encode formula in conjunctive normal form in the type $\mathsf{N} \otimes \mathsf{W}$, 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. Then # and | are used as separator for literals and clauses.

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 $\mathsf{s}\ell\mathsf{T}$. For the sake of simplicity, we sometimes omit to describe all terms in `ifw` or `iterw` , especially for the letters # and |, when they are not important.

First, we can easily define a term $occ_a : \mathsf{W}^I \multimap \mathsf{N}^I$ that gives the number of occurrences of $a \in \Sigma$ in a word. In the appendix 6.3, some important terms are defined. We have a term $Cbinarytounary : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{N}^I$ such that $Cbinarytounary\ \underline{n}\ \underline{w}$ computes the minimum between n and the unary representation of the binary integer w. We also have a term that gives the n^{th} bit (from right) of a binary word as a boolean $n^{th} : \mathsf{W}^I \multimap \mathsf{N}^I \multimap \mathsf{B}$. And finally, we have a term $Extract_a : \mathsf{W}^I \multimap \mathsf{W}^I \otimes \mathsf{W}^I$ 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.

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 $ClausetoBool : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^K \multimap \mathsf{B}$ such that, given the number of variables, a valuation and a word representing a clause, this term outputs the truth value of this clause using the valuation.
$ClausetoBool = \lambda n, w_v, w_c.\ \mathtt{let}\ w \otimes b = \mathtt{itern}(\lambda w' \otimes b'.\mathtt{let}\ w_0 \otimes w_1 = Extract_{\#}\ w'\ \mathtt{in}$
$w_0 \otimes (or\ b'\ (LittoBool\ n\ w_v\ w_1)), w_c \otimes \mathtt{ff})\ (occ_{\#} w_c)\ \mathtt{in}\ b$

With $LittoBool : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{W}^K \multimap \mathsf{B}$ converting a literal into the boolean given by the valuation : $LittoBool = \lambda n, w_v, w_l.\mathtt{ifw}(\lambda w'.n^{th}\ w_v\ (Cbinarytounary\ n\ w'),$
$\lambda w'.not\ (n^{th}\ w_v\ (Cbinarytounary\ n\ w')), \mathtt{ff})\ w_l.$

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

Testing all different valuations. Now all we have to do is to test this term with 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 $\mathsf{s}\ell\mathsf{T}$with base type-inputs in order to check if one valuation satisfies the formula. We use a constructor for iteration defined in the appendix 6.3 : $REC(V, t)\ \underline{n} \to^* V\ \underline{n-1}\ (V\ \underline{n-2}\ (\dots\ (V\ \mathtt{zero}\ t)\dots))$.
We can then give the term for SAT :
$SAT = \lambda n \otimes w.\mathtt{let}\ !r = \mathtt{iter}^!_N(!(\lambda n_0 \otimes n_1.\mathtt{succ}(n_0) \otimes [double](n_1)), !(\underline{0} \otimes \underline{1}))\ n\ \mathtt{in}$
$\mathtt{let}\ !w_f = coerc\ w\ \mathtt{in}\ !(\mathtt{let}\ n \otimes exp = r\ \mathtt{in}\ [\lambda n, exp, w_f.$
$REC(\lambda val, b.or\ b\ (FormulatoBool\ n\ (Cunarytobinary\ n\ val)\ w_f), \mathtt{ff})\ exp](n, exp, w_f)).$

The first iteration computes both 2^n and a copy of n. This technique is important as it shows that the linearity of EAL for base variables is not too constraining for the iteration. In the last line the term is a big "or" on the term $FormulatoBool$ applied to different valuations. And with that we have $SAT : \mathsf{N} \otimes \mathsf{W} \multimap !\mathsf{B}$.

3.4 Subject Reduction and Measure

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 [24]. 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.

Let us first express substitution lemmas for sEAL. There are 3 cases to consider, linear variables and discharged and non-discharged global variables.

▶ **Lemma 15** (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 $\pi' \lhd \Gamma_1, \Gamma_2 \mid \Delta \vdash M[M'/x] : T$. Moreover, for all n, $\mu_n(\pi') \leq \mu_n(\pi) + \mu_n(\sigma)$.*

The proof comes from the fact that rules are multiplicative for Γ, and so x only appears in one of the premises for each rule. Thus the proof σ is used only once in the new proof π'.

▶ **Lemma 16** (General substitution). *If $\pi \lhd \Gamma \mid \Delta, x : T' \vdash M : T$ and $\sigma \lhd \emptyset \mid \Delta \vdash M' : T'$ and the number of occurrences of x in M is less than K, then we have a proof $\pi' \lhd \Gamma \mid \Delta \vdash M[M'/x] : T$. Moreover, for all n, $\mu_n(\pi') \leq \mu_n(\pi) + K \cdot \mu_n(\sigma)$.*

This time, the non-linearity of the variable x induces a duplication of the proof σ, that's why the measure $\mu_n(\sigma)$ is also duplicated.

▶ **Lemma 17** (Discharged substitution lemma). *If $\pi \lhd \Gamma \mid \Delta', [\Delta], x : [T'] \vdash M : T$ and $\sigma \lhd \emptyset \mid \Delta \vdash M' : T'$ then we have a proof $\pi' \lhd \Gamma \mid \Delta', [\Delta] \vdash M[M'/x] : T$. Moreover, for all n, $\mu_n(\pi') \leq (\omega_0(\pi), (\mu_n^1(\pi) + \omega_1(\pi) \cdot \mu_{n-1}(\sigma)))$.*

The proof of this lemma relies directly on the previous one. Indeed, a variable with a discharged type can be used only after crossing a (!-Intro) rule, and then the upper bound on $\mu_n(\pi')$ comes from the previous lemma since the number of occurrences of x in M is less than $\omega_1(\pi)$.

Then, let us give two important definition, t_α and $\mathcal{R}$, in order to derive the upper bound on the number of reduction in sEAL.

▶ **Definition 18** (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 $\alpha \geq 1$ and $x_i \geq 2$ for all i :

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

▶ **Definition 19** ($\mathcal{R}$). We define a relation on vectors denoted $\mathcal{R}$. 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. $\mu \geq \tilde{2}$ and $\mu' \geq \tilde{2}$.
2. $\mu' <_{lex} \mu$. Thus, 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 $d \in \mathbb{N}, 1 \leq d \leq (\omega_{i_0} - \omega'_{i_0})$ such that $\forall j > i_0,\ \omega'_j \leq \omega_j \cdot (\omega_{i_0+1})^{d-1}$

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. We can then connect those two definitions :

▶ **Theorem 20.** *Let $\mu, \mu' \in \mathbb{N}^{n+1}$ and $\alpha \geq n$, $\alpha \geq 1$. If $\mathcal{R}(\mu, \mu')$ then $t_\alpha(\mu') < t_\alpha(\mu)$*

It 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 $\mathcal{R}$.

We can now state the subject reduction of sEAL and we show that the measure allows us to construct a bound on the number of reductions.

▶ **Theorem 21.** *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 uses the substitution lemma for reductions in which substitution appears, and for the others constructors, one can see that the measure given in the type system for sEAL is following this idea of the relation $\mathcal{R}$, e.g., in the reduction $[\lambda x_n \ldots x_1.t](M_1, \ldots, M_{n-1}, \underline{v}) \to [\lambda x_{n-1} \ldots x_1.t[\underline{v}/x_n]](M_1, \ldots, M_{n-1})$, the degree that appears at position 0 is here to compensate the growing of the measure at position 1. Now using the previous results, we can easily conclude our bound on the number of reductions.

▶ **Theorem 22.** *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.*

4 Complexity Results: Characterization of 2k-EXP and 2k-FEXP

Now that we have proved the preceding 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 $k \geq 0$, and each modality "!" in the type of a term will induce a difference of 2 in the height of the tower of exponentials. With exactly the same method, we also have a characterization of the classes $2k$-FEXP for $k \geq 0$.

Restricted Reductions and Values. First, we show that the precedent bound on the number of reductions in Theorem 22 can be improved. Indeed, if we restrict the possible reductions, we obtain a more precise bound.

▶ **Definition 23** (Reductions up to a Certain Depth)**.** For $i \in \mathbb{N}$, we define the i-reductions, that we note $\to_i$:

- $\forall i \geq 1, [t]() \to_i [t']()$ if $t \to t'$ in sℓT. Moreover, $[\underline{v}]() \to_i \underline{v}$
- 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 21 and 22.

▶ **Lemma 24.** *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 $\mathcal{R}(\mu_i(\tau) + \tilde{2}, \mu_i(\tau') + \tilde{2})$*

▶ **Theorem 25.** *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.*

▶ **Definition 26** (Values Associated to Restricted Reductions)**.** We 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 :

$V^0 := M$

$\forall i \geq 1, V^i := \lambda x.M \mid !V^{i-1} \mid V_0^i \otimes V_1^i \mid \mathtt{zero} \mid \mathtt{succ}(V^i) \mid \mathtt{ifn}(V_0^i, V_1^i) \mid \mathtt{iter}^!_N(V_0^i, V_1^i) \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(V_0^i, V_1^i) \mid \epsilon \mid \mathtt{s}_i(V^i) \mid \mathtt{ifw}(V_0^i, V_1^i, V_2^i) \mid \mathtt{iter}^!_W(V_0^i, V_1^i, V_2^i)$.

We can then prove the following lemma:

▶ **Lemma 27.** *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.*

The proof can be found in [7]. 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 an i-value.

A Characterization of $2k$-EXP. Now, we sketch how the type $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$ can characterize the class $2k$-EXP for $k \geq 0$. 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, where p is a polynomial. First we show that the number of reductions for such a term is bounded by a tower of exponentials of height $2k$.

▶ **Lemma 28.** *Let $\pi \lhd \cdot \mid \cdot \vdash t : !\mathsf{W} \multimap !^{k+1}\mathsf{B}$. 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 $!^{k+1}\mathsf{B}$, and a $(k+2)$-value of type $!^{k+1}\mathsf{B}$ is exactly of the form $!^{k+1}\mathtt{tt}$ or $!^{k+1}\mathtt{ff}$. So it is enough to only consider $(k+2)$-reductions to compute the result, by lemma 27. 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 25, 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 28.
Now we have to prove that we can simulate a Turing-machine in our calculus. This proof is usual in implicit complexity [5, 2]. A sketch of this proof can be found in the appendix, section 6.7. With this, using the lemma 28, we obtain the following theorem

▶ **Theorem 29.** *Terms of type $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$ characterize the class $2k$-EXP.*

As explained previously, 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 [5]. This characterization uses the same proof by replacing $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$ by $!\mathsf{W} \multimap !^{k+1}\mathsf{W}$.

Moreover, in EAL, we can characterize k-EXP with the type $!\mathsf{W} \multimap !^{k+1}\mathsf{B}$. The difference with sEAL can be explained by the fact that in EAL, in the type $\mathsf{N} \multimap \mathsf{N}$ we only have polynomials of degree 1 (polynomials in general have the type $!\mathsf{N} \multimap !\mathsf{N}$), whereas in our case, polynomials have the type $\mathsf{N} \multimap \mathsf{N}$.

5 Conclusion

We believe that our main contribution is to define a new methodology to combine size-based and level-based type systems, which we have illustrated here with the example of $\mathsf{s}\ell\mathsf{T}$ and EAL, but we think is of more general interest. In the present particular setting of sEAL we can wonder which enrichment we can add to EAL while keeping the properties, for instance: new data-types (lists, trees), the possibility to freely duplicate base types ... We should also investigate type inference techniques, by drawing inspiration from linear dependent types [11, 3] and EAL [8]. But more importantly we would like to explore to which other systems we could apply this methodology:

- First can we define a similar system in which we could move up one level of ! and stay in polynomial time? We conjecture that this could be obtained with EAL but replacing $\mathsf{s}\ell\mathsf{T}$ with a system of indexes of degree at most 1, instead of polynomial indexes. In this case we believe that the type $!\mathsf{W} \multimap !!\mathsf{B}$ would correspond to PTIME. An alternative choice could be to use a Non-size-increasing types system [19] instead of $\mathsf{s}\ell\mathsf{T}$.
- Can we define a system in which all levels stay in FPTIME? Beside the condition on indexes (degree at most 1) we would also need for that purpose to replace EAL with another level-based system. Light linear logic [15] is a natural candidate, but we would need to find a measure-based argument for its complexity bound, which is a challenging objective.

References

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

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

3 Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit computational complexity of subrecursive definitions and applications to cryptographic proofs (long version). Research report, ENS Lyon, 2015. URL: `https://hal.archives-ouvertes.fr/hal-01197456`.

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

5 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.

6 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.

7 Patrick Baillot and Alexis Ghyselen. Combining linear logic and size types for implicit complexity (long version). hal-01687224, [Research Report], 2018.

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, Jean-Yves Marion, and Jean-Yves 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 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'11, Proceedings*. ACM, 2011.

17 Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *19th Euro. Symp. on Prog.(ESOP'10)*, pages 287–306. Springer, 2010.

18 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.

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

20 Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'11, Proceedings*, pages 185–197. ACM, 2003.

21 John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 410–423, 1996.

22 Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, 2010*, pages 223–236, 2010.

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

24 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.

25 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.

26 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.

6 Appendix

6.1 Type System for Words and boolean in sℓT

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

$$\pi\lhd \frac{\sigma \lhd \Gamma \vdash t : \mathsf{W}^J \qquad J+1 \leq I}{\Gamma \vdash \mathtt{s}_i(t) : \mathsf{W}^I} \qquad \omega(\pi) = \omega(\sigma)$$

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

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

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

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

6.2 Some Intermediate Lemmas for the Subject Reduction

Index Variable Substitution and Subtyping. We give some intermediate lemmas in order to prove the subject reduction theorem. Some intuition and more detailed proofs can be found in the technical report [7]

▶ **Lemma 30** (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 31** (Index substitution). *Let I be an index.*

1. *Let J_1, J_2 be indexes such that $J_1 \leq J_2$ then $J_1[I/a] \leq J_2[I/a]$*
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. *$\omega(\pi[I/a]) = \omega(\pi)[I/a]$*

▶ **Lemma 32** (Monotonic index substitution). *Take J_1, J_2 such that $J_1 \leq J_2$.*
1. *Let I be an index, then $I[J_1/a] \leq I[J_2/a]$.*
2. *For any proof π, $\omega(\pi[J_1/a]) \leq \omega(\pi[J_2/a])$.*
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 $E[J_2/a] \sqsubset E[J_1/a]$*

▶ **Lemma 33.** *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, $\omega(\pi') \leq 1$.*

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

▶ **Lemma 34** (Value Substitution). *If $\pi \lhd \Gamma_1, d\Gamma, x : D' \vdash t : D$ and $\sigma \lhd \Gamma_2, d\Gamma \vdash V : D'$ then we have a proof $\pi' \lhd \Gamma_1, \Gamma_2, d\Gamma \vdash t[V/x] : D$. Moreover, if D' is a base type then $\omega(\pi') \leq \omega(\pi)$. Otherwise, $\omega(\pi') \leq \omega(\pi) + \omega(\sigma)$.*

This is proved by induction on π. For the base type case, we use lemma 33 to show that Γ_2 can be ignored, and then as $d\Gamma$ is duplicable, the proof is rather direct. For the non-base case, 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.

6.3 Examples in sℓT

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 $rev = \texttt{iterw}(\lambda w.\mathtt{s}_0(w), \lambda w.\mathtt{s}_1(w), \epsilon) : \mathsf{W}^I \multimap \mathsf{W}^I$.

Now we define $ITERW(V_0, V_1, t) = \lambda w.(\texttt{iterw}(V_0, V_1, t)(rev\ w)$ that is the iterator on words with the right order $(ITERW(V_0, V_1, t)\ \mathtt{s}_{i_1}(\mathtt{s}_{i_2}(\dots \mathtt{s}_{i_n}(\epsilon)\dots)) \to^* V_{i_1}(V_{i_2}(\dots V_{i_n}(t)\dots))$. 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 $REC(V, t)\ \underline{n} \to^* V\ \underline{n-1}\ (V\ \underline{n-2}\ (\dots\ (V\ \texttt{zero}\ t)\dots))$.
$REC(V, t) = \lambda n.\texttt{let}\ x \otimes y = (\texttt{itern}(\lambda r \otimes n'.(Vn'r) \otimes \texttt{succ}(n'), t \otimes \texttt{zero})\ n)\ \texttt{in}\ x$

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. This constructor can also be defined for words.

Addition for unary words. The addition can be written in sℓT. We give a sketch of the proof tree.

$$\pi_{add}(I,J) \lhd \dfrac{\dfrac{\dfrac{\dfrac{\mathsf{N}^{I+a} \sqsubset \mathsf{N}^{I+a}}{x : \mathsf{N}^I, y : \mathsf{N}^{I+a} \vdash y : \mathsf{N}^{I+a}} \quad I+a+1 \leq I+a+1}{x : \mathsf{N}^I, y : \mathsf{N}^{I+a} \vdash \texttt{succ}(y) : \mathsf{N}^{I+a+1}}}{x : \mathsf{N}^I \vdash \lambda y.\texttt{succ}(y) : \mathsf{N}^{I+a} \multimap \mathsf{N}^{I+a}[a+1/a]} \quad \dots}{x : \mathsf{N}^I \vdash \texttt{itern}(\lambda y.\texttt{succ}(y), x) : \mathsf{N}^J \multimap \mathsf{N}^{I+J}}}{\cdot \vdash \lambda x.\texttt{itern}(\lambda y.\texttt{succ}(y), x) : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I+J}}$$

$add = \lambda x.\texttt{itern}(\lambda y.\texttt{succ}(y), x)$, $\pi_{add}(I, J) \lhd \cdot \vdash add : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^{I+J}$. And the rules give us, for two integers n and m, $add\ \underline{n}\ \underline{m} \to \texttt{itern}(\lambda y.\texttt{succ}(y), \underline{n})\ \underline{m} \to^* (\lambda y.\texttt{succ}(y))^m\ \underline{n} \to^* \underline{n+m}$. The weight of this term is $\omega_{add}(I, J) = 1+J+1+J\cdot(1+1)[J/a] = 3J+2$

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 $\mathsf{s}\ell\mathsf{T}$ with a control on the number of bits. More precisely, we give a term $Cadd : \mathsf{N}^I \multimap \mathsf{W}^{J_1} \multimap \mathsf{W}^{J_2} \multimap \mathsf{W}^I$ 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. The term follows the usual idea for addition: the result is computed bit by bit, and we 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.
$Cadd = \lambda n, w_1, w_2.\ \mathtt{let}\ c' \otimes r' \otimes w_1' \otimes w_2' = \mathtt{itern}(\lambda c \otimes r \otimes w \otimes w'.\ match\ c, w, w'\ with$
$(\mathtt{ff}, \epsilon, \epsilon) \mapsto \mathtt{ff} \otimes \mathtt{s}_0(r) \otimes \epsilon \otimes \epsilon \mid \ldots \mid (\mathtt{tt}, \mathtt{s}_1(v), \mathtt{s}_1(v')) \mapsto \mathtt{tt} \otimes \mathtt{s}_1(r) \otimes v \otimes v', \mathtt{ff} \otimes \epsilon \otimes (rev\ w_1) \otimes (rev\ w_2))\ n\ \mathtt{in}\ r.$

For the typing of this term, we use in the iteration the type $\mathsf{B} \otimes \mathsf{W}^a \otimes \mathsf{W}^{J_1} \otimes \mathsf{W}^{J_2}$, 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 $Cunarytobinary : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{W}^I$ such that on the input n, n', this term computes the least n significant bit of the representation of n' in binary : $Cunarytobinary = \lambda n.\mathtt{itern}(\lambda w.Cadd\ n\ w\ (\mathtt{s}_1(\epsilon)), Cadd\ n\ \epsilon\ \epsilon)$

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 $\mathsf{s}\ell\mathsf{T}$. Nevertheless, given an additional information bounding the size of this unary word, we can give a term $Cbinarytounary : \mathsf{N}^I \multimap \mathsf{W}^J \multimap \mathsf{N}^I$ 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 $min : \mathsf{N}^I \multimap \mathsf{N}^J \multimap \mathsf{N}^I$. $min = \lambda n, n'.\mathtt{let}\ r_0 \otimes n_0 = (\mathtt{itern}(\lambda r_1 \otimes n_1.\mathtt{ifn}(\lambda p.\mathtt{succ}(r_1) \otimes p, r_1 \otimes \mathtt{zero})\ n_1, \mathtt{zero} \otimes n')\ n)\ \mathtt{in}\ r_0$

In order to type this term, we use in the iteration the type $\mathsf{N}^a \otimes \mathsf{N}^J$. Remark that this term allows us to erase the index J. Now that we have this term, we can define the following term $Cbinarytounary = \lambda n.\mathtt{iterw}(\lambda n'.min\ n\ (mult\ n'\ \underline{2}), \lambda n'.min\ n\ \mathtt{succ}(mult\ n'\ \underline{2}), \mathtt{zero})$

Some examples on words. We can define a term that gives the n^{th} bit (from right) of a binary word as a boolean :
$n^{th} = \lambda w, n.\mathtt{ifw}(\lambda w'.\mathtt{ff}, \lambda w'.\mathtt{tt}, \mathtt{ff})\ ((\mathtt{itern}(pred, rev\ w))\ n) : \mathsf{W}^I \multimap \mathsf{N}^I \multimap \mathsf{B}$, with
$pred : \mathsf{W}^I \multimap \mathsf{W}^I = \mathtt{ifw}(\lambda w.w, \lambda w.w, \epsilon)$.

We can also define a term of type $\mathsf{W}^I \multimap \mathsf{W}^I \otimes \mathsf{W}^I$ 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.
$Extract_a = \lambda w.\mathtt{let}\ b' \otimes w_0' \otimes w_1' = ITERW(V_0, V_1, V_\#, V_|, V_\epsilon)\ w\ \mathtt{in}\ w_0' \otimes w_1'$
with $V_a = \lambda b \otimes w_0 \otimes w_1.\mathtt{if}(\mathtt{tt} \otimes s_a(w_0) \otimes w_1, \mathtt{tt} \otimes w_0 \otimes w_1)\ b$
$\forall c \neq a, V_c = \lambda b \otimes w_0 \otimes w_1.\mathtt{if}(\mathtt{tt} \otimes s_c(w_0) \otimes w_1, \mathtt{ff} \otimes w_0 \otimes s_c(w_1))\ b$
$V_\epsilon = ff \otimes \epsilon \otimes \epsilon$

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.

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 $\mathsf{B}^n = \mathsf{B} \otimes \mathsf{B}^{n-1}$ with $\mathsf{B}^1 = \mathsf{B}$. 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 $\texttt{let } x_D \otimes y_{D'} = t \texttt{ in } t'$ 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 `tt` or `ff`. Moreover, there is a constructor to do a match-case on those states, $case_n(t_{0^n}, \dots, t_{1^n})$. 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 `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 $n \geq 0$:
$case_{n+1}(t_{0^{n+1}}, \dots, t_{1^{n+1}}) = \lambda s.\texttt{let } s'_{\mathsf{B}^n} \otimes x_{\mathsf{B}} = s \texttt{ in } case_n(t'_{0^n}, \dots, t'_{1^n})\ s'$ with, for all boolean word i, $t'_i = \texttt{if}(t_{i1}, t_{i0})\ x$.

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:

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

6.4 Adding Polynomial Time Functions in EAL

Here we explain very informally how we can add polynomial time functions in the calculus defined in [24], 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 $f\ \underline{n} \to \underline{f(n)}$, 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 $\mathsf{N} \multimap \mathsf{N}$.

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 $f(n) \leq (n+2)^d$. 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.

6.5 Type System for Words and Boolean in sEAL

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

$$\pi\lhd \frac{\sigma \lhd \Gamma \mid \Delta \vdash M : \mathsf{W}}{\Gamma \mid \Delta \vdash \mathtt{s}_i(M) : \mathsf{W}} \qquad \mu_n(\pi) = \mu_n(\sigma) + \mathbb{1}_1$$

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

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

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

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

6.6 Examples in sEAL

We give some examples of terms in sEAL, 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 sEAL.

- For base types A we have the coercion $A \multimap !A$. For example, for words, this is given by the term $coerc_w = \mathtt{iter}^!_W(!(\lambda w'.\mathtt{s}_0(w')), !(\lambda w'.\mathtt{s}_1(w')), !\epsilon)$, with $coerc_w\ \underline{w} \to^* !\underline{w}$
- We write $\lambda x \otimes y.M$ for the term $\lambda c.\mathtt{let}\ x \otimes y = c\ \mathtt{in}\ M$.

Polynomials and Tower of Exponentials in sEAL Recall that we defined polynomials in $\mathsf{s}\ell\mathsf{T}$. With this we can define polynomials in EAL with type $\mathsf{N} \multimap \mathsf{N}$ 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 sEAL with type $\mathsf{N} \multimap !\mathsf{N}$

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

With $\mathtt{iter}^!_N(!\lambda n.[\lambda x_1.mult\ x_1\ x_1](n), !\underline{2})\ \underline{k} \to^* !((\lambda n.[\lambda x_1.mult\ x_1\ x_1](n))^k\ \underline{2}) \to^* !(2^{2^k})$.

For an example of measure, for the subproof $\pi \lhd \cdot \mid \cdot \vdash \lambda n.[\lambda x_1.mult\ x_1\ x_1](n) : \mathsf{N} \multimap \mathsf{N}$, we have $depth(\pi) = 1$ and as the weight for $\sigma \lhd x_1 : \mathsf{N}^{a_1} \vdash_{\mathsf{s}\ell\mathsf{T}} mult\ x_1\ x_1 : \mathsf{N}^{a_1 \cdot a_1}$ is $\omega(\sigma) = 4 + a_1 + 3a_1^3$, we can deduce $\mu(\pi) = (1 + 1 + 1 \cdot (d(\omega(\sigma) + a_1 \cdot a_1) + 1), 1 + (\omega(\sigma) + a_1 \cdot a_1)[1/a_1]) = (6, 10)$

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 $k \geq 0$ and polynomial P with a term of type $\mathsf{N} \multimap !^k\mathsf{N}$.

Some other big examples, such as QBF_k and the $SUBSET\_SUM$ problem can be found in the technical report [7]

6.7 Simulation of a Turing Machine in sEAL

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 technical report [7].

Suppose given two variables $w : \mathsf{W}^{a_w}$ and $n : \mathsf{N}^{a_n}$, we note $Conf_b$ the type $\mathsf{W}^{a_w+b} \otimes \mathsf{B} \otimes \mathsf{W}^{a_w+b} \otimes \mathsf{B}^q$, 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 $w : \mathsf{W}^{a_w}, n : \mathsf{N}^{a_n} \vdash init : Conf_1$ 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 $\cdot \vdash final : Conf_b \multimap \mathsf{B}$ 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. The height of the tower of exponential we can compute in this calculus is closely linked to the difference of ! modalities between the input and the output. You can see this with the examples in the appendix 6.6. 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.