Combining Linear Logic and Size Types for Implicit Complexity

Patrick Baillot[1] **and Alexis Ghyselen**[2]

1 **Univ Lyon, CNRS, ENS de Lyon, Université Claude-Bernard Lyon 1, LIP, F-69342, Lyon Cedex 07, France**
patrick.baillot@ens-lyon.fr

2 **ENS Paris-Saclay, France**
ghyselen.alexis@gmail.com

Abstract

Several type systems have been proposed to statically control the time complexity of λ-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 limitations. Our system is based on elementary linear logic combined with linear size types and leads to characterizations of the complexity classes FPTIME and $2k$-FEXPTIME, for $k \geq 0$.

1998 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Linear logic, type systems, polynomial time complexity, size types

Digital Object Identifier 10.4230/LIPIcs.DICE.2018.23

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 [7]. This work on safe recursion leads to languages for polynomial time [12], for oracle functionals or for probabilistic computation [9, 17]. 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 [11] and elementary linear logic (ELL) for the classes k-FEXPTIME, for $k \geq 0$. [11, 2, 10]. 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 [13] : 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 [8, 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 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 [5] 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 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.

2 Elementary Affine Logic with Sizes

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 [5] 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 is 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 [16].

2.1 Elementary Linear Logic

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). We call elementary affine logic (EAL) the logic ELL with weakening. Basically, if we consider in EAL 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]. More precisely, we can characterize the class k-EXPTIME with terms of type $!\mathsf{W} \multimap !^{k+2}\mathsf{B}$ (where B is a type for booleans). The paper [16] gives a reformulation of the principles of EAL in an extended lambda-calculus with constructions for !. It also incorporates other features (references and multithreading) which we are not interested in here. In [16], a proof of termination in elementary time of this λ-calculus is given. The proof relies on the description of a function associating to each term a measure that strictly decreases for any reduction step. This function depends on a primordial notion in elementary linear logic: the *depth*, which is closely linked with the use of the "!" modality in terms. The advantage of this proof is robustness: by giving a large upper-bound on the number of reduction steps for a term, we obtain a local proof that does not rely on a particular strategy. This makes it easier to adapt this proof to extended version of the calculus.

2.2 Our Language : sEAL

Here, we define informally our language, denoted sEAL for Elementary Affine Logic with sizes. The idea behind sEAL is to enrich the elementary lambda-calculus by a kind of *bootstrapping*, consisting in adding more terms to the "basic" type $\mathsf{W} \multimap \mathsf{W}$. We first observed that the proof in [16] is robust enough to support an extension of EAL that consists in adding polynomial functions in this type. Following this observation, we plan to give this type enough terms for representing all polynomial time functions. The way we implement this 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 [15]. 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 [14, 1] and we call it $\mathsf{s}\ell\mathsf{T}$. The polynomial bound on this calculus is obtained by restricting the size expressions to polynomial expressions. The language $\mathsf{s}\ell\mathsf{T}$ is used for tuning first-order intensional expressivity: any first order term in $\mathsf{s}\ell\mathsf{T}$ can be represented in sEAL. Then, constructors from EAL can be used in sEAL, this allows us to deal with higher-order computation, non-linear use of functional arguments and non-polynomial iterations.

As for elementary linear logic, we can characterize some complexity classes in sEAL: for

$k \geq 0$, the class $2k$-FEXPTIME is characterized by the terms of type $!\mathsf{W} \multimap !^{k+1}\mathsf{W}$. The difference of computational power between sEAL and EAL is explained by the fact that in EAL, in the type $\mathsf{N} \multimap \mathsf{N}$ (N being the type for integers), we have only polynomials of degree 1, whereas in sEAL, we have all polynomials. This makes the iteration in sEAL more powerful that the usual one in EAL.

To show that this calculus has a better intensional expressivity and is more natural than the usual EAL, we work on usual problems that could be written in sEAL. In particular, we can define terms of type $\mathsf{W} \multimap !\mathsf{B}$ solving the SAT and the SUBSET-SUM problems. As expected, the type $\mathsf{W} \multimap !\mathsf{B}$ shows that we need a non-polynomial iteration (here corresponding to the exhaustive search) with intermediate polynomial functions defined in $\mathsf{s}\ell\mathsf{T}$. In the same way, we can also define a term solving QBF_k with type $\mathsf{W} \multimap !\mathsf{B}$.

Now that we detailed the results obtained with sEAL, let us describe more formally how the link between $\mathsf{s}\ell\mathsf{T}$ and EAL is performed in sEAL.

3 Rule for the sℓT Call

We present here a simplified version of a constructor in sEAL called the $\mathsf{s}\ell\mathsf{T}$-call, and we give a typing rule for this constructor with an integer input. We do not give the complete definition of $\mathsf{s}\ell\mathsf{T}$ and sEAL, but it can be found in the technical report [6].

▶ **Definition 1** (The $\mathsf{s}\ell\mathsf{T}$ call)**.** We note $(M, M', \dots)$ terms of sEAL, and we note $(t, t', \dots)$ terms of $\mathsf{s}\ell\mathsf{T}$. The $\mathsf{s}\ell\mathsf{T}$-call is a constructor in sEAL with the form $[\lambda x.t](M)$

Intuitively, $[\lambda x.t](M)$ represents the application of the function described by $\lambda x.t$ to M. For example, if $M \to^* \underline{n}$ with $\underline{n}$ encoding the integer n, and $t[x := \underline{n}] \to^* \underline{m}$ then we have $[\lambda x.t](M) \to^* \underline{m}$.

Now, for the typing, let us give some notations for types.

▶ **Definition 2** (Types)**.** The base type for integers is denoted by N in sEAL. In $\mathsf{s}\ell\mathsf{T}$, as explained previously, types are annotated with size expressions. We note I, J and we call *indexes* those kind of expressions. Indexes contain also variables, that we call *index variables*, and usually note $a, b, \dots$ And so, in $\mathsf{s}\ell\mathsf{T}$, integers have a type N^I.

The idea behind the typing rule for the $\mathsf{s}\ell\mathsf{T}$-call is to use those index variables to denote the (unknown) size of the term M, and then the typing in $\mathsf{s}\ell\mathsf{T}$ gives us a bound on the size of the output for the $\mathsf{s}\ell\mathsf{T}$-call.

Formally, this is expressed by this rule :

$$\frac{\cdot \vdash_{sEAL} M : \mathsf{N} \qquad x : \mathsf{N}^a \vdash_{\mathsf{s}\ell\mathsf{T}} t : \mathsf{N}^I}{\cdot \vdash_{sEAL} [\lambda x.t](M) : \mathsf{N}}$$

$\vdash_{sEAL}$ denotes type derivations in $sEAL$ and $\vdash_{\mathsf{s}\ell\mathsf{T}}$ denotes type derivations in $\mathsf{s}\ell\mathsf{T}$. In our work, this rule is generalized to any first order function $\lambda x_1, \dots, x_n.t$, but for the sake of simplicity, we present here only the case when $\lambda x.t : \mathsf{N} \multimap \mathsf{N}$. A difficulty in the computation of the upper bound on the number of reductions in $sEAL$ is to define a measure for this rule. This is done by first working on an upper-bound for typed terms in $\mathsf{s}\ell\mathsf{T}$(that depends on index variables), and then modifying the usual notion of depth in EAL to get rid of the index variables, still following the main idea behind [16].

4 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 believe is of more general interest. Can we define a similar system in which we could move up one level of ! and stay in polynomial time? Can we define a system in which all levels stay in FPTIME? Beside another condition on indexes, we would also need for that purpose to replace EAL with another level-based system. Light linear logic [11] would be a natural candidate, but we would need to find a measure-based argument for its complexity bound in order to apply the toolbox of the present work, 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. *Inf. Comput.*, 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). ENS Lyon, 2015.
4 Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Inf. Comput.*, 248:56–81, 2016.
5 Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *ESOP*, pages 104–124. Springer, 2010.
6 Patrick Baillot and Alexis Ghyselen. Combining linear logic and size types for implicit complexity (long version). hal-01687224, [Research Report], 2018.
7 Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. In *Proc. of the 24th annual ACM STOC*, pages 283–293. ACM, 1992.
8 Guillaume Bonfante, J-Y Marion, and J-Y Moyen. Quasi-interpretations a way to control resources. *TCS*, 412(25):2776–2796, 2011.
9 Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In *FOPARA*, pages 1–18. Springer, 2011.
10 Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Inf. Comput.*, 183(1):123–137, 2003.
11 Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.
12 Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *CSL*, pages 275–294. Springer, 1997.
13 Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003.
14 John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT*, pages 410–423, 1996.
15 Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *LMCS*, 8(4), 2011.
16 Antoine Madet and Roberto M Amadio. An elementary affine λ-calculus with multithreading and side effects. In *TLCA*, pages 138–152. Springer, 2011.
17 John Mitchell, Mark Mitchell, and Andre Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *FOCS*, pages 725–733, 1998.