Logarithmic space queries and regular transductions in the elementary affine λ-calculus

NGUYỄN Lê Thành Dũng
LIPN, Université Paris 13, France
nltd@nguyentito.eu

The elementary affine λ-calculus was introduced as a polyvalent setting for implicit computational complexity, allowing for characterizations of polynomial time and hyperexponential time predicates. We show that interesting perspectives for sub-polynomial complexity arise when we remove a certain feature (namely, recursive types) from the type system of this programming language.

Using Church-encoded strings, we capture regular languages, and of two different classes of transductions (functions from strings to strings which admit automata-theoretic definitions). As far as we know, this is the first time transduction classes are characterized in a "purely logical" functional calculus (concretely, in a programming language without a primitive data type for strings or lists).

Alternatively, we may choose to represent inputs as finite relational structures (finite models), taking inspiration from the work of Hillebrand et al. in the 90's on expressing database queries in the λ-calculus. Following this train of thought, we obtain what we believe to be a characterization of deterministic logarithmic space. So far, in a joint work with Pierre Pradic (accepted for presentation at ICALP 2019), we have managed to prove some encouraging partial results towards this conjecture.

1 Affine types for implicit computational complexity

The starting point of our work is the elementary affine λ-calculus [4], a typed functional programming language whose purpose is to characterize complexity classes. (This is the "type-theoretic" or "Curry-Howard" approach to implicit complexity.) It has the advantage of allowing multiple different classes to be captured in the same language, by merely changing the type of programs:

Theorem 1.1 (Baillot, De Benedetti and Ronchi [4])**.** *The programs of type* $!\texttt{Str} \multimap !^{k+2}\texttt{Bool}$ *in the elementary affine* λ*-calculus decide exactly the languages in the class* k-EXPTIME.

In particular, $!\texttt{Str} \multimap !!\texttt{Bool}$ *corresponds to* polynomial time (P).

This is a simplification of an earlier result [3] taking place in a variant of Elementary Linear Logic[1]. The latter was originally introduced by Girard [9] in order to characterize the elementary recursive functions. Here are some indications for the reader unfamiliar with linear or affine type systems:

- a program of type $A \multimap B$ uses its input of type A at most once to produce its output of type B;
- $!A$ means roughly "as many A's as you want", so a function which uses its argument multiple times can be given a type of the form $!A \multimap B$;
- in usual linear or affine logic, one can convert a $!A$ into a A; however, in the elementary affine λ-calculus, there is a restriction which makes the *exponential depth* (number of '!' modalities) meaningful, one cannot perform such a depth-changing operation – this is why the depth k of the output $!^k\texttt{Bool}$ (i.e. $!\ldots!\texttt{Bool}$ with k '!') controls the complexity;

[1] Baillot's original result [3] used Second-order Intuitionistic Elementary Affine Logic with least fixpoints. The elementary affine λ-calculus is a Curry-Howard counterpart of this logic as a typed λ-calculus; it simplifies it in some ways, and adds some tweaks to ensure desirable properties from a programming language point of view, such as subject reduction.

Submitted to:
LCC 2019

- $\mathtt{Str} = \forall\alpha.\mathtt{Str}[\alpha]$, with $\mathtt{Str}[\alpha] = !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$, is the type of Church encodings of binary strings: the string $w_1 \dots w_n$ is represented as the function which, for any type A takes $f_0 : A \multimap A$, $f_1 : A \multimap A$ and returns $f_{w_1} \circ \dots \circ f_{w_n}$.

The proof of Theorem 1.1 relies on the presence of *recursive types* in the type system to guarantee expressivity. A typical example is the type of Scott integers[2] $\mathtt{Nat}_S := \forall\alpha.\, \alpha \multimap (\mathtt{Nat}_S \multimap \alpha) \multimap \alpha$: in the elementary affine λ-calculus as defined in [4], this equation is a valid type definition[3].

The starting point of our story is to ask whether recursive types really are needed for the above theorem – if the goal is merely to characterize elementary recursive functions, then it is not the case. It turns out that they are indeed necessary for the characterization of P. Indeed, let us call EAλ the elementary affine λ-calculus *without recursive types*; then:

Theorem 1.2. *The* EAλ*-terms (i.e programs in* EAλ*) of type* $!\mathtt{Str} \multimap !!\mathtt{Bool}$ *decide exactly the* regular languages. *This is also the case for the* EAλ*-terms of type* $\mathtt{Str} \multimap !\mathtt{Bool}$.

We recently proved this theorem (or rather the analogous statement for Elementary Linear Logic) in [17]. The proof techniques are quite different from those used in [4]. Instead, we take inspiration from the tradition of implicit complexity in the simply typed λ-calculus, in particular from:

Theorem 1.3 (Hillebrand & Kanellakis [13])**.** *Let A is a simple type and t be a term in the* simply typed λ-calculus *of type* $\mathtt{Str}_\lambda[A] \to \mathtt{Bool}_\lambda$ *. Then t decides a* regular language. *Conversely, for every regular language, there exists some type A and some $t : \mathtt{Str}_\lambda[A] \to \mathtt{Bool}_\lambda$ which decides it.*

Here $\mathtt{Str}_\lambda[A] = (A \to A) \to (A \to A) \to (A \to A)$ and $\mathtt{Bool}_\lambda = o \to o \to o$, where o is a base type.

It should be noted at this point that regular languages are not quite a well-behaved complexity class, e.g. they are not closed under AC^0 reductions. The other side of the coin is that Theorem 1.2 hints at a connection between affine typing and automata; and this connection clearly depends on the use of Church encodings, in other words, on the representation of strings by their iterators. This opens up two avenues for investigation:

- One can search for other automata-theoretic classes of interest that can be characterized in EAλ. This is covered in section 2 for functions from strings to strings.
- On the other hand, one can hope to obtain a well-behaved sub-polynomial complexity class by changing the representation of inputs: this is the subject of section 3.

2 Transductions over Church-encoded inputs

Regular functions are a well-known class of transductions whose canonicity is established by their many equivalent defintions. By using the characterization by streaming string transducers [2], we prove:

Theorem 2.1. *The type* $\mathtt{Str} \multimap \mathtt{Str}$ *in* EAλ *captures exactly the* regular functions.

As for the class of functions corresponding to the type $!\mathtt{Str} \multimap !\mathtt{Str}$, it doesn't seem to be some pre-existing widespread class of transductions. However, it admits a simple definition starting from regular functions, and it is a subclass of the *polyregular functions* recently introduced by Bojańczyk [5].

Definition 2.2. Let $f : \Gamma^* \to I^*$, and for $i \in I$, $g_i : \Gamma^* \to \Sigma^*$. The *composition by substitutions* of f with the family $(g_i)_{i\in I}$ is the function $\mathrm{CbS}(f,(g_i)_{i\in I}) : w \mapsto g_{i_1}(w)\dots g_{i_n}(w)$ where $f(w) = i_1 \dots i_k$.

That is, we first apply f to the input, then every letter i in the result of f is substituted by (i.e. expanded into) the image of the original input by g_i. $\mathrm{CbS}(f,(g_i)_{i\in I})$ is thus a function $\Gamma^* \to \Sigma^*$.

[2]The idea is that integers are represented by their "pattern-matching" function (destructor): if n is a Scott integer, then $n\,x\,f$ morally means $\mathtt{case}\ n \mid 0 \mapsto x \mid \mathtt{succ}(m) \mapsto f\,m$.

[3]Formally, one uses a type fixpoint operator to express this as $\mathtt{Nat}_S := \mu\beta.\forall\alpha.\,\alpha \multimap (\beta \multimap \alpha) \multimap \alpha$.

Theorem 2.3. *The type* $!\mathtt{Str} \multimap !\mathtt{Str}$ *in* EAλ *captures exactly the smallest class containing regular functions and closed under both composition by substitutions and usual composition of functions. (To be precise, it contains exactly the functions over binary alphabets in this class.)*

Remark 2.4. Given Theorem 1.3, another interesting question is about the string-to-string functions definable in the simply typed λ-calculus. The functions implemented by $\mathtt{Str}_\lambda[o] \to \mathtt{Str}_\lambda[o]$ have been given various characterizations [21, 15], but don't seem to correspond to any automata-theoretic class.

But for the class of functions $\mathtt{Str}_\lambda[A] \to \mathtt{Str}_\lambda[o]$ where A may vary, not much is known about it, except for a result by Joly [14] relating it to a complexity constraint on the untyped λ-calculus. We have found a large class of transductions that can be expressed as λ-terms of such types: the closure by composition of HDT0L transductions [20, 7]. We believe that this is a first step towards reviving this question, after a few inexpressibility results by Statman (see [8]) suggested that it might not have a satisfying answer.

3 Using finite models as inputs: towards logarithmic space queries

Our main inspiration here is Hillebrand's PhD thesis [11], a junction point between implicit and descriptive complexity. The idea was to represent finite relational structures – a.k.a. finite models – inside the simply typed λ-calculus, using them to represent the inputs to programs. By doing so, Hillebrand et al. managed to characterize P [12], PSPACE [1] and k-EXPTIME/k-EXPSPACE [13] – the extensional completeness for the first two being established through descriptive complexity. (Actually, this idea of representing inputs as finite models already appeared in the early history of implicit complexity: Gurevich [10] showed in 1983 that in this setting, a form of primitive recursion captures *deterministic logarithmic space* (L).)

Definition 3.1. Let us fix a first-order relational signature Σ, i.e. a list of relation symbols $\{\leq, \mathscr{R}_1 \ldots, \mathscr{R}_k\}$ with their respective arities $r_0 = 2, r_1, \ldots, r_k$. A *totally ordered finite model* $\mathfrak{D}$ over Σ consists of a finite set D and an interpretation $\mathscr{R}_i^{\mathfrak{D}} \subseteq D^{r_i}$ for each relation symbol, such that $\leq^{\mathfrak{D}}$ is a total order.

The total ordering assumption is usual in descriptive complexity. We represent a finite model using the following EAλ types, where δ is a type variable corresponding to[4] the type of domain elements:

- $\mathtt{List}[\delta] = \forall\alpha.(!(\delta \multimap \alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha))$ for the list of domain elements
- $\mathtt{Rel}_r[\delta] = \delta \multimap \ldots \multimap \delta \multimap \mathtt{Bool}$ (with r arguments) for r-ary relations
- $\mathtt{C}[\delta] = \delta \multimap \delta \otimes \delta$ to allow duplication of domain elements ($A \otimes B$ is an affine type for pairs[5])

The following is an adaptation to EAλ of a result obtained for Elementary Linear Logic in a joint work with Pierre Pradic [18].

Theorem 3.2. *The class of queries computed by* EAλ *programs of the type below is* between L and NL*:*

$$\forall\delta.(!\mathtt{List}[\delta] \multimap !!\mathtt{Rel}_{r_0}[\delta] \multimap \ldots \multimap !!\mathtt{Rel}_{r_k} \multimap !!\mathtt{C}[\delta] \multimap !!\mathtt{Bool})$$

We have also achieved an exact characterization of L using an ad-hoc restriction of the type system. But we believe this restriction is unneccessary:

Conjecture 3.3. *The class of queries defined in the above theorem is* exactly L.

Although several characterizations of logarithmic space have already been devised in variants of linear logic [19, 6, 16], the above conjecture, if true, would arguably provide the most natural one.

[4] For a concrete finite model of domain size n, we instantiate $\delta = \forall\alpha.\,\alpha^n \multimap \alpha = \forall\alpha.\,\alpha \multimap \ldots \multimap \alpha$ (with n arguments).

[5] Which may be encoded as $A \otimes B = \forall\alpha.\,(A \multimap B \multimap \alpha) \multimap \alpha$.

References

[1] Serge Abiteboul & Gerd Hillebrand (1995): *Space usage in functional query languages*. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Georg Gottlob & Moshe Y. Vardi, editors: *Database Theory — ICDT '95*, 893, Berlin, Heidelberg, pp. 439–454, doi:10.1007/3-540-58907-4_33.

[2] Rajeev Alur & Pavol Černý (2010): *Expressiveness of streaming string transducers*. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, pp. 1–12, doi:10.4230/LIPIcs.FSTTCS.2010.1.

[3] Patrick Baillot (2015): *On the expressivity of elementary linear logic: Characterizing Ptime and an exponential time hierarchy*. *Information and Computation* 241, pp. 3–31, doi:10.1016/j.ic.2014.10.005.

[4] Patrick Baillot, Erika De Benedetti & Simona Ronchi Della Rocca (2018): *Characterizing polynomial and exponential complexity classes in elementary lambda-calculus*. *Information and Computation* 261, pp. 55–77, doi:10.1016/j.ic.2018.05.005.

[5] Mikołaj Bojańczyk (2018): *Polyregular Functions*. *CoRR* abs/1810.08760.

[6] Ugo Dal Lago & Ulrich Schöpp (2016): *Computation by interaction for space-bounded functional programming*. *Information and Computation* 248, pp. 150–194, doi:10.1016/j.ic.2015.04.006.

[7] Emmanuel Filiot & Pierre-Alain Reynier (2017): *Copyful Streaming String Transducers*. In Matthew Hague & Igor Potapov, editors: *Reachability Problems*, 10506, Cham, pp. 75–86, doi:10.1007/978-3-319-67089-8_6.

[8] Steven Fortune, Daniel Leivant & Michael O'Donnell (1983): *The Expressiveness of Simple and Second-Order Type Structures*. *Journal of the ACM* 30(1), pp. 151–185, doi:10.1145/322358.322370.

[9] Jean-Yves Girard (1998): *Light Linear Logic*. *Information and Computation* 143(2), pp. 175–204, doi:10.1006/inco.1998.2700.

[10] Yuri Gurevich (1983): *Algebras of feasible functions*. In: *24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*, Tucson, AZ, USA, pp. 210–214, doi:10.1109/SFCS.1983.5.

[11] Gerd G. Hillebrand (1994): *Finite Model Theory in the Simply Typed Lambda Calculus*. Ph.D. thesis, Brown University, Providence, RI, USA.

[12] Gerd G. Hillebrand & Paris C. Kanellakis (1994): *Functional Database Query Languages As Typed Lambda Calculi of Fixed Order (Extended Abstract)*. In: *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '94, ACM, New York, NY, USA, pp. 222–231, doi:10.1145/182591.182615.

[13] Gerd G. Hillebrand & Paris C. Kanellakis (1996): *On the Expressive Power of Simply Typed and Let-Polymorphic Lambda Calculi*. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, pp. 253–263, doi:10.1109/LICS.1996.561337.

[14] Thierry Joly (2001): *Constant time parallel computations in λ-calculus*. *Theoretical Computer Science* 266(1), pp. 975–985, doi:10.1016/S0304-3975(00)00380-7.

[15] Daniel Leivant (1993): *Functions over free algebras definable in the simply typed lambda calculus*. *Theoretical Computer Science* 121(1), pp. 309–321, doi:10.1016/0304-3975(93)90092-8.

[16] Damiano Mazza (2015): *Simple Parsimonious Types and Logarithmic Space*. In: *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, pp. 24–40, doi:10.4230/LIPIcs.CSL.2015.24.

[17] Lê Thành Dũng Nguyễn (2019): *On the elementary affine λ-calculus with and without type fixpoints*. Available at `https://hal.archives-ouvertes.fr/hal-02153709`. Submitted.

[18] Lê Thành Dũng Nguyễn & Pierre Pradic (2019): *From normal functors to logarithmic space queries*. In: *46th International Colloquium on Automata, Languages and Programming (ICALP'19)*, pp. 123:1–123:15, doi:10.4230/LIPIcs.ICALP.2019.123.

[19] Ulrich Schöpp (2007): *Stratified Bounded Affine Logic for Logarithmic Space*. In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pp. 411–420, doi:10.1109/LICS.2007.45.

[20] Géraud Sénizergues (2007): *Sequences of Level 1, 2, 3,..., k,...* In Volker Diekert, Mikhail V. Volkov & Andrei Voronkov, editors: *Computer Science – Theory and Applications*, 4649, Berlin, Heidelberg, pp. 24–32, doi:10.1007/978-3-540-74510-5_6.

[21] Marek Zaionc (1987): *Word operation definable in the typed λ-calculus*. *Theoretical Computer Science* 52(1), pp. 1–14, doi:10.1016/0304-3975(87)90077-6.