Transductions in affine logic

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. But these results depend on the presence of type fixpoints. Indeed, we showed in previous work that without type fixpoints, we get a characterization of regular languages instead of polynomial time.

This suggests connections between implicit complexity in linear logic and automata theory, which we investigate further here by looking at functions from strings to strings. It turns out that in the elementary affine λ-calculus, one can characterize two different classes of transductions. One of them is the well-known class of regular functions, while the other one seems to be strictly in-between regular functions and Bojańczyk's polyregular functions.

Among the many pre-existing definitions of regular functions, one involves streaming string transducers, a computational model whose definition explicitly involves an affine use of variables. Furthermore, the generalisation of this model from strings to ranked trees involves a relaxed affineness condition reminiscent of the additive conjunction of linear logic.

1 Initial context

Elementary Linear Logic (ELL) is a variant of linear logic designed for implicit computational complexity, a field concerned with the machine-free description of complexity classes. It was first introduced by Girard [8] in order to give a characterization of elementary recursive functions; this was refined by Baillot [3] into a characterization of the k-EXPTIME hierarchy, in an affine variant of ELL. A later improvement by Baillot, De Benedetti and Ronchi [4] was turning this logic into a type system for a functional calculus with good properties (e.g. subject reduction), called the *elementary affine λ-calculus*.

We wish to draw attention to a particular feature of this language: the presence of *type fixpoints*, a.k.a. recursive types (without any kind of "positivity" constraint). This is harmless for the normalization property, as the untyped version of the elementary affine λ-calculus is already normalizing. (The analogous property for ELL was already remarked in [8].) We write EAλ for the elementary affine λ-calculus *without* type fixpoints, and μEAλ for the original system in [4] including recursive types. Both EAλ and μEAλ also support polymorphism (second-order quantification).

Concerning the application of μEAλ to implicit complexity, the main result is:

Theorem 1.1 ([4]). *The programs of type* $!\mathtt{Str} \multimap\ !^{k+2}\mathtt{Bool}$ *in* μEAλ *decide exactly the languages in the class k-*EXPTIME. *In particular* $!\mathtt{Str} \multimap\ !!\mathtt{Bool}$ *corresponds to* polynomial time (P) *predicates.*

Here $\mathtt{Str} = \forall\alpha.\,\mathtt{Str}[\alpha]$, 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 $f_{w_1} \circ \dots \circ f_{w_n}$. $\mathtt{Bool} = \forall\alpha.\,\alpha \multimap \alpha \multimap \alpha$ is the usual second-order encoding of booleans.

The starting point of our story is to ask whether type fixpoints 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:

Submitted to:
TLLA 2019

Theorem 1.2. *The programs of type* $!\mathtt{Str} \multimap !!\mathtt{Bool}$ *in* EAλ *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 in [12]. This result is surprising for a few reasons:

- the class of languages obtained is unexpectedly small;
- the proof techniques are quite different from those used in [4] (instead of bounding the syntactic normalization process, we take inspiration from the tradition of implicit complexity in the simply typed λ-calculus and use *semantic evaluation*, see [14] and references therein);
- most importantly here, this connects EAλ to formal language theory.

That said, there is one main precedent and direct inspiration:

Theorem 1.3 (Hillebrand & Kanellakis [9])**.** *In the* simply typed λ-calculus*, the languages decided by terms of type* $\mathtt{Str}_\lambda[A] \to \mathtt{Bool}_\lambda$ *– A is a simple type that may be chosen depending on the language – are exactly the regular languages.*

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.

2 String functions in the elementary affine λ-calculus

The next thing to do is to look at the functions from strings to strings, both in μEAλ and in EAλ. In the former, the characterization of the k-EXPTIME hierarchy of predicates adapts nicely to the corresponding hierarchy of functions:

Theorem 2.1. *The programs of type* $!\mathtt{Str} \multimap !^{k+1}\mathtt{Str}$ *in* μEAλ *compute exactly the functions in the class* k-FEXPTIME*. In particular* $!\mathtt{Str} \multimap\ !!\mathtt{Bool}$ *corresponds to* polynomial time (FP) *functions.*

Despite being a very natural statement, this theorem does not appear in [4]. We proposed in [12] a short proof building on the results of [4].

Let us now consider the types $!\mathtt{Str} \multimap !\mathtt{Str}$ and $\mathtt{Str} \multimap \mathtt{Str}$ in EAλ. Observe that both of the classes of functions corresponding to those types

- are closed under composition, since the input and output types match;
- are regularity-preserving[1]: the inverse image of a regular language is regular.

This leads us to wonder: are these classes are in fact well-known classes of *transductions*, that is, functions computed by some flavor of automata with output? In the simply typed λ-calculus:

- the functions implemented by terms of type $\mathtt{Str}_\lambda[o] \to \mathtt{Str}_\lambda[o]$ have been given various characterizations [15, 11], but don't seem to correspond to any automata-theoretic class;
- the functions $\mathtt{Str}_\lambda[A] \to \mathtt{Str}_\lambda[o]$ (where A may vary) can have outputs whose length is any fixed tower of exponentials in the input length; they contain at least the closure by composition of the transductions defined by HDT0L systems [13, 7] but it is unclear whether this containment is strict.

(The latter class has also been studied by Joly [10], who provided a characterization in terms of untyped λ-terms subject to a kind of complexity constraint. But this is not at all enlightening as to the possible connections with automata.)

The situation is nicer in EAλ. We very recently obtained the following result:

[1]Proof: Let $f : \mathtt{Str} \multimap \mathtt{Str}$ and L be a regular language, recognized by some $p_L : \mathtt{Str} \multimap !\mathtt{Bool}$ thanks to Theorem 1.2; $p_L \circ f$ has the type $\mathtt{Str} \multimap !\mathtt{Bool}$, so it recognizes a regular language (by the other direction of Theorem 1.2), which is none other than $f^{-1}(L)$ (and the same argument works for $!\mathtt{Str} \multimap !\mathtt{Str}$).

Theorem 2.2. *The type* $\texttt{Str} \multimap \texttt{Str}$ *in* EAλ *captures exactly the* regular functions.

Regular functions are a well-known class of transductions whose canonicity is established by the wide variety of equivalent defintions. For instance, they are computed by deterministic two-way transducers, and also have a characterization in terms of Monadic Second-Order Logic [6].

To show that regular languages can be computed by EAλ-terms, we use the definition in terms of *streaming string transducers*, cf. section 3. The converse is significantly more difficult; it relies on evaluating EAλ-terms in a semantics developed by analyzing the normalization of second-order proof nets (this analysis is an ongoing joint work with P. Pistone, T. Seiller and L. Tortora de Falco; the proofs have not yet been properly written up and thoroughly checked).

As for the class of functions corresponding to the type $!\texttt{Str} \multimap !\texttt{Str}$, it doesn't seem to be some preexisting 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]. (In turn, polyregular functions can be expressed as compositions of HDT0L transductions. Though this is easy to prove, it does not appear in the literature to our knowledge. As a consquence, all polyregular functions are definable in the simply-typed λ-calculus.)

The provisional name for our new class is "polysequential". For the definition, it is most convenient to use strings over arbitrary finite alphabets.

Definition 2.3. 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 $\Gamma^* \to \Sigma^*$.

Definition 2.4. The class of *polysequential functions* is the smallest class containing regular functions and closed under both composition by substitutions and usual composition of functions.

Proposition 2.5. *Regular functions* $\subsetneq$ *polysequential functions* $\subseteq$ *polyregular functions.*

Conjecture 2.6. *The last inclusion above is also strict.*

Theorem 2.7. *The type* $!\texttt{Str} \multimap !\texttt{Str}$ *in* EAλ *captures exactly the polysequential functions (whose input and output alphabets are both binary).*

One may also consider what happens for *binary trees*, represented by their Church encoding

$$\texttt{BT} = \forall\alpha.\,!(\alpha \multimap \alpha \multimap \alpha) \multimap !\alpha \multimap !\alpha$$

We have not investigated yet the functions $!\texttt{BT} \multimap !\texttt{BT}$, but for now, what we know is:

Theorem 2.8. *The type* $\texttt{BT} \multimap \texttt{BT}$ *captures exactly the regular tree functions.*

3 Streaming string transducers

To prove the easy direction of Theorem 2.2, we use a characterization of regular functions proposed by Alur and Černý [2]. It based on one-way automata whose memory contains, in addition to a control state taken in a finite set, a finite number of string-valued registers.

Definition 3.1. A *(deterministic) streaming string transducer* (SST) consists of:

- a finite set Q of *states*, with an initial state $q_I \in Q$
- a finite set X of *variables* (or register names)

- a transition function $\delta : Q \times \{0,1\} \to Q \times ((\{0,1\}+X)^*)^X$
- an output function $F : Q \to (\{0,1\}+X)^*$

such that the *copyless assignment* condition holds: for all $q \in Q$, $x \in X$,

- $F(q)$ contains x at most once
- for all $a \in \{0,1\}$, $(q',u) \in \delta(q,a)$, x appears at most once among all the words $u(y)$ for $y \in X$

The term "copyless assignment", taken from the literature on automata theory, suggests some kind of affine typing condition on strings. Indeed, the idea is as follows: at each time step, depending on the letter that has been read, the new values of the registers are obtained from the old ones and from string constants by concatenations; the copyless assignement condition says that each for each register, its old value is used at most once in the computation of the new set of values. It is exactly this affineness that allows us to encode SSTs as EAλ-terms.

Remark 3.2. One may wonder what happens if this affineness condition is relaxed. Filiot and Reynier [7] proved that the functions computed by the so-called *copyful streaming string transducers* are the HDT0L transductions.

Formally, the semantics of a SST are defined as follows:

Definition 3.3. A configuration of a SST is a pair (q,s) with $q \in Q$ and $s : X \to \{0,1\}^*$. We write $(q,s) \longrightarrow_c (q',s')$ when $(q',u) = \delta(q,c)$ and for all $x \in X$, $s'(x) = s^*(u(x))$ where s^* extends s to a function $s^* : (\{0,1\}+X)^* \to \{0,1\}^*$ defined in the natural way.

The image of a word $w = w_1 \dots w_n$ is $s^*(F(q))$ where $(q_I, (x \in X \mapsto \varepsilon)) \longrightarrow_{w_1} \dots \longrightarrow_{w_n} (q,s)$.

Theorem 3.4 ([2])**.** *A function is regular if and only if it is computed by an SST.*

Translating a SST to a EAλ-term of type `Str` $\multimap$ `Str` is a programming exercise without any major difficulty. To show that a term of type `Str` $\multimap$ `Str` computes a regular function, we translate it into a *non-deterministic* SST using the semantics briefly mentioned in the previous section. This non-determinism is due to the "additive" behavior of second-order quantifiers in affine logic, with some vague connections to the slices in multiplicative-additive proof nets. We have also managed to define a notion of "multi-pass SST" corresponding to polysequential functions.

Remark 3.5. This "additive" behavior in the second-order affine λ-calculus can be explained by its ability to encode additive connectives. In particular, we use the encoding

$$1 \oplus \dots \oplus 1 := \forall \alpha.\, \alpha \multimap \dots \multimap \alpha \multimap \alpha$$

to represent the set of states of a SST.

This demonstrates the usefulness of the additive disjunction. As for the additive conjunction

$$A \,\&\, B := \forall \alpha.\, (\forall \beta.\, \beta \multimap (A \multimap \beta) \multimap (B \multimap \beta) \multimap \alpha) \multimap \alpha$$

it is used when proving the equivalence between the type `BT` $\multimap$ `BT` and regular tree functions.

Indeed, the generalization of SSTs to binary trees, *bottom-up ranked tree transducers (BRTTs)* [1], uses a "conflict relation" over its set of registers, reminiscent of the (complement of the) coherence relation of a monomial proof net. We translate a BRTT as a EAλ-term instantiating its input of type `BT` $= \forall \alpha.$`BT`$[\alpha]$ at a type containing additive conjunctions.

References

[1] Rajeev Alur & Loris D'Antoni (2017): *Streaming Tree Transducers*. Journal of the ACM 64(5), pp. 1–55, doi:10.1145/3092842.

[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] Joost Engelfriet & Hendrik Jan Hoogeboom (2001): *MSO definable string transductions and two-way finite-state transducers*. ACM Transactions on Computational Logic 2(2), pp. 216–254, doi:10.1145/371316.371512.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] Lê Thành Dũng Nguyễn (2019): *On the elementary affine λ-calculus with and without type fixpoints*. Available at `https://nguyentito.eu/ealreg.pdf`. Submitted.

[13] 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, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 24–32, doi:10.1007/978-3-540-74510-5_6.

[14] Kazushige Terui (2012): *Semantic Evaluation, Intersection Types and Complexity of Simply Typed Lambda Calculus*. In: 23rd International Conference on Rewriting Techniques and Applications (RTA'12), pp. 323–338, doi:10.4230/LIPIcs.RTA.2012.323.

[15] 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.