On the elementary affine λ-calculus with and without type fixpoints

Lê Thành Dũng Nguyễn

On the elementary affine λ-calculus with and without type fixpoints

NGUYỄN Lê Thành Dũng*
LIPN, Université Paris 13, Sorbonne Paris Cité, 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 rely on type fixpoints (a.k.a. recursive types), and it was unknown whether this feature of the type system was really necessary. We give a positive answer by showing that without type fixpoints, we get a characterization of regular languages instead of polynomial time. The proof uses the semantic evaluation method. We also propose an aesthetic improvement on the characterization of the function classes FP and k-FEXPTIME in the presence of recursive types.

1 Introduction

The elementary affine λ-calculus Elementary Linear Logic (ELL), introduced by Girard [9], is a logic that can be seen as a typed functional programming language through the proof-as-programs correspondence. Its typing rules ensure that a function can be expressed if and only if it is elementary recursive (as is expounded in detail in [7]), hence the name. (This is an instance of the "type-theoretic" or "Curry–Howard" approach to implicit computational complexity.) This was refined by Baillot [1] into a characterization of each level of the k-EXPTIME hierarchy, in an affine variant of ELL.

A later improvement by Baillot, De Benedetti and Ronchi [2] consisted in turning this logic into an actual type system for a functional calculus with good properties (e.g. subject reduction), called the *elementary affine λ-calculus*. In this paper, we shall call their system μEAλ – the reason for the μ will soon become clear. The main result about it is:

Theorem 1.1 ([2])**.** *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 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 $!^k\mathtt{Bool}$ (i.e. $!(\ldots(!\mathtt{Bool}))$ with k '!') controls the complexity;
- the type of booleans is defined as $\mathtt{Bool} = \forall\alpha.\, \alpha \multimap \alpha \multimap \alpha$, and it has two inhabitants;
- $\mathtt{Str} = \forall\alpha.\mathtt{Str}[\alpha]$, 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 $w_1 \ldots w_n \in \{0,1\}^*$ is represented as the function which, for any type A, takes as input $f_0 : A \multimap A$ and $f_1 : A \multimap A$, and returns $f_{w_1} \circ \ldots \circ f_{w_n}$.

*Partially supported by the ANR project ELICA (ANR-14-CE25-0005).

Submitted to:
DICE-FOPARA 2019

Type fixpoints and Scott encodings We wish to draw attention to a particular feature of this language: the presence of *type fixpoints*[1], a.k.a. *recursive types*. An example is the type of *Scott binary strings*:

$$\mathtt{Str}_S := \forall\alpha.\,(\mathtt{Str}_S \multimap \alpha) \multimap (\mathtt{Str}_S \multimap \alpha) \multimap \alpha \multimap \alpha$$

In the elementary affine λ-calculus as defined in [2], this recursive equation can be turned into a valid type definition, by using a fixed point operator μ on types (this explains our name $\mu\mathrm{EA}\lambda$):

$$\mathtt{Str}_S := \mu\beta.\forall\alpha.\,(\beta \multimap \alpha) \multimap (\beta \multimap \alpha) \multimap \alpha \multimap \alpha$$

The idea is that strings are represented by their "pattern-matching" function (destructor): if u is a Scott binary string, then $u\ f_0\ f_1\ x$ morally means "if u represents the empty word, return x; else, return f_c applied to v where $c \in \{0,1\}$ is the first letter and v represents the suffix". Formally, we associate to each string $w \in \{0,1\}^*$ a $\mu\mathrm{EA}\lambda$-term $S(w)$ of type $\mathtt{Str}_S$:

$$S(\varepsilon) = \lambda f_0.\lambda f_1.\lambda x.x \quad S(0\cdot w') = \lambda f_0.\lambda f_1.\lambda x.f_0\,S(w') \quad S(1\cdot w') = \lambda f_0.\lambda f_1.\lambda x.f_1\,S(w')$$

This encoding of strings has been used to give a characterization of function classes in $\mu\mathrm{EA}\lambda$:

Theorem 1.2 ([2])**.** *The programs of type* $!\mathtt{Str} \multimap !^{k+2}\mathtt{Str}_S$ *in* $\mu\mathrm{EA}\lambda$ *compute exactly the functions in the class* k-FEXPTIME*. In particular* $!\mathtt{Str} \multimap\ !!\mathtt{Str}_S$ *corresponds to* FP.

Our contributions There are two natural questions concerning the necessity of type fixpoints:

- In the interface: it is possible to characterize this hierarchy of function classes using a function type involving only Church encodings?
- In the implementation: the extensional completeness proof for the predicate classes (Theorem 1.1) makes use of the type $\mathtt{Str}_S$ (to represent configurations of Turing machines), even though this type does not appear in the statement; could one avoid recursive types in the proof? This question has been raised by Baillot in the conclusion of [1].

In this paper, we answer both questions. The first one has a positive answer:

Theorem 1.3. *The programs of type* $!\mathtt{Str} \multimap !^{k+1}\mathtt{Str}$ *in* $\mu\mathrm{EA}\lambda$ *compute exactly the functions in the class* k-FEXPTIME*. In particular* $!\mathtt{Str} \multimap\ !\mathtt{Str}$ *corresponds to* FP.

An advantage of this characterization is that it reflects the fact that composing a k-FEXPTIME function f with a l-FEXPTIME function g gives a $(k+l)$-FEXPTIME function: since any $\mu\mathrm{EA}\lambda$-term of type $A \multimap B$ lifts to a term of type $!^kA \multimap !^kB$ (this is called "functorial promotion", cf. Proposition 2.2), we can compose the terms $f : !\mathtt{Str} \multimap !^{k+1}\mathtt{Str}$ and $g^{(k)} : !^{k+1}\mathtt{Str} \multimap !^{(l+1)+k}\mathtt{Str}$ to obtain a term of type $!\mathtt{Str} \multimap !^{(k+l)+1}\mathtt{Str}$. In particular FP is closed under composition. A characterization of FP in $\mu\mathrm{EA}\lambda$ by a function type whose input and output types coincide was proposed in [2], but it is less natural: a string is represented as a pair of its length (Church-encoded) and its contents (Scott-encoded).

As for the second question, we should first mention that Girard's original characterization of elementary recursive functions in ELL does not involve type fixpoints. This can be replayed in the elementary affine λ-calculus *without* type fixpoints, which we shall denote by $\mathrm{EA}\lambda$.

[1] A remark for the readers acquainted with typed λ-calculi: there is no "positivity" constraint imposed, yet those recursive types are 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 [9].

Theorem 1.4 ([1])**.** *The class of elementary recursive functions is the union, over $k \in \mathbb{N}$, of the classes of functions computed by programs of type* $!\mathtt{Str} \multimap !^k\mathtt{Str}$ *in* EAλ.

(The detailed proof given in [1] is for Elementary Affine Logic; it can be directly transposed to EAλ.)

However, the characterization of P by $!\mathtt{Str} \multimap !!\mathtt{Bool}$ fails in EAλ, as we show:

Theorem 1.5. *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* $\mathtt{Str} \multimap !\mathtt{Bool}$.

This result is surprising for a few reasons: the class of languages obtained is unexpectedly small, and it hints at connections between EAλ and formal language theory (the conclusion will discuss this further). The proof techniques for the above theorem are quite different from those used in [2]: instead of bounding the syntactic normalization process, we take inspiration from the tradition of implicit complexity in the *simply typed λ-calculus* (STλ), in particular from:

Theorem 1.6 (Hillebrand & Kanellakis [14])**.** *In the* simply typed λ-calculus*, the languages decided by terms of type* $\mathtt{Str}_{\mathrm{ST}\lambda}[A] \to \mathtt{Bool}_{\mathrm{ST}\lambda}$ *– A is a simple type that may be chosen depending on the language – are exactly the regular languages.*

Here $\mathtt{Str}_{\mathrm{ST}\lambda}[A] = (A \to A) \to (A \to A) \to (A \to A)$ and $\mathtt{Bool}_{\mathrm{ST}\lambda} = o \to o \to o$, where o is a base type. This is proved using the *semantic evaluation* method (see [19] and references therein). To make this method work in our case, we need a new result in denotational semantics:

Lemma 1.7. *The* second-order affine λ-calculus Aλ2 *– i.e. the subsystem of* EAλ *without the exponential modality '!' – admits a non-trivial* finite *semantics.*

By "non-trivial" we mean distinguishing the two inhabitants of $\mathtt{Bool} = \forall\alpha.\, \alpha \multimap \alpha \multimap \alpha$. The term "second-order" refers to the (impredicative) polymorphism supported by both μEAλ and EAλ – indeed, the types $\mathtt{Bool}$, $\mathtt{Str}$ and $\mathtt{Str}_S$ all contain second-order quantifiers ($\forall$). The lemma means morally that one cannot represent infinite data types in μEAλ without using the exponential modality – whereas in μEAλ, the exponential-free type $\mathtt{Str}_S$ encodes the infinite set $\{0,1\}^*$.

Thus, motivated by this question in implicit complexity, we set out to establish the above lemma, and came up with two approaches:

- a "category-theoretic" solution consists in showing the finiteness of a pre-existing model based on coherence spaces and normal functors; this is the subject of another paper [15];
- a "syntactic" solution, developed in a joint work with P. Pistone, T. Seiller and L. Tortora de Falco, relies on a careful combinatorial study of second-order proof nets; it will be written up in an upcoming paper.

The further development of these semantic tools has led to more results on EAλ and/or on Elementary Linear Logic without type fixpoints, which are beyond the scope of the present paper. This includes an already published joint work with P. Pradic [16] on logarithmic space.

Plan of the paper We recall from [2] the definitions of EAλ and μEAλ in Section 2, and then quickly prove Theorem 1.3 in Section 3. The bulk of the paper is Section 4, dedicated to proving Theorem 1.5. The conclusion (Section 5) discusses the above-mentioned new perspectives on EAλ opened up by our results and by refinements of Lemma 1.7.

Acknowledgments This work owes a great deal to Thomas Seiller's supervision. Thanks also to Patrick Baillot, Alexis Ghyselen, Damiano Mazza (an extremely fruitful discussion with Thomas and him triggered this work) and Pierre Pradic.

2 The elementary affine λ-calculus

The syntax of elementary affine λ-terms and the reduction rules are given by

$$t,u ::= x \mid \lambda x.t \mid \lambda\, !x.t \mid t\,u \mid !t \qquad (\lambda x.t)\,u \longrightarrow_\beta t\{x := u\} \quad (\lambda\, !x.t)\,(!u) \longrightarrow_! t\{x := u\}$$

where x is taken in a countable set of variables, and $t\{x := u\}$ refers to the substitution of all free occurrences of x in t by u. The reduction rules $\longrightarrow_\beta$ and $\longrightarrow_!$ are actually the contextual closure of the rules given above, for the obvious notion of context (see [2] for details).

We shall also write $\texttt{let}\ !x \leftarrow u\ \texttt{in}\ t$ for $(\lambda\, !x.t)\,u$ (this is just some "syntactic sugar"). The notion of *depth* of a subterm in a term, defined as the number of *exponential modalities* $!(-)$ ("exponentials" for short) surrounding the subterm, will play an important role.

As an example, let us formally define the Church-encoded binary strings:

$$\text{for } w = w_1 \dots w_n \in \{0,1\}^*, \quad \overline{w} = \lambda\, !f_0.\, \lambda\, !f_1.\, !(\lambda x.\, f_{w_1}\,(\dots(f_{w_n}\,x)\dots))$$

The above is essentially Simpson's linear λ-calculus with thunks [18]. (Other examples of linear λ-calculi with explicit exponentials are given in [12].) We shall now turn this untyped calculus into EAλ by endowing it with its type system – an adaptation of Coppola *et al.*'s Elementary Type Assignment System [8]. The grammar of types for EAλ is

$$A ::= \alpha \mid S \qquad S ::= \sigma \multimap \tau \mid \forall\alpha.S \qquad \sigma,\tau ::= A \mid\, !\sigma$$

The two first classes of types are called respectively *linear* and *strictly linear*. (We follow the terminology of [2]; "linear" does not mean exponential-free, it merely means that the head connective is not an exponential.) The reason for restricting quantification to strictly linear types is a technical subtlety related to subject reduction (see [8, §7.2]).

The typing judgements involve a context split into three parts: they are of the form $\Gamma \mid \Delta \mid \Theta \vdash t : \sigma$. The idea is that the partial assignements Γ, Δ and Θ of variables to types correspond respectively to linear, non-linear and "temporary" variables; accordingly, Γ maps variables to linear types (denoted A above), Δ maps variables to types of the form $!\sigma$, while Θ maps variables to arbitrary types. The domains of Γ, Δ and Θ are required to be pairwise disjoint. The derivation rules for EAλ are:

variable rules
$$\frac{}{\Gamma, x : A \mid \Delta \mid \Theta \vdash x : A} \qquad \frac{}{\Gamma \mid \Delta \mid \Theta, x : \sigma \vdash x : \sigma}$$

abstraction rules
$$\frac{\Gamma, x : A \mid \Delta \mid \Theta \vdash t : \tau}{\Gamma \mid \Delta \mid \Theta \vdash \lambda x.t : A \multimap \tau} \qquad \frac{\Gamma \mid \Delta, x : !\sigma \mid \Theta \vdash t : \tau}{\Gamma \mid \Delta \mid \Theta \vdash \lambda\, !x.t : !\sigma \multimap \tau}$$

application rule[2]
$$\frac{\Gamma \mid \Delta \mid \Theta \vdash t : \sigma \multimap \tau \quad \Gamma' \mid \Delta \mid \Theta \vdash u : \sigma}{\Gamma \uplus \Gamma' \mid \Delta \mid \Theta \vdash t\,u : \tau}$$

quantifier rules[3]
$$\frac{\Gamma \mid \Delta \mid \Theta \vdash t : S}{\Gamma \mid \Delta \mid \Theta \vdash t : \forall\alpha.S} \qquad \frac{\Gamma \mid \Delta \mid \Theta \vdash t : \forall\alpha.S}{\Gamma \mid \Delta \mid \Theta \vdash t : S\{\alpha := A\}}$$

functorial promotion rule
$$\frac{\varnothing \mid \varnothing \mid \Theta \vdash t : \sigma}{\Gamma \mid\, !\Theta, \Delta \mid \Theta' \vdash\, !t : !\sigma}$$

[2] $\Gamma \uplus \Gamma'$ means $\Gamma \cup \Gamma'$ with the assumption that the domains of Γ and Γ' are disjoint.

[3] In the introduction rule (left), α must not appear as a free variable in Γ, Δ and Θ.

In these rules, following the conventions established above, A stands for a linear type, S stands for a strictly linear type and σ and τ stand for arbitrary types. In particular, in the quantifier elimination rule, α can only be instantiated by a linear type. So, for instance, one cannot give the type $!\beta \multimap !\beta$ to $\lambda x.x$ through a quantifier introduction followed by a quantifier elimination; indeed, as one would expect, the only normal term of this type is $\lambda\, !x.\, !x$. (Despite this, the polymorphism is still impredicative.)

Coming back to the example of Church binary strings, one can show by induction that

$$\text{for } w = w_1 \dots w_n \in \{0,1\}^*, \quad x : \alpha \mid \varnothing \mid f_0 : \alpha \multimap \alpha, f_1 : \alpha \multimap \alpha \vdash f_{w_1}(\dots(f_{w_n} x)\dots) : \alpha$$

and deduce from this that $\vdash \overline{w} : \mathtt{Str}$ (recall that $\mathtt{Str} = \forall\alpha.\, !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$).

The system $\mu\mathrm{EA}\lambda$ is obtained by extending the grammar of types with $S ::= \dots \mid \mu\alpha.S$, and adding new derivation rules for the type fixpoint operator μ:

$$\mu\text{-fold} \quad \frac{\Gamma \mid \Delta \mid \Theta \vdash t : S\{\alpha := \mu\alpha.S\}}{\Gamma \mid \Delta \mid \Theta \vdash t : \mu\alpha.S} \qquad \mu\text{-unfold} \quad \frac{\Gamma \mid \Delta \mid \Theta \vdash t : \mu\alpha.S}{\Gamma \mid \Delta \mid \Theta \vdash t : S\{\alpha := \mu\alpha.S\}}$$

Let us recall two basic properties satisfied both by $\mathrm{EA}\lambda$ and $\mu\mathrm{EA}\lambda$, all proved in [2].

Proposition 2.1 (Stratification and linearity [2, Lemma 27])**.** *Let t be a typable term.*

- *for any subterm of the form $\lambda\, !x.u$ of t, all the occurrences of x must be at depth 1 in u;*
- *for any subterm $\lambda x.u$ of t, there is* at most one *occurrence of x in u, whose depth must be 0 in u.*

As a consequence, the reduction rules are depth-preserving.

Proposition 2.2 (k-fold functorial promotion [2, Proposition 28])**.** *Let $t : \sigma_1 \multimap \dots \multimap \sigma_n \multimap \tau$ is a closed elementary affine λ-term and $k \geq 1$. There is a term $t^{(k)} : !^k\sigma_1 \multimap \dots !^k\sigma_n \multimap !^k\tau$ such that $t^{(k)}\,(!^k u_1) \dots (!^k u_n)$ and $!^k(t\,u_1 \dots u_n)$ have the same normal form for all closed terms $u_i : \sigma_i$ ($i \in \{1,\dots,n\}$).*

3 The k-FEXPTIME hierarchy in $\mu\mathrm{EA}\lambda$ (proof of Theorem 1.3)

First, the soundness part of Theorem 1.3 follows immediately from Theorem 1.2.

Proposition 3.1. *All functions represented by $\mu\mathrm{EA}\lambda$-terms of type $!\mathtt{Str} \multimap !^{k+1}\mathtt{Str}$ are in k-FEXPTIME.*

Proof. There exists a coercion $!\mathtt{Str} \multimap !^2\mathtt{Str}_S$ (by completeness part of Theorem 1.2 applied to the identity function in FP) which lifts by functorial promotion (Proposition 2.2) to $!^{k+1}\mathtt{Str} \multimap !^{k+2}\mathtt{Str}_S$. So any function represented by a term of type $!\mathtt{Str} \multimap !^{k+1}\mathtt{Str}$ is also represented by a term of type $!\mathtt{Str} \multimap !^{k+2}\mathtt{Str}_S$. Thus the soundness part of Theorem 1.2 applies. □

For the extensional completeness, we also take Theorem 1.2 as our starting point. The idea is to convert $!\mathtt{Str}_S$ into $\mathtt{Str}$ with the help of an auxiliary integer which provides an upper bound on the length of the string. (Similar ideas appear in [3].)

We shall use the type of *Church natural numbers* and the usual second-order encoding of pairs:

$$\mathtt{Nat} = \forall\alpha.\, !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \qquad \sigma \otimes \tau = \forall\alpha.\, (\sigma \multimap \tau \multimap \alpha) \multimap \alpha$$

The aforementioned upper bound will be an inhabitant of the type $\mathtt{Nat}$. An integer $n \in \mathbb{N}$ is represented in $\mathtt{Nat}$ by the iterator $f \mapsto f^n$ (formally, $\overline{n} = \lambda\, !f.\, !(\lambda x.\, f\,(\dots(f\,x)\dots))$ with n times f).

To help readability we extend the syntax with the abbreviation

- $u \otimes v := \lambda f.\, f\, u\, v$ so that $u \otimes v : \sigma \otimes \tau$ if $u : \sigma$ and $v : \tau$

given in [2], and introduce some additional syntactic sugar:

- $\mathtt{let}\ x \otimes y \leftarrow u\ \mathtt{in}\ t := u\,(\lambda x.\,\lambda y.\,t)$ for $u : \sigma \otimes \tau$, and $\lambda(x \otimes y).t := \lambda z.\,\mathtt{let}\ x \otimes y \leftarrow z\ \mathtt{in}\ t$
- $\mathtt{case}\ u \mid \mathtt{0}x \mapsto a \mid \mathtt{1}y \mapsto b \mid \varepsilon \mapsto c := u\,(\lambda x.\,a)\,(\lambda y.\,b)\,c$ for $u : \mathtt{Str}_S$

The affine projections $\pi_i = \lambda(x_1 \otimes x_2).x_i$ ($i \in \{1,2\}$) are also defined in [2].

Remark 3.2. Our definition of $\lambda(x \otimes y).t$ is much simpler that the one given in [2], but the drawback is that it only works when the type of t is linear, i.e. its head connective is not an exponential. Indeed, $u : \sigma \otimes \tau$ can be instantiated to $u : (\sigma \multimap \tau \multimap A) \multimap A$ by the quantifier elimination rule only when A is linear. This condition will hold in our use cases below.

Now that we are equipped with all these data types, we can make progress on our proof.

Lemma 3.3. *There exists a* μEAλ*-term* $\mathtt{cast} : \mathtt{Nat} \multimap\ !\mathtt{Str}_S \multimap \mathtt{Str}$ *which converts a Scott encoding into a Church encoding, provided that the integer argument is greater or equal to the length of the string.*

Proof. Our implementation of `cast` instantiates the input `Nat` on $(\alpha \multimap \alpha) \otimes \mathtt{Str}_S$ where α is the eigenvariable of the $\forall$ in the output `Str` (recall that $S(\varepsilon)$ refers to the Scott encoding of the empty word):

$$\mathtt{cast} = \lambda n.\,\lambda\,!w.\,\lambda\,!f_0.\,\lambda\,!f_1.\,\mathtt{let}\ !g \leftarrow n\,!(\lambda(h \otimes u).t)\ \mathtt{in}\ !(\pi_1\,(g\,((\lambda x.x) \otimes w)))$$

$$\text{with } t = \mathtt{let}\ f \otimes v \leftarrow (\mathtt{case}\ u \mid \mathtt{0}v \mapsto f_0 \otimes v \mid \mathtt{1}v \mapsto f_1 \otimes v \mid \varepsilon \mapsto (\lambda z.z) \otimes S(\varepsilon))\ \mathtt{in}\ (\lambda x.h\,(f\,x)) \otimes v$$

To explain this functional program, let us reformulate it as an imperative algorithm: t can be considered as the body of a `for` loop which alters two mutable variables $h : (\alpha \multimap \alpha)$ and $u : \mathtt{Str}_S$. At each iteration, if u is non-empty, its first letter is popped (viewing u as a mutable stack) and h is post-composed with either f_0 or f_1 depending on this letter.

After n iterations starting from $h = (\lambda x.x)$ and $u = w$, if w is the Scott encoding of $w_1 \dots w_m$, the result obtained is $(f_{w_1} \circ \dots \circ f_{w_N}) \otimes (S(w_{N+1} \dots w_m))$ where $N = \min(n,m)$. In particular, if $n \geq m$, the first component will be $f_{w_1} \circ \dots \circ f_{w_m}$ – which corresponds to the definition of the Church encoding. □

To obtain the desired upper bound, we recall a lemma from [2]. It is used in the proof of Theorem 1.2 in order to simulate Turing machines.

Lemma 3.4 ([2])**.** *Let* $\mathscr{M}$ *be a* k*-FEXPTIME Turing machine. There is a* EAλ*-term* $t_{\mathscr{M}} : !\mathtt{Str} \multimap\ !^{k+1}\mathtt{Nat}$ *computing an upper bound on the running time of* $\mathscr{M}$ *on the given input string.*

We now have all the ingredients for the extensional completeness proof.

Theorem 3.5. *All* k-FEXPTIME *functions can be represented by* μEAλ*-terms of type* $!\mathtt{Str} \multimap\ !^{k+1}\mathtt{Str}$.

Proof. Consider any function computed by a k-FEXPTIME Turing machine $\mathscr{M}$. By the completeness part of Theorem 1.2, we can choose a μEAλ-term $f : !\mathtt{Str} \multimap\ !^{k+2}\mathtt{Str}_S$ computing this function. We also choose a term $t_{\mathscr{M}}$ satisfying the conditions of the above lemma. Then the term

$$\lambda\,!w.\,\mathtt{cast}^{(k+1)}\,(t_{\mathscr{M}}\,!w)\,(f\,!w) : !\mathtt{Str} \multimap\ !^{k+1}\mathtt{Str}$$

– where $\mathtt{cast}^{(k+1)}$ is the $(k+1)$-fold functorial promotion of `cast` – computes the same function as $\mathscr{M}$. Indeed, the assumption of Lemma 3.3 is satisfied, since for a Turing machine, the length of the output is bounded by the running time. □

4 Regular languages in EAλ (proof of Theorem 1.5)

In this section, we wish to show that, in EAλ (*without* fixpoints):

- all terms $t : !\texttt{Str} \multimap !!\texttt{Bool}$ decide regular languages;
- moreover, all regular languages can be decided by terms $t : \texttt{Str} \multimap !\texttt{Bool}$.

By functorial promotion, the class of languages characterized by $\texttt{Str} \multimap !\texttt{Bool}$ is included in the class corresponding to $!\texttt{Str} \multimap !!\texttt{Bool}$, so this will entail that both are exactly the class of regular languages. The situation is the opposite of the previous section: the second item (extensional completeness) is easy, while the first (soundness) is hard.

Regular languages admit many well-known equivalent definitions, e.g. regular expressions and finite automata (with many variants: non-determinism, bidirectionality, etc.). The classic characterization which will prove useful for us is:

Theorem 4.1. *A language is regular if and only if it can be expressed as $\varphi^{-1}(S)$, where $\varphi : \{0,1\}^* \to M$ is a monoid morphism, M is a* finite *monoid and $S \subseteq M$.*

4.1 Extensional completeness

Proposition 4.2. *All regular languages can be decided by* EAλ*-terms of type* $\texttt{Str} \multimap !\texttt{Bool}$.

Proof. Let $\varphi : \{0,1\}^* \to M$ be a morphism to a finite monoid M. Without loss of generality, we may assume that the underlying set of M is $\{1,\dots,k\}$, and the identity element of the monoid is 1. We represent the monoid elements in EAλ as inhabitants of the type $\texttt{M} = \forall\alpha.\,\alpha \multimap \dots \alpha \multimap \alpha$; the element i is mapped to the term $m_i = \lambda x_1.\dots.\lambda x_k.x_i$. We define:

- $\delta_c = \lambda m.\, m\, m_{\varphi(c)\cdot 1} \dots m_{\varphi(c)\cdot k} : \texttt{M} \multimap \texttt{M}$ for $c \in \{0,1\}$
- for $S \subseteq M$, $\chi_S = \lambda m.\, m\, b_1 \dots b_k : \texttt{M} \multimap \texttt{Bool}$ where $b_i = \texttt{true}$ (resp. $\texttt{false}$) if $i \in S$ (resp. $i \notin S$).

Then the language $\varphi^{-1}(S)$ is decided by the term $\lambda w.\,\texttt{let}\ !d \leftarrow w\, !\delta_0\, !\delta_1\ \texttt{in}\ !(\chi_S\,(d\,m_1))$. □

Next, to prepare the ground for our proof of soundness in EAλ, we review our direct inspiration in the simply typed λ-calculus: the proof of one direction of Theorem 1.6. The goal is to show that any simply typed λ-term $t : \texttt{Str}_{\mathrm{ST}\lambda}[A] \to \texttt{Bool}_{\mathrm{ST}\lambda}$, where A is an arbitrary simple type, decides a language $\mathscr{L}_{\mathrm{ST}\lambda}(t)$ which is *regular*. This was done using automata in [14], but we find it simpler to work with monoid morphisms (though this is, in the end, merely a different presentation of the same proof).

4.2 A short soundness proof for Hillebrand and Kanellakis's theorem (sketch)

We shall omit the subscripts in the types $\texttt{Str}_{\mathrm{ST}\lambda}[A]$ and $\texttt{Bool}_{\mathrm{ST}\lambda}$ in this subsection.

Let us fix a simple type A. The fundamental idea is that, given any *denotational semantics* $[\![-]\!]$:

- the denotation $[\![\overline{w}]\!] \in [\![\texttt{Str}[A]]\!]$ of the encoding of $w \in \{0,1\}^*$ is enough to determine $[\![t\,\overline{w}]\!] \in [\![\texttt{Bool}]\!]$ – this is simply the compositionality of the semantics;
- provided the semantics is *non-trivial*, i.e. $[\![\texttt{true}]\!] \neq [\![\texttt{false}]\!]$, this subsequently determines $t\,\overline{w}$.

Formally, let us define $\varphi_A : \{0,1\}^* \to [\![\texttt{Str}[A]]\!]$ by $\varphi_A(w) = [\![\overline{w}]\!]$; then if $[\![-]\!]$ is non-trivial,

$$\mathscr{L}_{\mathrm{ST}\lambda}(t) = \varphi_A^{-1}(\{\omega \in [\![\texttt{Str}[A]]\!] \mid [\![t]\!](\omega) = [\![\texttt{true}]\!]\})$$

To show that $\mathscr{L}_{\mathrm{ST}\lambda}(t)$ is regular, we shall apply Theorem 4.1 to this equation. We must make sure that:

- $[\![\mathtt{Str}[A]]\!]$ can be endowed with a monoid structure, in such a way that φ is a monoid morphism – this is caused by the use of *Church encodings*;
- $[\![\mathtt{Str}[A]]\!]$ is finite – thanks to the existence of a *finite semantics* for the simply typed λ-calculus.

Our choice of semantics, to satisfy both conditions, is the usual interpretation of types by mere sets (called the "full type frame" in [14]): $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$, with $[\![o]\!] = \{0,1\}$ for the base type. Any choice for $[\![o]\!]$ with at least two elements makes the semantics non-trivial. Furthermore, since $[\![o]\!]$ is finite, the denotations of all types are also finite.

Finally, in order to define a monoid structure on $[\![\mathtt{Str}[A]]\!]$, observe that

$$[\![\mathtt{Str}[A]]\!] = \left([\![A \to A]\!]^{[\![A \to A]\!]}\right)^{[\![A \to A]\!]} \cong \mathrm{End}([\![A]\!])^{\mathrm{End}([\![A]\!])^2}$$

where $\mathrm{End}([\![A]\!])$ is the monoid of maps from $[\![A]\!]$ to itself, endowed with function composition. Thus, the right-hand side can be seen as a product of monoids. Proving that φ is a morphism can then be done componentwise; the condition to be checked can be expressed as:

$$\forall (f_0, f_1) \in \mathrm{End}([\![A]\!])^2,\ (w \mapsto [\![\overline{w}]\!]\,(f_0, f_1)) \text{ is a morphism } \{0,1\}^* \to \mathrm{End}([\![A]\!])$$

By definition, $\overline{w} = \lambda f_0.\, \lambda f_1.\, \lambda x.\, f_{w_1}\,(\ldots(f_{w_n}\,x)\ldots)$ (where $w = w_1 \ldots w_n$) so

$$\forall (f_0, f_1) \in \mathrm{End}([\![A]\!])^2,\ [\![\overline{w}]\!]\,(f_0, f_1) = f_{w_1} \circ \ldots \circ f_{w_n}$$

therefore φ is none other than the product, over all $(f_0, f_1) \in \mathrm{End}([\![A]\!])^2$, of the monoid morphisms $\{0,1\}^* \to \mathrm{End}([\![A]\!])$ defined by $c \mapsto f_c$ for $c \in \{0,1\}$.

Remark 4.3. This reasoning can be made to work with any finite semantics of STλ, not just sets. An interesting choice is the "linearized Scott model"[4]: as remarked by Terui [19], in that semantics, the points in the denotation of a Church-encoded word correspond to nondeterministic finite automata accepting that word. This idea is also at the heart of Grellois and Melliès's semantic approach to higher-order model checking [11, 10].

4.3 Soundness for regular languages in EAλ

Our goal is now to emulate the above proof to show that the EAλ-terms of type $!\mathtt{Str} \multimap !!\mathtt{Bool}$ decide regular languages. (The result for $\mathtt{Str} \multimap !\mathtt{Bool}$ then follows by functorial promotion.) While the core of the semantic evaluation argument is similar, we need to do some syntactic analysis first before coming to this point.

4.3.1 Some lemmas and a truncation operation

Our proof relies on some general properties of EAλ. The two following ones were established in [2].

Proposition 4.4 (Reading property for booleans [2, Lemma 31(i)])**.** *The only closed inhabitants of the type* $!!\mathtt{Bool}$ *are* $!!\mathtt{true}$ *and* $!!\mathtt{false}$*. (*$\mathtt{true} = \lambda x.\, \lambda y.\, x$ *and* $\mathtt{false} = \lambda x.\, \lambda y.\, y$*)*

Proposition 4.5 (!-inversion [2, Lemma 29(i)])**.** *If* $\varnothing \mid \Delta \mid \varnothing \vdash t : !\sigma$*, then* $t = !t'$ *for some term* t'*.*

[4] This model is obtained from a semantics of linear logic as its exponential co-Kleisli category, i.e. via the translation $A \to B := !A \multimap B$. The resulting category embeds fully and faithfully into the usual category of Scott domains and continuous functions, hence the name. See [19] for a short self-contained definition.

We will also make use of a *truncation* operation on EAλ-terms. (To our knowledge, it has not appeared previously in the literature.) Its purpose is to erase all exponentials. This will be how the *stratification* property of EAλ (cf. Proposition 2.1) comes into play.

Definition 4.6. The *truncation at depth 0* $\|-\|_0$ is defined inductively on terms as:

$$\|!t\|_0 = (\lambda x.x) \quad \|(\lambda\, !x.t)\|_0 = \lambda x.\|t\|_0 \quad \|\lambda x.t\|_0 = \lambda x.\|t\|_0 \quad \|t\,u\|_0 = \|t\|_0\|u\|_0 \quad \|x\|_0 = x$$

and on types as (using the abbreviation[5] $1 = \forall\alpha.\,\alpha \multimap \alpha$):

$$\|!\sigma\|_0 = 1 \quad \|\sigma \multimap \tau\|_0 = \|\sigma\|_0 \multimap \|\tau\|_0 \quad \|\alpha\|_0 = \alpha \quad \|\forall\alpha.\sigma\|_0 = \forall\alpha.\|\sigma\|_0$$

Proposition 4.7. *If a typing judgment* $\Gamma \mid \Delta \mid \varnothing \vdash t : \sigma$ *is derivable in* EAλ*, then, writing* $\|\Gamma\|_0$ *for* $x_1 : \|\tau_1\|_0, \ldots, x_n : \|\tau_n\|_0$ *if* $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$*, the judgment* $\|\Gamma\|_0 \mid \varnothing \mid \varnothing \vdash \|t\|_0 : \|\sigma\|_0$ *is derivable. In particular, if* $t : \sigma$ *is a closed term, then* $\|t\|_0 : \|\sigma\|_0$*.*

Proof. By a mostly straightforward induction on the type derivation. Even so, let us treat a case involving a small subtlety: when the derivation ends with a quantifier elimination. In that case, the induction hypothesis gives us the typing judgment $\|\Gamma\|_0 \mid \varnothing \mid \varnothing \vdash \|t\|_0 : \forall\alpha.\|S\|_0$, and from this we must derive $\|\Gamma\|_0 \mid \varnothing \mid \varnothing \vdash t : \|S\{\alpha := \sigma\}\|_0$. What the same instantiation rule can give us from our premise is $\|\Gamma\|_0 \mid \varnothing \mid \varnothing \vdash t : \|S\|_0\{\alpha := \|\sigma\|_0\}$. One is therefore led to hope that $\|S\|_0\{\alpha := \|\sigma\|_0\} = \|S\{\alpha := \sigma\}\|_0$. Indeed, this can be checked by distinguishing, for each occurrence of α in σ, two possible cases: either it is at depth 0 and remains in $\|S\|_0$, or at depth ≥ 1 and is erased in $\|S\|_0$. □

Remark 4.8. The above proof is the reason why we do not generalize here our truncation operation to a "truncation at depth k" for $k \geq 1$, which would erase all exponentials of depth $> k$. Indeed, a typical example for which the above reasoning would fail is the truncation at depth 1 of $\forall\alpha.\,!\alpha \multimap \alpha$ instantiated with $\alpha := !\tau$. So these higher depths truncations would need additional conditions to be well-behaved.

Proposition 4.9. *For all* $k \in \mathbb{N}$ *and all* EAλ*-terms* t, t'*, if* $t \longrightarrow t'$*, then* $\|t\|_0 \longrightarrow^* \|t'\|_0$ *(this also applies to untyped terms which satisfy the stratification property, i.e. the conclusions of Proposition 2.1).*

Proof. If the redex contracted in t to obtain t' is at depth ≥ 1, then one can prove that $\|t\|_0 = \|t'\|_0$.

Otherwise, by induction on the context of the redex, one can restrict to the case where $t = u\,v$ and the application of u to v is the contracted redex. We proceed by case analysis:

- If $u = \lambda x.u'$, then $t' = u'\{x := v\}$. We use the fact that x appears only at depth zero in u' (Proposition 2.1) to show that $\|u'\{x := v\}\|_0 = \|u'\|_0\{x := \|v\|_0\}$. The latter is a reduct of $\|u\|_0\|v\|_0 = \|t\|_0$.
- If $u = \lambda\, !x.u'$, then $v = !v'$ and $t' = u'\{x := v'\}$. Moreover, x appears only at depth 1 in u' (again by Proposition 2.1). Therefore, $\|u'\|_0$ does not contain x as a free variable; thus, $\|t'\|_0 = \|u'\|_0$ is a reduct of $\|t\|_0 = (\lambda x.\|u'\|_0)\,\|v\|_0$. □

A final general observation (unrelated to truncation) before delving into the soundness proof itself:

Proposition 4.10. *Let* t *be a term a free variable* x*. Suppose that* $t = t'\{x_1 := x\}\ldots\{x_n := x\}$*, where each* x_i *appears only once in* t' *(so* n *is the number of occurrences of* x *in* t*), and* $\Gamma \mid \Delta \mid \Theta, x : A \vdash t : \tau$*, where* A *is linear (i.e. not of the form* $!\sigma$*). Then* $\Gamma, x_1 : A, \ldots, x_n : A \mid \Delta \mid \Theta \vdash t : \tau$*. We write* $t' = t\{x := x_1, \ldots, x_n\}$ *for this situation. (Such a* t' *always exists given* t*.)*

Proof. By induction on typing derivations, replacing each rule of the form $\ldots \mid \ldots \mid \ldots, x : A \vdash x : \sigma$ by a rule of the form $\ldots, x_i : A, \ldots \mid \ldots \mid \ldots \vdash x_i : A$ for some $i \in \{1, \ldots, n\}$. □

[5] This is justified as $\forall\alpha.\,\alpha \multimap \alpha$ is the unit to the tensor product used in Section 3.

4.3.2 Syntactic analysis

We can now start looking at the languages decided by EAλ-terms.

Lemma 4.11. *For any* EAλ*-term* $t : !\mathtt{Str} \multimap !!\mathtt{Bool}$*, there exists* $u : \mathtt{Str}[\sigma_1] \multimap \ldots \multimap \mathtt{Str}[\sigma_n] \multimap !\mathtt{Bool}$ *(for some* $n \in \mathbb{N}$*) such that, for all* $s : \mathtt{Str}$*,* $t\,!s$ *and* $!(u\,s \ldots s)$ *have the same normal form.*

Proof. First, one may take t to be in normal form. In that case, the only possible redex in $t\,!s$ is the application at the root. Moreover, $t\,!s$ must be reducible since it is neither $!!\mathtt{true}$ nor $!!\mathtt{false}$, cf. Proposition 4.4. Therefore, $t = (\lambda\,!x.t')$ (the case $t = (\lambda x.t')$ can be excluded because then t would be of type $A \multimap \tau$ where A is linear, in particular $A \neq !\mathtt{Str}$).

The next step is to prove that $t = \lambda\,!x.\,!t''$ for some EAλ-term t''. According to the typing rules, the judgment $\varnothing \mid \varnothing \mid \varnothing \vdash t : !\mathtt{Str} \multimap !!\mathtt{Bool}$ can only be proven by first establishing $\varnothing \mid x : !\mathtt{Str} \mid \varnothing \vdash t' : !!\mathtt{Bool}$. According to the !-inversion property (Proposition 4.5), since the first and third part of the typing context are empty and the head connective of the type is '!', t' must be of the form $!t''$.

Finally, since $\varnothing \mid \varnothing \mid x : \mathtt{Str} \vdash t' : !!\mathtt{Bool}$ must hold (it is the only premise which can lead to the above judgement on t'), we can apply Proposition 4.10 to t'' (indeed, the type $\mathtt{Str}$ is linear). Then, the term $u = \lambda x_1.\ldots\lambda x_m.t''\{x := x_1,\ldots,x_m\}$ (where x occurs m times in t'') enjoys the property claimed in the lemma statement. □

Let us focus on the case $n = 1$ for a moment, and do the same kind of analysis again.

Lemma 4.12. *Let* $u : \mathtt{Str}[\sigma] \multimap !\mathtt{Bool}$ *be an* EAλ*-term, and let* $\tau = \sigma \multimap \sigma$.

There exist EAλ*-terms* $f_0 : \tau$*,* $f_1 : \tau$ *and* $g : \tau \multimap \ldots \multimap \tau \multimap !\mathtt{Bool}$ *(with* m *times* τ*, for some* $m \in \mathbb{N}$*) such that for all* $s : \mathtt{Str}$*, if* $s\,!f_0\,!f_1 \longrightarrow^* !h$*, then* $u\,s$ *and* $!(g\,h \ldots h)$ *have the same normal form.*

Proof. We assume that u is in normal form. Since the head connective of $\mathtt{Str}[\sigma]$ is not '!', $u = \lambda x.v$ and $x : \mathtt{Str}[\sigma] \mid \varnothing \mid \varnothing \vdash v : !\mathtt{Bool}$. We may assume that v contains x as a free variable; otherwise, u is a constant function and the conclusion we want holds trivially (take $m = 0$).

Let us examine in general the shape of $v : !\theta$ in normal form (where θ is not necessarily $\mathtt{Bool}$) such that x appears free in v and $x : \mathtt{Str}[\sigma] \mid \varnothing \mid \varnothing \vdash v : !\theta$. By [2, Lemma 29(ii)], v must be an application: $v = p\,q_1 \ldots q_k$ where p is not an application and $k \geq 1$. Observe that p cannot be of the form $\lambda y.\,p'$, since $p\,q_1$ would then be a redex. There are two possible cases:

- $p = x$, and then $\theta = \sigma \multimap \sigma = \tau$, $k = 2$ and the closed EAλ-terms $q_1, q_2 : !\tau$ must be of the form $q_i = !q_i'$ by !-inversion (Proposition 4.5)
- $p = (\lambda\,!y.\,p')$, in which case x must appear free in q_1. Indeed, suppose for the sake of contradiction that q_1 is closed; then $\varnothing \mid \varnothing \mid \varnothing \vdash q_1 : \theta_1 = !\rho$ for some ρ, therefore !-inversion gives us $q_1 = !r$ for some r, so $p\,q_1$ would be a redex.

In the second case, we may furthermore take $k = 1$ w.l.o.g.: if $k \geq 2$, then for all $s : \mathtt{Str}[\sigma]$, the term $((\lambda\,!y.\,p'\,q_2 \ldots q_k)\,q_1)\{x := s\}$ has the same normal form as $v\{x := s\}$ (this is analogous to Regnier's σ-equivalence by redex permutations [17]). And since $\varnothing \mid y : !\rho \mid \varnothing \vdash p'\,q_2 \ldots q_k : !\theta$, the normal form of $p'\,q_2 \ldots q_k$ is of the form $!p''$ (this is again an application of !-inversion).

To recapitulate, either $v = x\,!f_0\,!f_1$ or $v = (\lambda\,!y.\,!p)\,v' = \mathtt{let}\ !y \leftarrow v'\ \mathtt{in}\ !p$ where x appears free in v', but not in p. In the latter case, we have $x : \mathtt{Str}[\sigma] \mid \varnothing \mid \varnothing \vdash v' : !\theta'$. So, by induction on the size of terms,

$$v = \mathtt{let}\ !y_1 \leftarrow (\ldots(\mathtt{let}\ !y_l \leftarrow x\,!f_0\,!f_1\ \mathtt{in}\ !p_l)\ldots)\ \mathtt{in}\ !p_1$$

As a consequence, for all $s : \mathtt{Str}$, if $s\,!f_0\,!f_1 \longrightarrow^* !h$ (recall that $f_0, f_1 : \tau$ are closed) then

$$u\,s = (\lambda x.\, v)\,s \longrightarrow v\{x := s\} \longrightarrow^* !(p_1\{y_1 := (\dots p_l\{y_l := h\}\dots)\})$$

(Morally, we are still trying to permute redexes; the reader may check that there is an analogy between the above operation and Carraro and Guerrieri's $V\,((\lambda x.L)N) \rightsquigarrow (\lambda x.V\,L)N$ rule [5] for the call-by-value λ-calculus.)

Let $r = p_1\{y_1 := (\dots p_l\{y_l := z\}\dots)\}$, where z is a fresh variable, so that the right-hand side can be written as $!(r\{z := h\})$. Since $h : \tau$ is a closed subterm of $!(r\{z := h\}) : !\mathtt{Bool}$ (we are using subject reduction here), then it must be true that $\varnothing \mid \varnothing \mid z : \tau \vdash r : \mathtt{Bool}$. Let us now apply Proposition 4.10, using the fact that $\tau = \sigma \multimap \sigma$ is linear: for some $m \in \mathbb{N}$, $z_1 : \tau, \dots, z_m : \tau \mid \varnothing \mid \varnothing \vdash r\{z := z_1, \dots, z_m\} : \mathtt{Bool}$.

Finally, we take $g = \lambda z_1. \dots \lambda z_m. r\{z := z_1, \dots, z_m\}$. The lemma statement holds with the f_0, f_1, m and g that we have constructed. □

The last purely syntactic step is to use the truncation operation to formulate a variation of the above lemma. The point is to be able to decide the membership in the language defined by an EAλ-term by computing purely in Aλ2. This sets the stage for the use of a semantics of Aλ2.

Lemma 4.13. *Let $u : \mathtt{Str}[\sigma] \multimap !\mathtt{Bool}$ be an* EAλ*-term, and let $\tau = \|\sigma \multimap \sigma\|_0$.*

There exist Aλ2*-terms $f_0 : \tau$, $f_1 : \tau$ and $g : \tau \multimap \dots \multimap \tau \multimap !\mathtt{Bool}$ (with m times τ, for some $m \in \mathbb{N}$) such that for all $w \in \{0,1\}^*$, if $\overline{w}\,!f_0\,!f_1 \longrightarrow^* !h$, then $u\,\overline{w}$ and $!(g\,h\dots h)$ have the same normal form.*

(Recall that $\overline{w} : \mathtt{Str}$ is the Church encoding of w in EAλ*.)*

Proof. Thanks to the previous lemma, there exist $f_0' : \sigma \multimap \sigma$, $f_1' : \sigma \multimap \sigma$ and $g' : (\sigma \multimap \sigma) \multimap \dots \multimap (\sigma \multimap \sigma) \multimap !\mathtt{Bool}$ with m times τ, for some $m \in \mathbb{N}$, such that the conclusion holds by replacing τ by $\sigma \multimap \sigma$ and f_0, f_1, g by f_0', f_1', g'. The only issue is that f_0', f_1', g' might not be in Aλ2. The idea is therefore to take $f_0 = \|f_0'\|_0$, $f_1 = \|f_1'\|_0$ and $g = \|g'\|_0$, and to check that this works.

Let $h' : \sigma \multimap \sigma$ be such that $\overline{w}\,!f_0'\,!f_1' \longrightarrow^* !h'$. Since $\overline{w} = \lambda\,!a_0.\,\lambda\,!a_1.\,!(\lambda x.\, a_{w_1}\,(\dots(a_{w_n}\,x)\dots))$,

$$\lambda x.\, f'_{w_1}\,(\dots(f'_{w_n}\,x)\dots) \longrightarrow^* h' \quad \text{and by truncation} \quad \lambda x.\, f_{w_1}\,(\dots(f_{w_n}\,x)\dots) \longrightarrow^* \|h'\|_0$$

So if h is such that $\overline{w}\,!f_0\,!f_1 \longrightarrow^* !h$, then by confluence [2, Lemma 8], h and $\|h'\|_0$ have the same normal form. Therefore, $g\,h\dots h$ and $g\,\|h'\|_0 \dots \|h'\|_0$ have the same normal form. But the latter is none other than $\|g'\,h'\dots h'\|_0$.

To conclude, observe that:

- the normal form of $!(g'\,h'\dots h')$ is the same as that of $u\,\overline{w}$ by the previous lemma;
- by Proposition 4.4, the normal form of $g'\,h'\dots h'$ is some $b \in \{\mathtt{true}, \mathtt{false}\}$;
- since $\|b\|_0 = b$, $!\|g'\,h'\dots h'\|_0$ has the same normal form as $u\,\overline{w}$.

By the discussion above, this means that $!(g\,h\dots h)$ and $u\,\overline{w}$ have the same normal form, as desired. □

4.3.3 Semantic evaluation

We are now ready to conclude our proof of soundness by adapting Hillebrand and Kanellakis's argument. Let $[\![-]\!]$ be any non-trivial *finite* semantics of Aλ2 – the notion of finiteness we need is that $[\![A]\!]$ has finitely semantic inhabitants for all Aλ2 types A. (Equivalently, if our semantics is a category with a terminal object 1, we require $\mathrm{Hom}(1, [\![A]\!])$ to be finite for all A.) Recall that although such a semantics is a central ingredient in our proof, we have simply assumed its existence, which is proved elsewhere (see Lemma 1.7 and the subsequent discussion).

Definition 4.14. Let A be a Aλ2 type. We define $\Phi_A(w)(\gamma_0,\gamma_1) = \gamma_{w_1} \circ \ldots \circ \gamma_{w_n}$ for $w \in \{0,1\}^*$ and $(\gamma_0,\gamma_1) \in \mathrm{End}(\llbracket A \rrbracket)^2$. In other words, $\Phi_A : \{0,1\}^* \to \mathrm{End}(\llbracket A \rrbracket)^{\mathrm{End}(\llbracket A \rrbracket)^2}$ is the monoid morphism sending $c \in \{0,1\}$ to $(\gamma_0,\gamma_1) \mapsto \gamma_c$.

Here $\mathrm{End}(\llbracket A \rrbracket)$ refers to the monoid of endomorphisms of $\llbracket A \rrbracket$ in the semantics.

Proposition 4.15. *Let $w \in \{0,1\}^*$ and $\bar{w} : \mathtt{Str}$ be its encoding. For any* Aλ2 *type σ and* Aλ2*-terms $f_0, f_1 : \sigma \multimap \sigma$, $\bar{w}\,!f_0\,!f_1$ normalizes into some $!h$ with $h : \sigma \multimap \sigma$, and $\Phi_A(w)(\llbracket f_0 \rrbracket, \llbracket f_1 \rrbracket) = \llbracket g \rrbracket$.*

Proof. As in the case of the simply typed λ-calculus, this is by definition of the Church encoding. □

Lemma 4.16. *Let $u : \mathtt{Str}[\sigma_1] \multimap \ldots \multimap \mathtt{Str}[\sigma_n] \multimap !\mathtt{Bool}$.*

For all $w_1, \ldots, w_n \in \{0,1\}^*$, the normal form of $u\overline{w_1} \ldots \overline{w_n}$ is completely determined by the functions $\Phi_{\|\sigma_1\|_0}(w_1), \ldots, \Phi_{\|\sigma_n\|_0}(w_n)$. As a consequence, the following language is regular:

$$\{w \in \{0,1\}^* \mid u\overline{w} \ldots \overline{w} \longrightarrow^* !\mathtt{true}\}$$

Proof. We start with the case $n = 1$, in which $u : \mathtt{Str}[\sigma] \multimap !\mathtt{Bool}$. Let $f_0 : \tau$, $f_1 : \tau$ and $g : \tau$ be given by Lemma 4.13, where $\tau = \|\sigma \multimap \sigma\|_0 = \|\sigma\|_0 \multimap \|\sigma\|_0$. For all $w \in \{0,1\}^*$, if $\overline{w}\, f_0\, f_1 \longrightarrow^* !h$, then $u\overline{w} \longrightarrow^* b$ for some $b \in \{\mathtt{true}, \mathtt{false}\}$ such that $g\,h \ldots h \longrightarrow^* b$. Since f_0, f_1 and g are in Aλ2, so is h (provided it is normal), and $\llbracket g\,h \ldots h \rrbracket = \llbracket g \rrbracket (\llbracket h \rrbracket, \ldots, \llbracket h \rrbracket)$ by compositionality. Therefore

$$\llbracket b \rrbracket = \llbracket g \rrbracket \left(\Phi_{\|\sigma\|_0}(w)(\llbracket f_0 \rrbracket, \llbracket f_1 \rrbracket), \ldots, \Phi_{\|\sigma\|_0}(w)(\llbracket f_0 \rrbracket, \llbracket f_1 \rrbracket)\right)$$

thanks to the previous proposition. Since our semantics is *non-trivial*, i.e. $\llbracket b \rrbracket = \llbracket \mathtt{true} \rrbracket \iff b = \mathtt{true}$, $\Phi_{\|\sigma\|_0}(w)$ thus determines the normal form of $u\overline{w}$.

The result for arbitrary $n \geq 1$ is obtained by induction on n by repeatedly applying the case $n = 1$.

The consequence is that the language $\{w \in \{0,1\}^* \mid u\overline{w} \ldots \overline{w} \longrightarrow^* !\mathtt{true}\}$ can be written using only conditions on $\Phi_{\|\sigma_i\|_0}(w)$ ($i \in \{1, \ldots, n\}$), so it is the preimage of some subset of $\prod_{i=1}^n \mathrm{End}(\llbracket \sigma_i \rrbracket)^{\mathrm{End}(\llbracket \sigma_i \rrbracket)^2}$ by the monoid morphism $w \mapsto (\Phi_{\|\sigma_1\|_0}(w), \ldots, \Phi_{\|\sigma_n\|_0}(w))$. □

This suffices to conclude the soundness proof. Let $t : !\mathtt{Str} \multimap !!\mathtt{Bool}$. Then, by Lemma 4.11,

$$\{w \in \{0,1\}^* \mid t\,!\overline{w} \longrightarrow^* !!\mathtt{true}\} = \{w \in \{0,1\}^* \mid u\overline{w} \ldots \overline{w} \longrightarrow^* !\mathtt{true}\}$$

for some $u : \mathtt{Str}[\sigma_1] \multimap \ldots \multimap \mathtt{Str}[\sigma_n] \multimap !\mathtt{Bool}$. The regularity of this language then follows from the above lemma.

4.4 Overcoming the expressivity barrier

Analyzing the our soundness proof for regular languages in EAλ reveals that fundamentally, what restricts the computational power is a conjunction of two facts:

1. the input $\mathtt{Str}$ is instantiated on some types $\sigma_1, \ldots, \sigma_n$ *known in advance*;
2. these σ_i are morally finite data types, since they admit finite semantics.

This makes it impossible to iterate over, say, the configurations of a Turing machine, since their size depends on the input and the type σ_i cannot "grow" to accomodate data of variable size.

If we stay at depth 2 in EAλ, there is no way of avoiding the second fact (one can always truncate the σ_i to exponential-free types), so if we want to retrieve a larger complexity class than regular languages without resorting to type fixpoints, we should try to circumvent the first obstacle. That means that the σ_i should vary with the input. Thus, we are led to consider that *inputs should provide types*:

- the encoding of an input x would be a term $t_x : \mathtt{Inp}[A_x]$, for some type $\mathtt{Inp}$ with one parameter;
- this t_x would then be given as argument to a program of type $\forall\alpha.\,\mathtt{Inp}[\alpha] \multimap \mathtt{Bool}$.

In other words, we are considering *existential* input types. Indeed, if we were to extend EAλ with existential quantifiers[6], there would be an isomorphism $(\exists\alpha.\,\mathtt{Inp}[\alpha]) \multimap \mathtt{Bool} \cong \forall\alpha.\,(\mathtt{Inp}[\alpha] \multimap \mathtt{Bool})$.

Remark 4.17. In fact there is a third fact which plays a role in bridling the complexity: the shape of the type $\mathtt{Str}$ which codes sequential iterations (but the same could be said of Church encodings of free algebras – with such inputs one characterizes regular tree languages).

For instance, let us consider as inputs circuits represented by the type

$$\forall X.!X \multimap !(X \multimap X \multimap X) \multimap !(X \multimap X \otimes X) \multimap !X$$

where $!X$ corresponds to $\mathtt{true}$ constants, $!(X \multimap X \multimap X)$ corresponds to $\mathtt{nand}$ gates, and $!(X \multimap X \otimes X)$ corresponds to duplication gates used to represent fan-out. Then instantiating this with $X = \mathtt{Bool}$ and the obvious evaluation maps gives us an encoding of the circuit value problem, which is P-complete.

Although this input type seems morally less legitimate than Church encodings, it is hard to pinpoint precisely why it should be rejected.

5 Conclusion

This paper started with a positive result: there exists a characterization of FP and k-FEXPTIME in μEAλ whose statement is very simple. However, the characterization of regular languages in EAλ, which takes up the rest of the paper, could be seen as a negative result: it demonstrates the lack of expressivity of EAλ without type fixpoints. (This is the spirit of Section 4.4.) Indeed, the small class of regular languages not quite a well-behaved complexity class, e.g. it is not closed under AC^0 reductions.

That said, one can also read Theorem 1.5 as positive evidence of a connection between affine typing and automata. 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.
- On the other hand, one can hope to obtain a well-behaved sub-polynomial complexity class by changing the representation of inputs, following the suggestions of Section 4.4.

We are currently working on the first research direction, by attempting to capture classes of *transductions*, i.e. of functions computed by automata with output. As of the time of writing, it seems likely that in EAλ, $\mathtt{Str} \multimap \mathtt{Str}$ captures the well-known class of *regular functions* (see the introduction of [4] for an overview of classical transduction classes, including regular functions), and that the class defined by $!\mathtt{Str} \multimap !\mathtt{Str}$ also admits an automata-theoretic characterization.

As for the second one, it is the topic of a sequel[7] paper [16] (joint work with P. Pradic) which studies an input type inspired by finite model theory, following Hillebrand's thesis [13]. We obtain what we believe to be a characterization of *deterministic logarithmic space* (L), and manage to prove that the class we capture is between L and NL[8].

[6] The reason this extension is not incorporated is that existentials can be encoded: $\exists\alpha.\,\tau := \forall\beta.\,(\forall\alpha.\,\tau \multimap \beta) \multimap \beta$.

[7] This sequel has been published first, although the results in the present paper were mostly obtained before.

[8] Actually, a more precise upper bound is L with an oracle for *unambiguous* non-deterministic logarithmic space.

The importance of semantics A novelty in our approach is that we betray the original spirit of "light logics" such as Light Linear Logic and Elementary Linear Logic [9], which consisted in bounding the complexity of normalization "geometrically", independently of types. Here:

- geometry still plays an important structuring role, reflected by our use of a "truncation at depth zero" operation, which may be of independent interest;
- but our fine-grained analysis also requires to take into account the influence of types through semantics.

Though we are not the first to apply semantics to obtain inexpressivity results in light logics (cf. e.g. [6]), our recent discovery of a finite semantics of linear polymorphism (cf. the discussion below the statement of Lemma 1.7) opens up new possibilities. The above-mentioned sequel on logarithmic space is an illustration of this new way of working in EAλ and its variants: the best upper bound that we have is obtained using the effectiveness of the second-order coherence space model studied in [15].

Open questions Aside from the perspectives already mentioned, there is an obvious question that remains after Theorem 1.5: what about $!\mathtt{Str} \multimap !^{k+2}\mathtt{Bool}$ (resp. $!\mathtt{Str} \multimap !^{k+1}\mathtt{Str}$) for $k \geq 1$? For now, what we know about the corresponding complexity class is that:

- it is contained in k-EXPTIME (resp. k-FEXPTIME), since the soundness results for μEAλ apply *a fortiori* to EAλ;
- it contains $(k-1)$-EXPTIME (resp. $(k-1)$-FEXPTIME), by adapting the proofs given in [1].

We must confess that we have no idea about what class $!\mathtt{Str} \multimap !^{k+2}\mathtt{Bool}$ corresponds to, let alone about a proof strategy. Our only guesses is that the first containment is strict, and that semantics can prove useful for this problem.

References

[1] 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.

[2] 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.

[3] Patrick Baillot & Alexis Ghyselen (2018): *Combining Linear Logic and Size Types for Implicit Complexity*. In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, pp. 9:1–9:21, doi:10.4230/LIPIcs.CSL.2018.9.

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

[5] Alberto Carraro & Giulio Guerrieri (2014): *A Semantical and Operational Account of Call-by-Value Solvability*. In: *Foundations of Software Science and Computation Structures (FoSSaCS'14)*, pp. 103–118, doi:10.1007/978-3-642-54830-7_7.

[6] Ugo Dal Lago & Patrick Baillot (2006): *On light logics, uniform encodings and polynomial time*. Mathematical Structures in Computer Science 16(4), pp. 713–733, doi:10.1017/S0960129506005421.

[7] Vincent Danos & Jean-Baptiste Joinet (2003): *Linear logic and elementary time*. Information and Computation 183(1), pp. 123–137, doi:10.1016/S0890-5401(03)00010-5.

[8] Simona Ronchi Della Rocca, Ugo Dal Lago & Paolo Coppola (2008): *Light Logics and the Call-by-Value Lambda Calculus*. Logical Methods in Computer Science Volume 4, Issue 4, doi:10.2168/LMCS-4(4:5)2008.

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

[10] Charles Grellois (2016): *Semantics of linear logic and higher-order model-checking*. Ph.D. thesis, Université Denis Diderot Paris 7. Available at `https://tel.archives-ouvertes.fr/tel-01311150/`.

[11] Charles Grellois & Paul-André Melliès (2015): *Finitary Semantics of Linear Logic and Higher-Order Model-Checking*. In: *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015*, pp. 256–268, doi:10.1007/978-3-662-48057-1_20.

[12] Giulio Guerrieri & Giulio Manzonetto (2019): *The Bang Calculus and the Two Girard's Translations*. *Electronic Proceedings in Theoretical Computer Science* 292, pp. 15–30, doi:10.4204/EPTCS.292.2.

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

[14] 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.

[15] Lê Thành Dũng Nguyễn (2019): *Around finite second-order coherence spaces*. *CoRR* abs/1902.00196.

[16] 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. 151:1–151:15, doi:10.4230/LIPIcs.ICALP.2019.151.

[17] Laurent Regnier (1994): *Une équivalence sur les lambda-termes*. *Theoretical Computer Science* 126(2), pp. 281–292, doi:10.1016/0304-3975(94)90012-4.

[18] Alex Simpson (2005): *Reduction in a Linear Lambda-Calculus with Applications to Operational Semantics*. In: *16th International Conference on Term Rewriting and Applications (RTA'05)*, pp. 219–234, doi:10.1007/978-3-540-32033-3_17.

[19] 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.