Implicit Automata in Typed λ-Calculi I: Aperiodicity in a Non-Commutative Logic

Lê Thành Dũng Nguyễn
Laboratoire d'informatique de Paris Nord, Villetaneuse, France
Laboratoire Cogitamus (http://www.cogitamus.fr/indexen.html)
nltd@nguyentito.eu

Pierre Pradic
Department of Computer Science, University of Oxford, UK
pierre.pradic@cs.ox.ac.uk

Abstract

We give a characterization of *star-free languages* in a λ-calculus with support for *non-commutative affine types* (in the sense of linear logic), via the algebraic characterization of the former using *aperiodic monoids*. When the type system is made commutative, we show that we get *regular languages* instead. A key ingredient in our approach – that it shares with higher-order model checking – is the use of *Church encodings* for inputs and outputs. Our result is, to our knowledge, the first use of non-commutativity in implicit computational complexity.

2012 ACM Subject Classification Theory of computation → Algebraic language theory; Theory of computation → Linear logic

Keywords and phrases Church encodings, ordered linear types, star-free languages

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.135

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version Full version with appendices: https://hal.archives-ouvertes.fr/hal-02476219

Acknowledgements The initial trigger for this line of work was twofold: the serendipitous rediscovery of Hillebrand and Kanellakis's theorem by Damiano Mazza, Thomas Seiller and the first author, and Mikołaj Bojańczyk's suggestion to the second author to look into connections between transducers and linear logic. Célia Borlido proposed looking at star-free languages at the Topology, Algebra and Categories in Logic 2019 summer school.
During its long period of gestation, this work benefited from discussions with many other people: Pierre Clairambault, Amina Doumane, Marie Fortin, Jérémy Ledent, Paolo Pistone, Lorenzo Tortora de Falco, Noam Zeilberger and others that we may have forgotten to mention.
We also thank the anonymous reviewers for their constructive feedback, especially for their highly relevant pointers to the literature.

1 Introduction

A type-theoretic implicit automata theory. This paper explores connections between the languages recognized by automata and those definable in certain typed λ-calculi (minimalistic functional programming languages). It is intended to be the first in a series, whose next installments will investigate the functions computable by transducers (automata with output, see e.g. [15, 36]). Insofar as programming language theory is related to proof theory, via the Curry–Howard correspondence, we are therefore trying to *bridge logic and automata*. That said, our work does not fit in the "logics as specification languages" paradigm, exemplified by the equivalence of recognition by finite-state automata and Monadic Second-Order Logic (MSO). One could sum up the difference by analogy with the two main approaches to machine-free complexity: *implicit computational complexity (ICC)* and *descriptive complexity*.

Both aim to characterize complexity classes without reference to a machine model, but the methods of ICC have a more computational flavor.

programming paradigm	declarative	functional
complexity classes	Descriptive Complexity	Implicit Computational Complexity
automata theory	subsystems of MSO	**this paper** (and planned sequels)

To our knowledge, very few works have looked at this kind of "type-theoretic" or "proof-theoretic" ICC for automata. Let us mention a few recent papers (that we will discuss further in §7) concerning transducers [13, 10] and multi-head automata [46, 28] and, most importantly, a remarkable result from 1996 that provides our starting point:

▶ **Theorem 1.1** (Hillebrand & Kanellakis [24, Theorem 3.4]). *A language $L \subseteq \Sigma^*$ can be defined in the* simply typed λ-calculus *by some* closed λ-term *of type* $\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$ *for some type A (that may depend on L) if and only if it is a* regular language.

Let us explain this statement. We consider a grammar of simple types with a single base type: $A, B ::= o \mid A \to B$, and use the *Church encodings* of booleans and strings:

$$\mathtt{Bool} = o \to o \to o \qquad\qquad \mathtt{Str}_\Sigma = (o \to o) \to \ldots \to (o \to o) \to o \to o$$

with $|\Sigma|$ arguments of type $(o \to o)$, where Σ is a finite alphabet. Moreover, given any other chosen type A, one can form the type $\mathtt{Str}_\Sigma[A]$ by substituting A for the ground type o:

▶ **Notation 1.2.** For types A and B, we denote by $B[A]$ the substitution $B\{o := A\}$ of every occurrence of o in B by A.

Every closed λ-term t of type $\mathtt{Str}_\Sigma$ can also be seen as a term of type $\mathtt{Str}_\Sigma[A]$. (This is a way to simulate a modicum of parametric polymorphism in a monomorphic type system.) It follows that any closed λ-term of type $\mathtt{Str}_\Gamma[A] \to \mathtt{Bool}$ in the simply typed λ-calculus defines a predicate on strings, i.e. a language $L \subseteq \Sigma^*$.

Although little-known[1], Hillebrand and Kanellakis's theorem should not be surprising in retrospect: there are strong connections between Church encodings and automata (see e.g. [45, 48, 34]), that have been exploited in particular in *higher-order model checking* for the past 15 years [2, 38, 25, 21, 23, 49]. This is not a mere contrivance: these encodings have been a canonical data representation for λ-calculi for much longer[2].

Star-free languages. We would like to extend this result by characterizing strict subclasses of regular languages, the most famous being the *star-free languages*. Recall that the canonicity of the class of regular languages is firmly established by its various definitions: regular expressions, finite automata, definability in MSO and the algebraic characterization.

▶ **Theorem 1.3** (cf. [44, §II.2.]). *A language $L \subseteq \Sigma^*$ is regular if and only if for some* finite monoid M*, some subset $P \subseteq M$ and some* monoid morphism $\varphi \in \mathrm{Hom}(\Sigma^*, M)$*, $L = \varphi^{-1}(P)$.*

Similarly, the seminal work of Schützenberger, Petrone, McNaughton and Papert in the 1960s (see [47] for a historical discussion) has led to many equivalent definitions for star-free languages, with the algebraic notion of *aperiodicity* playing a key role:

[1] See e.g. Damiano Mazza's answer to this MathOverflow question: `https://mathoverflow.net/q/296879`
[2] They were introduced for booleans and integers by Church in the 1930s, and later generalized by Böhm and Berarducci [12], see also `http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html`. (Similar ideas appear around the same time in [31].) As for the refined encodings with linear types that we use later, they already appear in Girard's founding paper on linear logic [17, §5.3.3].

▶ **Definition 1.4.** *A monoid M is* aperiodic *when any sequence of iterated powers is eventually constant, i.e. for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.*

▶ **Theorem 1.5** (cf. [47])**.** *For a language $L \subseteq \Sigma^*$, the following conditions are equivalent:*

- *L is defined by some* star-free regular expression: *$E, E' ::= \varnothing \mid \{a\} \mid E \cup E' \mid E \cdot E' \mid E^c$ where a can be any letter in Σ and E^c denotes the* complement *of E ($[\![E^c]\!] = \Sigma^* \setminus [\![E]\!]$);*
- *$L = \varphi^{-1}(P)$ for some finite and* aperiodic *monoid M, some subset $P \subseteq M$ and some monoid morphism $\varphi \in \mathrm{Hom}(\Sigma^*, M)$;*
- *L is recognized by a deterministic finite automaton whose transition monoid is* aperiodic*;*
- *L is definable in first-order logic.*

Attempting to capture star-free languages in a λ-calculus presents a serious methodological challenge: they form a strict subclass of uniform AC^0, and, as far as we know, type-theoretic ICC has never managed before to characterize complexity classes as small as this.

Non-commutative affine types. Monoids appear in typed λ-calculi when one looks at the functions from a type A to itself, i.e. at the (closed) terms of type $A \to A$. At first glance, it seems difficult indeed to enforce the aperiodicity of such monoids via a type system. For instance, one needs to rule out $\mathtt{not} = \lambda b.\, \lambda x.\, \lambda y.\, b\, y\, x : \mathtt{Bool} \to \mathtt{Bool}$ since it "has period two": its iteration yields the sequence (modulo $\beta\eta$-conversion) $\mathtt{not}, \mathtt{id}, \mathtt{not}, \mathtt{id}, \ldots$ (where $\mathtt{id} = \lambda b.\, b$) which is not eventually constant. Observe that $\mathtt{not}$ essentially *exchanges* the two arguments of b; to exclude it, we are therefore led to require functions to *use their arguments in the same order that they are given in.*

It is well-known that in order to make such a *non-commutative* λ-calculus work – in particular to ensure that non-commutative λ-terms are closed under β-reduction – one needs to make the type system *affine*, that is, to restrict the duplication of data. This is achieved by considering a type system based on Girard's linear[3] logic [17], a system whose "resource-sensitive" nature has been previously exploited in ICC [20, 19]. Not coincidentally, the theme of non-commutativity first appeared in a form of linear logic *ante litteram*, namely the Lambek calculus [29], and resurfaced shortly after the official birth of linear logic: it is already mentioned by Girard in a 1987 colloquium [18].

We shall therefore introduce and use a variant of Polakow and Pfenning's Intuitionistic Non-Commutative Linear Logic [39, 40], making a distinction between two kinds of function arrows: $A \multimap B$ and $A \to B$ are, respectively, the types of affine functions and non-affine functions from A to B. Accordingly:

▶ **Definition 1.6.** *A type is said to be* purely affine *if it does not contain the "$\to$" connective.*

In our system that we call the *$\lambda\wp$-calculus*, the types of Church encodings become

$$\mathtt{Bool} = o \multimap o \multimap o \qquad\qquad \mathtt{Str}_\Sigma = (o \multimap o) \to \ldots \to (o \multimap o) \to (o \multimap o)$$

where $\mathtt{Str}_\Sigma$ has $|\Sigma|$ arguments[4] of type $(o \multimap o)$. Setting $\mathtt{true} = \lambda^\circ x.\, \lambda^\circ y.\, x : \mathtt{Bool}$ and $\mathtt{false} = \lambda^\circ x.\, \lambda^\circ y.\, y : \mathtt{Bool}$ for the rest of the paper, we can now state our main result:

▶ **Theorem 1.7.** *A language $L \subseteq \Sigma^*$ is* star-free *if and only if it can be defined by a closed $\lambda\wp$-term of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ for some* purely affine *type A (that may depend on L).*

[3] The main difference between so-called linear and affine type systems is that the latter allow *weakening*, that is, to not use some argument. Typically, $\lambda x.\, \lambda y.\, x$ is affine but not linear while $\lambda x.\, x\, x$ is neither linear nor affine. The type system that we use in this paper is affine, not strictly linear.

[4] $o \multimap o$ occurs $|\Sigma| + 1$ times in $\mathtt{Str}_\Sigma$: $|\Sigma|$ arguments plus the output.

However, if we use the *commutative* variant of the $\lambda\wp$-calculus instead, then what we get is the class of regular languages (Theorem 5.1), just as in Hillebrand and Kanellakis's theorem.

As far as we know, non-commutative type systems have never been applied to implicit complexity before (but they have been used to control the expressivity of a domain-specific programming language [26]). Previous works indeed tend to see non-commutative λ-terms (or proof nets) as static objects, and to focus on their topological aspects (e.g. [6, 51, 35]), though there is another tradition relating self-dual non-commutativity to process algebras[5] [41, 22].

Proof strategy. As usual in implicit computational complexity, the proof of Theorem 1.7 consists of a *soundness* part – "every $\lambda\wp$-definable language is star-free" – and an *extensional completeness* part – the converse implication. In our case, soundness is a corollary of the following property of the purely affine fragment of the $\lambda\wp$-calculus – what one might call the *planar*[6] *affine λ-calculus* (cf. [1, 51]):

▶ **Theorem 1.8** (proved in §3). *For any* purely affine *type A, the set of closed $\lambda\wp$-terms of type $A \multimap A$, quotiented by $\beta\eta$-convertibility and endowed with function composition $(f \circ g = \lambda^{\circ} x.\, f\,(g\,x))$, is a* finite and aperiodic monoid.

Extensional completeness turns out here to be somewhat deeper than the "programming exercise of limited theoretical interest" [33, p. 137] that one generally finds in ICC. Indeed, we have only managed to encode star-free languages in the $\lambda\wp$-calculus by relying on a powerful tool from semigroup theory: the *Krohn–Rhodes decomposition* [27].

Plan of the paper. After having defined the $\lambda\wp$-calculus in §2, we prove Theorem 1.7: soundness is treated in §3 and extensional completeness in §4. Then we discuss the analogous results for the commutative variant of the $\lambda\wp$-calculus and its extension with additives (§5), our plans for the next papers in the series (§6) and finally some related work (§7).

Prerequisites. We assume that the reader is familiar with the basics of λ-calculi and type systems, but require no prior knowledge of automata theory. This choice is motivated by the impression that it is more difficult to introduce the former than the latter in a limited number of pages. Nevertheless, we hope that our results will be of interest to both communities.

2 Preliminaries: the $\lambda\wp$-calculus and Church encodings

The terms and types of the $\lambda\wp$-calculus are defined by the respective grammars

$$A, B ::= o \mid A \to B \mid A \multimap B \qquad t, u ::= x \mid t\,u \mid \lambda^{\to} x.\, t \mid \lambda^{\circ} x.\, t$$

As always, the $\lambda\wp$ terms are identified up to α-equivalence (both $\lambda^{\to}$ and λ° are binders). There are two rules for β-reduction (closed under contexts)

$$(\lambda^{\to} x.\, t)\,u \longrightarrow_{\beta} t\{x := u\} \qquad (\lambda^{\circ} x.\, t)\,u \longrightarrow_{\beta} t\{x := u\}$$

[5] This connection with the sequential composition of processes can be seen as a sort of embodiment of Girard's slogan "time is the contents of non-commutative linear logic" [18, IV.6]. But generally, these works follow a "proof search as computation" paradigm (logic programming) rather than "normalization as computation" (functional programming).

[6] Hence our choice of name: the "Weierstraß P" character "$\wp$" in "$\lambda\wp$" stands for "planar".

$$\frac{}{\Gamma \uplus \{x : A\} \mid \varnothing \vdash x : A} \qquad \frac{\Gamma \mid \Delta \vdash t : A \to B \qquad \Gamma \mid \varnothing \vdash u : A}{\Gamma \mid \Delta \vdash t\,u : B} \qquad \frac{\Gamma \uplus \{x : A\} \mid \Delta \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\to} x.\, t : A \to B}$$

$$\frac{}{\Gamma \mid x : A \vdash x : A} \qquad \frac{\Gamma \mid \Delta \vdash t : A \multimap B \qquad \Gamma \mid \Delta' \vdash u : A}{\Gamma \mid \Delta \cdot \Delta' \vdash t\,u : B} \qquad \frac{\Gamma \mid \Delta \cdot (x : A) \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\circ} x.\, t : A \multimap B}$$

$$\frac{\Gamma \mid \Delta \vdash t : A}{\Gamma \mid \Delta' \vdash t : A} \text{ when } \Delta \text{ is a subsequence of } \Delta'$$

Figure 1 The typing rules of the $\lambda\wp$-calculus (see Appendix C in [37] for examples of derivations).

and the remaining conversion rules are the expected η-reduction/η-expansion rules.

The typing judgements make use of dual contexts (a common feature originating in [7]): they are of the form $\Gamma \mid \Delta \vdash t : A$ where t is a term, A is a type, Γ is a set of bindings of the form $x : B$ (x being a variable and B a type), and Δ is an *ordered list* of bindings – this order is essential for non-commutativity. The typing rules are given in Figure 1, where $\Delta \cdot \Delta'$ denotes the *concatenation of the ordered lists* Δ *and* Δ'. For both $\Gamma, \Gamma', \ldots$ and $\Delta, \Delta', \ldots$ we require each variable to appear at most once on the left of a colon.

▶ Remark 2.1. Unlike Polakow and Pfenning's system [39, 40], the $\lambda\wp$-calculus:

- contains two function types instead of four[7], with the top two rows of Figure 1 corresponding almost exactly[8] to the rules given for those connectives in [39];
- is affine instead of linear, as expressed by the "ordered weakening" rule at the bottom of Figure 1 – this seems important to get enough expressive power for our purposes[9].

▶ Remark 2.2. Morally, the non-affine variables "commute with everything". More formally, one could translate the $\lambda\wp$-calculus into a non-commutative version of Intuitionistic Affine Logic whose exponential modality "!" incorporates the customary rules (see e.g. [50])

$$\frac{\Gamma, !A, B, \Delta \vdash C}{\Gamma, B, !A, \Delta \vdash C} \qquad \qquad \frac{\Gamma, B, !A, \Delta \vdash C}{\Gamma, !A, B, \Delta \vdash C}$$

▶ **Proposition 2.3.** *The $\lambda\wp$-calculus enjoys subject reduction and admits normal forms (that is, every well-typed $\lambda\wp$-term is convertible to a β-normal η-long one).*

Proof sketch. This is routine: subject reduction follows from a case analysis, while the fact that the simply typed λ-calculus has normal forms entails that the $\lambda\wp$-calculus also does (the obvious translation preserves the β-reduction and η-expansion relations). ◀

We have already seen the type $\mathtt{Str}_\Sigma = (o \multimap o) \to \ldots \to (o \multimap o) \to (o \multimap o)$ of Church-encoded strings in the introduction. Let us now introduce the term-level encodings:

[7] Our "$\to$" and "$\multimap$" are called "intuitionistic functions" and "right ordered functions" in [39]; we have no counterpart for the "linear [commutative] functions" and "left ordered functions" in the $\lambda\wp$-calculus.
[8] The only difference is that we drop the linear commutative context.
[9] Usually, the linear/affine distinction does not matter for implicit computational complexity if we allow collecting the garbage produced during the computation in a designated part of the output, as in e.g. [30]. But non-commutativity obstructs the free movement of garbage.

▶ **Definition 2.4.** *Let Σ be a finite alphabet, $w = w[1] \dots w[n] \in \Sigma^*$ be a string, and for each $c \in \Sigma$, let t_c be a $\lambda\wp$-term (on which the next proposition will add typing assumptions). We abbreviate $(t_c)_{c\in\Sigma}$ as $\vec{t}_\Sigma$, and define the $\lambda\wp$-term $w^\dagger(\vec{t}_\Sigma) = \lambda^\circ x.\, t_{w[1]}\,(\dots\,(t_{w[n]}\,x)\dots)$.*

Given a total order $c_1 < \dots < c_{|\Sigma|}$ on the alphabet $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$, the Church encoding *of any string $w \in \Sigma^*$ is $\overline{w} = \lambda^{\to} f_{c_1}.\ \dots\ \lambda^{\to} f_{c_{|\Sigma|}}.\, w^\dagger(\vec{f}_\Sigma)$.*

This is simpler than the notation might suggest: as an example, for $\Sigma = \{a, b\}$ with $a < b$, $\overline{baa} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^\circ x.\ f_b\ (f_a\ (f_a\ x))$. Our choice of presentation is meant to stress the role of the *open* subterm $(baa)^\dagger(\vec{f}_{\{a,b\}}) = \lambda^\circ x.\ f_b\ (f_a\ (f_a\ x))$, cf. Remark 2.9.

We now summarize the classical properties of the Church encoding of strings.

▶ **Proposition 2.5.** *We reuse the notations of the above definition.*

- *Assume that there is a type A and a typing context $\Gamma \mid \Delta$ such that for all $c \in \Sigma$, $\Gamma \mid \Delta \vdash t_c : A \multimap A$. Then $\Gamma \mid \Delta \vdash w^\dagger(\vec{t}_\Sigma) : A \multimap A$.*
- *In particular, $\{f_c : o \multimap o \mid c \in \Sigma\} \mid \varnothing \vdash w^\dagger(\vec{f}_\Sigma) : o \multimap o$ for any* variables *$(f_c)_{c\in\Sigma}$.*
- *Furthermore, in the case of variables, $w \in \Sigma^* \mapsto w^\dagger(\vec{f}_\Sigma)$ is in fact a* bijection *between the strings over Σ and the $\lambda\wp$-terms u such that $\{f_c : o \multimap o \mid c \in \Sigma\} \mid \varnothing \vdash u : o \multimap o$ and considered up to $\beta\eta$-conversion*[10].
- *It follows from the above that $w \in \Sigma^* \mapsto \overline{w}$ is a bijection from Σ^* to the set of* closed *$\lambda\wp$-terms of type $\mathtt{Str}_\Sigma$ modulo $\beta\eta$.*
- *Finally, with the assumptions on t_c of the first item, we have $\overline{w}\, t_{c_1}\ \dots\ t_{c_{|\Sigma|}} \longrightarrow^*_\beta w^\dagger(\vec{t}_\Sigma)$.*

▶ **Example 2.6.** Given two closed $\lambda\wp$-terms $t_a, t_b : \mathtt{Bool} \multimap \mathtt{Bool}$, one can define the term $g = \lambda^\circ s.\ s\ t_a\ t_b\ \mathtt{false} : \mathtt{Str}_{\{a,b\}}[\mathtt{Bool}] \multimap \mathtt{Bool}$. Then for any $w = w[1] \dots w[n] \in \{a, b\}^*$, we have $g\,\overline{w} \longrightarrow^*_\beta w^\dagger(\vec{t}_{\{a,b\}})\ \mathtt{false} \longrightarrow^*_\beta t_{w[1]}\ (\dots\ (t_{w[n]}\ \mathtt{false}))$.

- For $t_a = \lambda^\circ x.\ \mathtt{true}$ and $t_b = \lambda^\circ x.\ x$, g decides the language of words in $\{a, b\}^*$ that contain at least one a; this language is indeed star-free as it can be expressed as $\varnothing^c a \varnothing^c$.
- Coming back to a point raised in the introduction, if negation were definable by a $\lambda\wp$-term $\mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$, then for $t_a = t_b = \mathtt{not}$, the language decided by g would consist of words of odd length: a standard example of regular language that is not star-free.

▶ Remark 2.7. Actually, the $\lambda\wp$-term $\mathtt{not}' : \lambda^\circ b.\ b\ \mathtt{false}\ \mathtt{true} : \mathtt{Bool}[\mathtt{Bool}] \multimap \mathtt{Bool}$ does "define negation". A point of utmost importance is that because of the heterogeneity of the input and output types, this term does not contradict Theorem 1.8 and *cannot be iterated by a Church-encoded string.* Monomorphism is therefore crucial for us: if our type system had actual polymorphism, one could give $\mathtt{not}'$ the type $(\forall\alpha.\ \mathtt{Bool}[\alpha]) \multimap (\forall\alpha.\ \mathtt{Bool}[\alpha])$, whose input and output types are equal, and then the words of odd length would be $\lambda\wp$-definable.

An analogous phenomenon in the simply typed λ-calculus is that one can define $n \mapsto 2^n$ on the type of Church numerals $\mathtt{Nat}$ by a term of type $\mathtt{Nat}[o \to o] \to \mathtt{Nat}$, but not by a term of type $\mathtt{Nat} \to \mathtt{Nat}$ (since iterating it would give rise to a tower of exponentials of variable height, which is known to be inexpressible by any $\mathtt{Nat}[A] \to \mathtt{Nat}$).

Yet our ersatz of polymorphism still allows for some form of compositionality that will prove useful in several places in §4 (the proof may be found in Appendix B in [37]):

▶ **Lemma 2.8.** *If $\vdash t : A[T] \multimap B$ and $\vdash u : B[U] \multimap C$, then $\vdash \lambda^\circ x.\, u\,(t\,x) : A[T[U]] \multimap C$.*

▶ Remark 2.9. One final observation on Church encodings: when the context Γ of non-affine variables contains $f_c : o \multimap o$ for each $c \in \Sigma$, then any string $w \in \Sigma^*$ can be represented as

[10] η-conversion is necessary to identify $\lambda^{\to} f.\, f : \mathtt{Str}_{\{a\}}$ with $\overline{a} = \lambda^{\to} f.\ \lambda^\circ x.\ f\ x : \mathtt{Str}_{\{a\}}$.

the open $\lambda\wp$-term $\Gamma \mid \ldots \vdash w^\dagger(\vec{f_\Sigma}) : o \multimap o$ in that context, and such strings can even be concatenated by function composition. The point is that this gives us a kind of *purely affine type of strings*, which will allow us in §4.2 to encode sequential transducers as $\lambda\wp$-terms of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$ for some purely affine type A (compare Theorem 1.7).

3 Proof of soundness

As stated in the introduction, the soundness part of our main Theorem 1.7 will follow from Theorem 1.8, so we start this section by proving the latter. First, the monoid structure on the closed $\lambda\wp$-terms of *any* type $A \multimap A$ can be verified routinely: both $(f \circ g) \circ h$ and $f \circ (g \circ h)$ β-reduce to $\lambda^\circ x.\, f\ (g\ (h\ x))$, and $\lambda^\circ x.\, x$ provides the identity element. The finiteness of this monoid for A *purely affine* comes from a slightly more general statement:

▶ **Proposition 3.1.** *For any purely affine type B, there are finitely many $\beta\eta$-equivalence classes of closed $\lambda\wp$-terms of type B.*

Proof. This is a well-known property of affine type systems: here, non-commutativity plays no role. We provide a proof in Appendix B in [37]. ◀

The substantial part of Theorem 1.8 is the *aperiodicity* of this monoid. It is here that non-commutativity comes into play. Morally, it is a kind of monotonicity condition that $\lambda\wp$-terms obey. A first idea would therefore be to seek to exploit the fact that the monoid of monotone functions on an ordered set is aperiodic. What we end up using is closely related:

▶ **Lemma 3.2.** *For any $k \in \mathbb{N}$, the monoid of* partial non-decreasing *functions from $\{1, \ldots, k\}$ to itself (endowed with usual function composition) is aperiodic.*

Proof. Let $f : \{1, \ldots, k\} \rightharpoonup \{1, \ldots, k\}$ be non-decreasing. For any $i \in \{1, \ldots, k\}$, the sequence $(f^n(i))_{n\in\mathbb{N}}$ is either non-increasing or non-decreasing as long as it is defined (depending on whether $i \geq f(i)$ or $i \leq f(i)$); so at some $n = N_i$, either it becomes undefined or it reaches a fixed point of f. By taking $N = \max_{1\leq i\leq k} N_i$, we have $f^N = f^{N+1}$. ◀

This underlies the proof of the key lemma below, that allows one to reduce the aperiodicity of some $t : A \multimap A$ to the aperiodicity of $\lambda\wp$-terms at smaller types.

▶ **Notation 3.3.** $\Delta \vdash t : A$ is an abbreviation for $\varnothing \mid \Delta \vdash t : A$ (indeed, the context of non-affine variables will be generally empty in our proof).

▶ **Notation 3.4.** Let $u_1, \ldots, u_k$ and $v_1, \ldots, v_l$ be $\lambda\wp$-terms. The notation $\vec{v}[\vec{y} := \vec{u}]$ denotes the componentwise parallel substitution $(v_i[y_1 := u_1, \ldots, y_k := u_k])_{1\leq i\leq l}$.

▶ **Lemma 3.5.** *Let $t = \lambda^\circ x.\, \lambda^\circ y_1. \ldots \lambda^\circ y_m.\, x\, u_1 \ldots u_k$ be a well-typed closed $\lambda\wp$-term of type $A \multimap A$ in* η-long form*, so that $x : A,\ y_1 : B_1,\ \ldots,\ y_k : B_k \vdash x\, u_1 \ldots u_k : o$ with $A = B_1 \multimap \ldots \multimap B_k \multimap o$. Then:*

- *$t^n = t \circ \ldots \circ t$ (n times) is β-convertible to $\lambda^\circ x.\, \lambda^\circ y_1. \ldots \lambda^\circ y_k.\, x\, u_1^{(n)} \ldots u_k^{(n)}$ where $\vec{u}^{(0)} = (y_1, \ldots, y_k)$, $\vec{u}^{(n+1)} = \vec{u}^{(n)}[\vec{y} := \vec{u}]$;*
- *For large enough $n \in \mathbb{N}$, each $u_i^{(n+1)}$ depends only on $u_i^{(n)}$ for the same $i \in \{1, \ldots, k\}$. More precisely, there exists $N \in \mathbb{N}$ such that for all $i \in \{1, \ldots, k\}$ there exists a well-typed closed $\lambda\wp$-term $t'_i : B_i \multimap B_i$ such that for all $n \geq N$, $u_i^{(n+1)} = t'_i\ u_i^{(n)}$.*

Proof. The first item is established by induction: abbreviating $\lambda^{\circ} y_1. \dots \lambda^{\circ} y_k.$ as $\lambda^{\circ}\vec{y}.$,

$$\begin{aligned} t \circ (\lambda^{\circ} x. \lambda^{\circ}\vec{y}. x\, u_1^{(n)} \dots u_k^{(n)}) &=_\beta \lambda^{\circ} x. \lambda^{\circ}\vec{y}. (\lambda^{\circ}\vec{y}. x\, u_1^{(n)} \dots u_k^{(n)})\, u_1 \dots u_k \\ &=_\beta \lambda^{\circ} x. \lambda^{\circ}\vec{y}. x\, (u_1^{(n)}[\vec{y} := \vec{u}]) \dots u_k^{(n)}([\vec{y} := \vec{u}]) \end{aligned}$$

(We invite to reader to reproduce the full computation to check that no spurious capture of free variables happens.)

For the second item, let us define the partial function $\mu_{\vec{u}} : \{1, \dots, k\} \rightharpoonup \{1, \dots, k\}$ by $\mu_{\vec{u}}(i) = j \iff y_i \in \mathrm{FV}(u_j)$. ($\mathrm{FV}(u)$ denotes the set of free variables of u.) The relation on the right-hand side of the equivalence is indeed a partial function because of the affineness of $t = \lambda^{\circ} x. \lambda^{\circ} y_1. \dots \lambda^{\circ} y_k. z\, u_1 \dots u_k$. One can also show that for all $n \in \mathbb{N}$, $\mathrm{FV}(u_i^{(n)}) = \{y_j \mid (\mu_{\vec{u}})^n(j) = i\}$.

As a consequence of non-commutativity, $\mu_{\vec{u}}$ is non-decreasing. This is because for the typing judgment on $x\, u_1 \dots u_k$ to hold, there must exist $\Delta_1, \dots, \Delta_k$ such that:

- for all $j \in \{1, \dots, k\}$, $\Delta_j \vdash u_j$ and $\forall i,\ y_i \in \mathrm{FV}(u_j) \iff (y_i : B_i) \in \Delta_j$;
- $\Delta_1 \cdot \dots \cdot \Delta_k$ is an *ordered subsequence* of $(y_1 : B_1) \cdot \dots \cdot (y_k : B_k)$.

Therefore, by Lemma 3.2, there exists $N \in \mathbb{N}$ such that $(\mu_{\vec{u}})^N = (\mu_{\vec{u}})^{N+1}$.

Next, let $i \in \{1, \dots, k\}$. We may reformulate our goal as finding $t'_i : B_i \multimap B_i$ such that $t'_i\, u_i^{(N+n)} =_{\beta\eta} u^{(N+n+1)}$ for all $n \in \mathbb{N}$. The simple case is when $i \notin (\mu_{\vec{u}})^N(\{1, \dots, k\})$: $u_i^{(N)}$ has no free variables, so $u_i^{(N+1)} = u_i^{(N)}[\vec{y} := \vec{u}] = u_i^{(N)}$: we may then take $t'_i = \lambda^{\circ} z.\, z$. For the remainder of the proof we assume otherwise, that is, we take i in the range of $(\mu_{\vec{u}})^N$.

First, $\vec{u}^{(n+1)} = \vec{u}[\vec{y} := \vec{u}^{(n)}]$ because parallel substitution is associative[11]. Thus,

$$\forall n \in \mathbb{N},\ u_i^{(N+n+1)} = u_i \left[y_j := u_j^{(N+n)} \text{ for } j \in \{1, \dots, k\} \text{ such that } \mu_{\vec{u}}(j) = i \right]$$

Any $j \in \{1, \dots, k\} \setminus \{i\}$ such that $\mu_{\vec{u}}(j) = i$ is not a fixed point of $\mu_{\vec{u}}$, and therefore is not in the range of $(\mu_{\vec{u}})^N$ since $(\mu_{\vec{u}})^N = (\mu_{\vec{u}})^{N+1} = \mu_{\vec{u}} \circ (\mu_{\vec{u}})^N$. By the simple case already treated, we then have $u_j^{(N+n)} = u_j^{(N)}$. This allows us to write the above equation as

$$u_i^{(N+n+1)} = r_i[y_i := u_i^{(N+n)}] \quad \text{where} \quad r_i = u_i \left[y_j := u_j^{(N)} \text{ for } j \neq i \text{ s.t. } \mu_{\vec{u}}(j) = i \right]$$

Using β-conversion, $u_i^{(N+n+1)} =_\beta (\lambda^{\circ} y_i. r_i)\, u_i^{(N+n)}$. We conclude by setting $t'_i = (\lambda^{\circ} y_i. r_i)$. It is clear that this $\lambda\wp$-term is closed, but one should check that it is well-typed; to do so, one convenient observation is that the $u_j^{(N)}$ are closed (because $j \notin (\mu_{\vec{u}})^N(\{1, \dots, k\})$) and well-typed (as closed subterms of a reduct of the N-fold composition t^N). ◀

The remainder of the proof of Theorem 1.8 is essentially bureaucratic.

Proof of the aperiodicity part of Theorem 1.8. Let $t : A \multimap A$; our goal is to show that the sequence $t^n = t \circ \dots \circ t$ is eventually constant modulo $\beta\eta$. We shall do so by *induction on the size of A*. The type A is *purely affine* by assumption, and can therefore be written as $B_1 \multimap \dots \multimap B_m \multimap o$ where the B_i are also purely affine for $i \in \{1, \dots, m\}$. The base case $m = 0$ being trivial, we assume $m \geq 1$. In this case, by Proposition 2.3, t has an η-long β-normal form $t = \lambda^{\circ} x. \lambda^{\circ} y_1. \dots \lambda^{\circ} y_m. z\, u_1 \dots u_k$ where z is a variable. There are two cases:

- $z = y_i$ for some i. Then $(y_i : B_i) \cdot \Delta \vdash z\, u_1 \dots u_k$ by application rule (we omit the non-affine context Γ which will always be empty during this proof). The abstraction rule only allows introducing $\lambda^{\circ} y_i$ when $(y_i : B_i)$ is on the right, so by then Δ must have been

[11] More precisely, $(\vec{t_1}[\vec{x} := \vec{t_2}])[\vec{y} := \vec{t_3}] = \vec{t_1}[\vec{x} := \vec{t_2}[\vec{y} := \vec{t_3}]]$ when $\vec{y} \cap (\mathrm{FV}(\vec{t_1}) \setminus \vec{x}) = \varnothing$.

entirely emptied out by previous abstractions. This means that $\lambda^\circ y_i. \ldots \lambda^\circ y_m. z\, u_1 \ldots u_k$ is a closed term, so in particular it contains no free occurrence of x: t is a constant function from A to A. So the sequence of iterations stabilizes from $n = 1$.

- $z = x$, which entails $k = m$ since the variable x is of type $A = B_1 \multimap \ldots \multimap B_m \multimap o$ and we must have $x : A,\ y_1 : B_1,\ \ldots,\ y_m : B_m \vdash x\, u_1 \ldots u_k : o$. Lemma 3.5 gives us closed $\lambda\wp$-terms $t'_i : B_i \multimap B_i$ $(i \in \{1, \ldots, k\})$ whose iterates eventually determine those of t. Since the type B_i has size strictly smaller than A, the induction hypothesis applies: each $((t'_i)^n)_{n \in \mathbb{N}}$ is eventually constant modulo $\beta\eta$. Therefore, this is also the case for t. ◀

Let us now apply Theorem 1.8 to the $\lambda\wp$-terms defining languages.

▶ **Lemma 3.6.** *Let $\Sigma = \{c_1, \ldots, c_{|\Sigma|}\}$ be a finite alphabet, A be a* purely affine *type and $t : \mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ be a closed $\lambda\wp$-term. Then there exist some closed $\lambda\wp$-terms $g_c : A \multimap A$ for $c \in \Gamma$ and $h : (A \multimap A) \multimap \mathtt{Bool}$ such that $t =_{\beta\eta} \lambda^\circ s.\, h\, (s\, g_{c_1} \ldots g_{c_{|\Sigma|}})$.*

Proof. By inspection of the normal form of t, see Appendix B in [37]. ◀

Reusing the notations of this lemma, let us define $\varphi : \Sigma^* \to \{v \mid v : A \multimap A\}/{=_{\beta\eta}}$ to be the monoid morphism such that $\varphi(c) = g_c$ for $c \in \Sigma$. Then for all $w \in \Sigma^*$, $\varphi(w) = w^\dagger(\vec{g}_\Sigma)$ (in the quotient): by a similar computation than for $f \circ (g \circ h) =_{\beta\eta} (f \circ g) \circ h$, we have $g_{w[1]} \circ \ldots \circ g_{w[n]} \longrightarrow^*_\beta w^\dagger(\vec{g}_\Sigma)$. Therefore, by Proposition 2.5, $\varphi^{-1}(\{v \mid h\, v =_{\beta\eta} \mathtt{true}\})$ is none other than the language defined by the $t : \mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ in the lemma. Thus, L fits the second definition of star-free languages given in Theorem 1.5: indeed, the codomain of φ is finite and aperiodic by Theorem 1.8. This proves the soundness part of Theorem 1.7.

4 Expressiveness of the $\lambda\wp$-calculus

We now turn to the *extensional completeness* part in Theorem 1.7: our goal is to construct, for any star-free language, a closed $\lambda\wp$-term of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ (for some purely affine A) that defines this language. To do so, the most convenient way that we have found is to take a detour through automata that compute an output string instead of a single bit (acceptance/rejection). We will recall the notion of *aperiodic sequential function* (Definition 4.4), and then establish that:

▶ **Theorem 4.1.** *Any* aperiodic sequential function $\Sigma^* \to \Pi^*$ *can be expressed by a $\lambda\wp$-term of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$ for some* purely affine *type A.*

The advantage of working with this class of functions is that they can be assembled from small "building blocks" by function composition, as the *Krohn–Rhodes decomposition* (Theorem 4.8) tells us. Our proof strategy for the above theorem will consist in encoding these blocks (Lemma 4.10) and composing them together (as a special case of Lemma 2.8).

To deduce the desired result, we rely on two lemmas (proved in Appendix B in [37]):

▶ **Lemma 4.2.** *If a language $L \subseteq \Sigma^*$ is* star-free, *then its* (string-valued) indicator function *$\chi_L : \Sigma^* \to \{1\}^*$, defined by $\chi_L(w) = 1$ if $w \in L$ and $\chi_L(w) = \varepsilon$ otherwise, is aperiodic sequential.*

▶ **Lemma 4.3.** *There exists a $\lambda\wp$-term $\mathtt{nonempty} : \mathtt{Str}_{\{1\}}[\mathtt{Bool}] \multimap \mathtt{Bool}$ that tests whether its input string is non-empty.*

Let L be a star-free language. Combining Lemma 4.2 and Theorem 4.1, χ_L is definable by some $\lambda\wp$-term $\mathtt{indic}_L : \mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_{\{1\}}$ where A is purely affine. To compose this with the non-emptiness test of Lemma 4.3, we use Lemma 2.8 again: the $\lambda\wp$-term

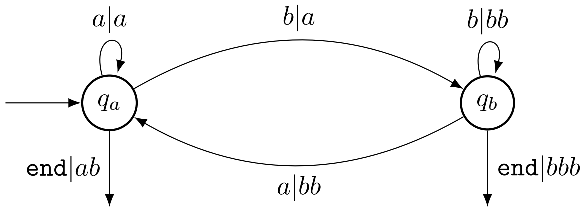

Figure 2 A schematic representation of a sequential transducer whose formal definition is $Q = \{q_a, q_b\}$, $\delta(q,a) = (q_a, a)$ and $\delta(q,b) = (q_b, bb)$ for $q \in Q$, $q_I = q_a$, $F(q_a) = ab$ and $F(q_b) = bbb$.

$t_L = \lambda^\circ x.\ \texttt{nonempty}\ (\texttt{indic}_L\ x) : \texttt{Str}_\Sigma[A[\texttt{Bool}]] \multimap \texttt{Bool}$ defines L. Since A and $\texttt{Bool}$ are purely affine, so is $A[\texttt{Bool}]$: we just deduced extensional completeness from Theorem 4.1. Proving the latter is the goal of the rest of this section.

4.1 Reminders on automata theory

Sequential transducers are among the simplest models of automata with output. They are deterministic finite automata which can append a word to their output at each transition, and at the end, they can add a suffix to the output depending on the final state. The definition is classical; a possible reference is [44, Chapter V].

▶ **Definition 4.4.** *A* sequential transducer *with input alphabet* Σ *and output alphabet* Π *consists of a set of* states Q*, a* transition function $\delta : Q \times \Sigma \to Q \times \Pi^*$*, an* initial state $q_I \in Q$*, and a* final output function $F : Q \to \Pi^*$*. We abbreviate* $\delta_i = \pi_i \circ \delta$ *for* $i \in \{1, 2\}$*, where* $\pi_1 : Q \times \Pi^* \to Q$ *and* $\pi_2 : Q \times \Pi^* \to \Pi^*$ *are the projections of the product.*

Given an input string $w = w[1] \dots w[n] \in \Sigma^*$*, the* run *of the transducer over* w *is the sequence of states* $q_0 = q_I$*,* $q_1 = \delta_{\mathrm{st}}(q_0, w[1])$*, …,* $q_n = \delta_{\mathrm{st}}(q_{n-1}, w[n])$*. Its* output *is obtained as the concatenation* $\delta_{\mathrm{out}}(q_0, w[1]) \cdot \ldots \cdot \delta_{\mathrm{out}}(q_{n-1}, w[n]) \cdot F(q_n)$*.*

A sequential function *is a function* $\Sigma^* \to \Pi^*$ *computed as described above by some sequential transducer.*

▶ **Definition 4.5.** *The* transition monoid *of a sequential transducer is the submonoid of* $Q \to Q$ *(endowed with reverse function composition:* $fg = g \circ f$*) generated by the maps* $\{\delta_{\mathrm{st}}(-, c) \mid c \in \Sigma\}$ *(where* $\delta_{\mathrm{st}}(-, c)$ *stands for* $q \mapsto \delta_{\mathrm{st}}(q, c)$*).*

A sequential transducer is said to be aperiodic *when its transition monoid is aperiodic. A function that can be computed by such a transducer is called an* aperiodic sequential function*.*

▶ **Example 4.6.** The transducer in Figure 2 computes $f : w \in \{a, b\}^* \mapsto a \cdot \psi(w) \cdot b$ where ψ is the monoid morphism that doubles every b: $\psi(a) = a$ and $\psi(b) = bb$. Its transition monoid T is generated by $G = \{(\delta_{\mathrm{st}}(-, a) : q \mapsto q_a), (\delta_{\mathrm{st}}(-, b) : q \mapsto q_b)\}$; one can verify that $T = G \cup \{\mathrm{id}\}$ and therefore $\forall h \in T,\ h \circ h = h$. Thus, f is an aperiodic sequential function.

▶ Remark 4.7. The converse to Lemma 4.2 is also true; more generally, the preimage of a star-free language by an aperiodic sequential function is star-free, and the preimage of a regular language is regular. But we will not need this here.

▶ **Theorem 4.8** (Krohn–Rhodes decomposition, aperiodic case, cf. Appendix A in [37])**.** *Any* aperiodic sequential function $f : \Sigma^* \to \Pi^*$ *can be realized as a composition* $f = f_1 \circ \ldots \circ f_n$ *(with* $f_i : \Xi_i^* \to \Xi_{i-1}^*$*,* $\Xi_0 = \Pi$ *and* $\Xi_n = \Sigma$*) where each function* f_i *is computed by some aperiodic sequential transducer* with 2 states.

Figure 2 gives an example of aperiodic transducer with two states.

▶ Remark 4.9. This is not the standard way to state this theorem, though one may find it in the literature, usually without proof (e.g. [10, §1.1]); see [8] for a tutorial containing a proof sketch of this version. In Appendix A in [37], we show how Theorem 4.8 follows from the more usual statement on wreath products of monoid actions.

4.2 Encoding aperiodic sequential transducers

Thanks to the Krohn–Rhodes decomposition and to the fact that the string functions definable in the $\lambda\wp$-calculus (as specified by Theorem 4.1) are closed under composition (by Lemma 2.8), the following entails Theorem 4.1, thus concluding our completeness proof.

▶ **Lemma 4.10.** *Any function* $\Sigma^* \to \Pi^*$ *computed by some aperiodic sequential transducer* with 2 states *can be expressed by some* $\lambda\wp$*-term of type* $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$*, for a* purely affine *type* A *depending on the function.*

Let us start by exposing the rough idea of the encoding's trick using set-theoretic maps. We reuse the notations of Definition 4.4 and assume w.l.o.g. that the set of states is $Q = \{1, 2\}$.

Suppose that at some point, after processing a prefix of the input, the transducer has arrived in state 1 (resp. 2) and in the meantime has outputted $w \in \Pi^*$. We can represent this "history" by the pair (κ_w, ζ) (resp. (ζ, κ_w)) where

$$\zeta, \kappa_w : \Pi^* \to \Pi^* \qquad \zeta : x \mapsto \varepsilon \qquad \kappa_w : x \mapsto w \cdot x$$

For instance, in the case of Example 4.6, after reading a string $s = s'b$, the transducer is in the state q_b and has outputted[12] $w = a \cdot \psi(s')$, which we represent as $(\zeta, \kappa_{a\cdot\psi(s')})$ (taking $q_a = 1$ and $q_b = 2$; ψ is described in Example 4.6). In general, some key observations are

$$\zeta \circ \kappa_w = \zeta \qquad \kappa_w \circ \kappa_{w'} = \kappa_{ww'} \qquad \kappa_w(w')\zeta(w'') = \zeta(w'')\kappa_w(w') = ww'$$

Now, consider an input letter $c \in \Sigma$; how to encode the corresponding transition $\delta(-, c)$ as a transformation on the pair encoding the current state and output history? It depends on the state transition $\delta_{\mathrm{st}}(-, c)$; we have thanks to the above identities:

- $(h, g) \mapsto (h \circ \kappa_{\delta_{\mathrm{out}}(1,c)}, g \circ \kappa_{\delta_{\mathrm{out}}(2,c)})$ when $\delta_{\mathrm{st}}(-, c) = \mathrm{id}$;
- $(h, g) \mapsto (\kappa_{h(\delta_{\mathrm{out}}(1,c))g(\delta_{\mathrm{out}}(2,c))}, \zeta)$ when $\delta_{\mathrm{st}}(-, c) : q' \mapsto 1$ (note that $h = \zeta$ xor $g = \zeta$);
- $(h, g) \mapsto (\zeta, \kappa_{h(\delta_{\mathrm{out}}(1,c))g(\delta_{\mathrm{out}}(2,c))})$ when $\delta_{\mathrm{st}}(-, c) : q' \mapsto 2$;
- The remaining case $\delta_{\mathrm{st}}(-, c) : q \mapsto 3 - q$ is *excluded by aperiodicity.* This point is crucial: this case would correspond to $(h, g) \mapsto (g \circ \kappa_{\delta_{\mathrm{out}}(2,c)}, h \circ \kappa_{\delta_{\mathrm{out}}(1,c)})$ which morally "uses its arguments h, g in the wrong order".

Coming back to Example 4.6, let us say that after the transducer has read a prefix $s = s'b$ of its input string as we previously described, the next letter is a. Then the expression $h(\delta_{\mathrm{out}}(1, c))g(\delta_{\mathrm{out}}(2, c))$ above is in this case $\zeta(a)\kappa_{a\cdot\psi(s')}(bb) = \varepsilon \cdot a \cdot \psi(s') \cdot bb = a \cdot \psi(s)$ which is indeed the output that the transducer produces after reading the input prefix $sa = s'ba$.

Next, we must transpose these ideas to the setting of the $\lambda\wp$-calculus.

[12] This is indeed $a \cdot \psi(s')$ and not $a \cdot \psi(s) = a \cdot \psi(s') \cdot bb$. If the input turns out to end there, the final output function will provide the missing suffix $F(q_b) = bbb$ to obtain $f(s) = a \cdot \psi(s) \cdot b = a \cdot \psi(s') \cdot bbb$.

Proof of Lemma 4.10. We define the $\lambda\wp$-term meant to compute our sequential function as

$$\lambda^{\circ} s.\, \lambda^{\to} f_{a_1}.\, \dots\, \lambda^{\to} f_{a_{|\Pi|}}.\, \texttt{out}\,(s\, \texttt{trans}_{c_1}\, \dots\, \texttt{trans}_{c_{|\Sigma|}}) : \texttt{Str}_{\Sigma}[A] \multimap \texttt{Str}_{\Pi}$$

where $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$, $\Pi = \{a_1, \dots, a_{|\Pi|}\}$ and, writing $\Gamma = \{f_a : o \multimap o \mid a \in \Pi\}$,

$$\Gamma \mid \varnothing \vdash \texttt{trans}_c : A \multimap A \quad (\text{for all } c \in \Sigma) \qquad \Gamma \mid \varnothing \vdash \texttt{out} : (A \multimap A) \multimap (o \multimap o)$$

In the presence of this non-affine context Γ, the type $S = o \multimap o$ morally serves as a purely affine type of strings, as mentioned in Remark 2.9. Moreover this "contextual encoding of strings" supports concatenation (by function composition), leading us to represent the maps ζ and κ_w as open terms of type $T = S \multimap S$ that use non-affinely the variables f_a for $a \in \Pi$.

We shall take the type A, at which the input $\texttt{Str}_{\Sigma}$ is instantiated, to be $A = T \multimap T \multimap S$, which is indeed purely affine as required by the theorem statement. This can be seen morally as a type of continuations [42] taking pairs of type $T \otimes T$ (although our $\lambda\wp$-calculus has no actual $\otimes$ connective). Without further ado, let us program (the typing derivations for some of the following $\lambda\wp$-terms are given in Appendix C in [37]):

- $\texttt{cat} = \lambda^{\circ} w.\, \lambda^{\circ} w'.\, \lambda^{\circ} x.\, w\,(w'\,x) : S \multimap S \multimap o \multimap o = S \multimap S \multimap S = S \multimap T$ plays the roles of both the concatenation operator and of $w \mapsto \kappa_w$ (thanks to currying)
- $\texttt{zeta} = \lambda^{\circ} w'.\, \lambda^{\circ} x.\, x : S \multimap o \multimap o = T$
- $u_q = \delta_{\text{out}}(q,c)^{\dagger}(\vec{f_\Pi}) : o \multimap o$ (by Proposition 2.5) represents the output word $\delta_{\text{out}}(q,c)$ that corresponds to a given input letter $c \in \Sigma$ and state $q \in Q = \{1,2\}$
- case $\delta_{\text{st}}(q,c) = q$: $\texttt{trans}_c = \lambda^{\circ} k.\, \lambda^{\circ} h.\, \lambda^{\circ} g.\, k\,(\lambda^{\circ} y.\, h\,(\texttt{cat}\, u_1\, y))\,(\lambda^{\circ} z.\, g\,(\texttt{cat}\, u_2\, z))$ – if we wanted to handle the excluded case $\delta_{\text{st}}(q,c) = 3 - q$, we would write a similar term with the occurrences of h and g exchanged $(\lambda^{\circ} k.\, \lambda^{\circ} h.\, \lambda^{\circ} g.\, k\,(\lambda^{\circ} y.\, g\, \dots)\,(\lambda^{\circ} z.\, h\, \dots))$, violating the non-commutativity requirement (contrast with the proof of Theorem 5.4);
- case $\delta_{\text{st}}(q,c) = 1$: $\texttt{trans}_c = \lambda^{\circ} k.\, \lambda^{\circ} h.\, \lambda^{\circ} g.\, k\,(\texttt{cat}\,(\texttt{cat}\,(h\, u_1)\,(g\, u_2)))\, \texttt{zeta}$
- case $\delta_{\text{st}}(q,c) = 2$: $\texttt{trans}_c = \lambda^{\circ} k.\, \lambda^{\circ} h.\, \lambda^{\circ} g.\, k\, \texttt{zeta}\,(\texttt{cat}\,(\texttt{cat}\,(h\, u_1)\,(g\, u_2)))$
- $\texttt{out} = \lambda^{\circ} j.\, j\,(\lambda^{\circ} h.\, \lambda^{\circ} g.\, \texttt{cat}\,(h\, v_1)\,(g\, v_2))\,(\lambda^{\circ} x.\, x)\, \texttt{zeta}$, where $v_q = F(q)^{\dagger}(\vec{f_\Pi})$ represents the output suffix for state $q \in \{1,2\}$, assuming w.l.o.g. that the initial state is 1 (also, here $\lambda^{\circ} x.\, x$ represents κ_ε since the latter is the identity on Π^*)

We leave it to the reader to check that these $\lambda\wp$-terms have the expected computational behavior; again, see Appendix C in [37] for typing derivations. Note that in functional programming terms, the use of continuations turns the "right fold" of the Church-encoded input string into a "left fold", and the latter fits with the left-to-right processing of a sequential transducer. ◀

5 Regular languages in extensions of the $\lambda\wp$-calculus

5.1 The commutative case

The $\lambda\wp$-calculus adds two restrictions to the simply typed λ-calculus, namely affineness and non-commutativity, with the latter depending on the former as already mentioned. One could wonder whether affineness by itself would be enough to characterize star-free languages. We now show that it is not the case.

The *commutative* variant of the $\lambda\wp$-calculus – let us call this variant the *λa-calculus*[13] – has the same grammar of types and terms as the $\lambda\wp$-calculus (cf. §2). The typing rules are also given by Figure 1, but their interpretation differs from the previous one as follows:

[13] *a* standing for "affine".

Δ, Δ' stand for *sets* of bindings $x : A$, $\Delta \cdot \Delta'$ denotes the *disjoint union* of sets, and one must read "subset" instead of "subsequence". In other words, the main difference is that in the λa-calculus, the affine context Δ does not keep track of the *ordering* of variables.

By plugging this commutative system in the statement of our main result (Theorem 1.7), we get *regular languages* instead of star-free languages:

▶ **Theorem 5.1.** *A language $L \subseteq \Sigma^*$ is* regular *if and only if it can be defined by a closed λa-term of type* $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ *for some purely affine type A (that may depend on L).*

Proof. Soundness is a consequence of Hillebrand and Kanellakis's Theorem 1.1, by a simple translation from the λa-calculus to the simply typed λ-calculus which "forgets affineness".

For extensional completeness, consider a regular language $L = \varphi^{-1}(P)$ where P is a subset of a finite monoid M and $\varphi : \Sigma^* \to M$ is a morphism (cf. Theorem 1.3). If we represent an element $m \in M$ by a M-indexed bit vector v_m such that $v_m[i] = 1 \iff i = m$, then a translation $m \mapsto mp$ can be represented by a *purely disjunctive* formula:

$$v_{mp}[i] = v_m[j_1] \vee \ldots \vee v_m[j_k] \text{ where } \{j_1, \ldots, j_k\} = \{j \in M \mid jp = i\}$$

Moreover, this is *linear* in the following sense: given a fixed $p \in M$, each index $j \in M$ is involved in the right-hand side of this formula for exactly one $i \in M$.

Let $\mathtt{ttt} = \lambda^\circ x.\,\mathtt{true} : \mathtt{Bool} \multimap \mathtt{Bool}$ and $\mathtt{fff} = \lambda^\circ x.\,x : \mathtt{Bool} \multimap \mathtt{Bool}$. This makes the type $B = \mathtt{Bool} \multimap \mathtt{Bool}$ into a kind of type of booleans that supports a disjunction of type $B \multimap B \multimap B$ (by function composition) and a type-cast function of type $B \multimap \mathtt{Bool}$ (by applying to `false`). (Of course B has other closed inhabitants besides `ttt` and `fff`, but we only use those two.) Using this type and the "iteration+continuations" recipe of the proof of Lemma 4.10, one can define a λa-term of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ that decides the language L with $A = B \multimap \ldots \multimap B \multimap \mathtt{Bool}$ (with $|M|$ arguments of type B). ◀

Let us go further. According to Theorem 4.1, the $\lambda\wp$-calculus can define all aperiodic sequential functions; we show that as one can expect, the aperiodicity condition is lifted when moving to the commutative λa-calculus. However, the trick used in the direct encoding of the above proof does not work, and we have only managed to encode general sequential functions by resorting to the Krohn–Rhodes theorem.

▶ **Theorem 5.2** (Krohn–Rhodes decomposition, non-aperiodic case, cf. Appendix A in [37])**.** *Any* sequential function $f : \Sigma^* \to \Pi^*$ *can be realized as a composition $f = f_1 \circ \ldots \circ f_n$ (with $f_i : \Xi_i^* \to \Xi_{i-1}^*$, $\Xi_0 = \Pi$ and $\Xi_n = \Sigma$) where each function f_i is computed by some sequential transducer whose transition monoid is either* aperiodic *or a* group*.*

▶ Remark 5.3. By Theorem 4.8, the aperiodic transducers among the f_i can be further decomposed into two-state aperiodic transducers.

▶ **Theorem 5.4.** *Any sequential function $\Sigma^* \to \Pi^*$ can be expressed by some λa-term of type* $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$*, for a* purely affine *type A depending on the function.*

Proof sketch. First, by Theorem 4.1, we can already encode aperiodic sequential functions, since every well-typed $\lambda\wp$-term is also a well-typed λa-term. One can also show that Lemma 2.8 applies to the λa-calculus. By the general Krohn–Rhodes theorem, we just need to handle the case of a sequential transducer whose transition monoid is a group.

The idea, in terms of set-theoretic maps as in our explanation of the proof of Lemma 4.10 (whose notations we borrow here), is as follows. The current state $q \in Q$ and output history $w \in \Pi^*$ is represented by a Q-indexed family $(g_{q'})_{q' \in Q}$ of functions such that $g_q = \kappa_w$ and for

$q' \neq q$, $g_{q'} = \zeta$. The transition $\delta(-,c)$ is represented by $(g_q)_{q\in Q} \mapsto (g_{\sigma(q)} \circ \kappa_{\delta_{\mathrm{out}}(\sigma(q),c)})_{q\in Q}$ where $\sigma = (\delta_{\mathrm{st}}(-,c))^{-1}$ – the latter is well-defined because the group assumption means that $\delta_{\mathrm{st}}(-,c)$ is a permutation of Q. The final output is obtained at the end as the concatenation $g_{q_1}(F(q_1)) \dots g_{q_n}(F(q_n))$ where $Q = \{q_1, \dots, q_n\}$ (with an arbitrary enumeration of Q).

The elaboration of the corresponding λa-term is left to the reader. Keep in mind that the reason this term will not be well-typed for the $\lambda\wp$-calculus is that the inversions in the permutation $\delta_{\mathrm{st}}(-,c)$ correspond to violations of non-commutative typing. ◀

5.2 Extension with additive pairs

Let's look at what happens if we add the *additive conjunction* connective of linear logic to the $\lambda\wp$-calculus. The $\lambda\wp^{\&}$*-calculus* is obtained by adding $A, B ::= \dots \mid A \mathbin{\&} B$ to the grammar of types and $t, u ::= \dots \mid \langle t, u\rangle \mid \pi_1\, t \mid \pi_2\, t$ for terms, with the typing rules

$$\frac{\Gamma \mid \Delta \vdash t : A \quad \Gamma \mid \Delta \vdash u : B}{\Gamma \mid \Delta \vdash \langle t, u\rangle : A \mathbin{\&} B} \qquad \frac{\Gamma \mid \Delta \vdash t : A_1 \mathbin{\&} A_2}{\Gamma \mid \Delta \vdash \pi_i\, t : A_i} \qquad \text{(see [39, §4])}$$

the β-reduction rules $\pi_i\, \langle t_1, t_2\rangle \longrightarrow_\beta t_i$, and the corresponding η-conversion rules.

Recall that we discussed both in the introduction and in Remark 2.7 the need to prevent the existence of a $\lambda\wp$-term of type $\texttt{Bool} \multimap \texttt{Bool}$ for negation. However, if we use the additive conjunction to define the type $\texttt{Bool}^{\&} = (o \mathbin{\&} o) \multimap o$, the following are well-typed $\lambda\wp^{\&}$-terms:

$$\texttt{true}^{\&} = \lambda^{\circ} p.\ \pi_1\, p \qquad \texttt{false}^{\&} = \lambda^{\circ} p.\ \pi_2\, p \qquad \texttt{not}^{\&} = \lambda^{\circ} b.\ \lambda^{\circ} p.\ b\ \langle \pi_2\, p, \pi_1\, p\rangle$$

More generally:

▶ **Proposition 5.5.** *Let* $\texttt{Fin}^{\&}(n) = (o \mathbin{\&} \dots \mathbin{\&} o) \multimap o$. *For all* $n \in \mathbb{N}$, *there is a canonical bijection between* $\{1, \dots, n\}$ *and the closed* $\lambda\wp^{\&}$*-terms of type* $\texttt{Fin}^{\&}(n)$. *Furthermore, using this encoding, every map* $\{1, \dots, n_1\} \times \dots \times \{1, \dots, n_k\} \to \{1, \dots, m\}$ *can be defined by a closed* $\lambda\wp^{\&}$*-term of type* $\texttt{Fin}^{\&}(n_1) \multimap \dots \texttt{Fin}^{\&}(n_k) \multimap \texttt{Fin}^{\&}(m)$.

▶ **Corollary 5.6.** *Every* regular *language can be defined by a closed* $\lambda\wp^{\&}$*-term of type* $\texttt{Str}_\Sigma[A] \multimap \texttt{Bool}$ *for some purely affine type* A *– we consider "&" as an affine connective and therefore allow it in* A.

Proof idea. Take $A = \texttt{Fin}^{\&}(|M|)$ where M is any finite monoid that recognizes the language as specified in Theorem 1.3. (We could also prove the converse by relying on an extension of Hillebrand and Kanellakis's Theorem 1.1 to the simply typed λ-calculus with products.) ◀

Similarly, one could show that the addition of the *additive disjunction* "$\oplus$" of linear logic to the $\lambda\wp$-calculus would be sufficient to encode all regular languages.

5.3 On regular and first-order tree languages: a discussion

There is a rich theory of *tree automata* that extends the notion of regular language to trees over ranked alphabets instead of strings. Such trees admit Church encodings; for instance, for an alphabet with arities $(a : 2, b : 2, x : 0)$ (i.e. for trees with two kind of binary nodes and one kind of leaf) one would have $\texttt{Tree}_{(2,2,0)} = (o \multimap o \multimap o) \to (o \multimap o \multimap o) \to o \to o$.

▶ Remark 5.7. A string over an alphabet $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$ can be seen as a tree with arities $(c_1 : 1,\ \dots,\ c_{|\Sigma|} : 1, \varepsilon : 0)$. This would lead to defining the type of Church-encoded strings as $\texttt{Str}'_\Sigma = (o \multimap o) \to \dots \to (o \multimap o) \to o \to o$. Our type $\texttt{Str}_\Sigma$, which is the traditional choice in linear logic (see the discussion on Church numerals in [17, §5.3.2]), is a bit more precise since it expresses that such a “unary tree” can only contain one ε node. But as there exist conversion functions $\mathtt{Str}_\Sigma \multimap \mathtt{Str}'_\Sigma$ and $\mathtt{Str}'_\Sigma[o \multimap o] \multimap \mathtt{Str}_\Sigma$, this choice does not make much difference (thanks again to Lemma 2.8).

We shall not go into the details of tree automata here, but the knowledgeable reader may check that Proposition 5.5 can be used to encode all *regular tree languages* over $(a : 2, b : 2, x : 0)$ as closed $\lambda\wp^{\&}$-terms of type $\mathtt{Tree}_{(2,2,0)}[A] \multimap \mathtt{Bool}$ for purely affine A. Predictably, this fails for the $\lambda\wp$-calculus without additive connectives. More noteworthy is the failure of the trick used to prove Theorem 5.1 for the commutative λa-calculus when one replaces strings with trees. Thus, it seems (though this remains conjectural) that *the additives of linear logic might be required to express some regular tree languages.*

We believe that this is no accident and that some fundamental difficulty of automata theory is being manifested here. Indeed, if we had a characterization of regular tree languages in the λa-calculus, we could expect that moving to the $\lambda\wp$-calculus would yield the *first-order tree languages*, which are the commonly accepted counterpart of star-free languages for trees. (Recall from Theorem 1.5 that definability in first-order logic is among the equivalent definitions of star-free languages.) However, while Theorem 1.5 demonstrates that star-free languages are well-understood, the situation is quite different for first-order tree languages: there is no known algebraic characterization, and neither is there any known algorithm to decide whether a tree automaton recognizes a first-order language (see e.g. [9, §3]). Another argument for the necessity of additives, discussed in the next section, comes from transducers.

6 Next episode preview: transducers in typed λ-calculi

We started from Hillebrand and Kanellakis’s Theorem 1.1 and obtained an analogous statement for star-free languages instead of regular languages. Another direction that we could have pursued is to replace languages by *functions*, by looking at the type $\mathtt{Str}_\Sigma[A] \to \mathtt{Str}_\Pi$. Indeed, an immediate consequence of this “regular = λ-definable” result is:

▶ **Corollary 6.1.** *If $f : \Sigma^* \to \Pi^*$ is definable by a closed simply typed λ-term of type $\mathtt{Str}_\Sigma[A] \to \mathtt{Str}_\Pi$, then for any regular language $L \subseteq \Pi^*$, $f^{-1}(L) \subseteq \Sigma^*$ is also regular.*

Proof idea. Let $u : \mathtt{Str}_\Pi[B] \to \mathtt{Bool}$ and $t : \mathtt{Str}_\Sigma[A] \to \mathtt{Str}_\Pi$ be simply typed λ-terms defining L and f respectively. Then $f^{-1}(L)$ is defined by $\lambda x.\, u\,(t\,x)$ which is well-typed with type $\mathtt{Str}_\Sigma[A[B]] \to \mathtt{Bool}$ (analogously to Lemma 2.8). ◀

This suggests a connection between these λ-definable string functions and automata theory. But while it is not too hard to define functions of hyperexponential growth in the simply typed λ-calculus, most classes of string functions from automata theory (see [36] for a recent survey) grow much more slowly (polynomially or even linearly in the input size). The challenge then becomes to *restrict the expressiveness via types* to capture such classes. This calls for the recipes that have worked here, namely affine types and non-commutativity.

▷ Claim 6.2 (to be proved in a sequel). The functions definable by closed terms of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$, for purely affine A, are the *MSO transductions*[14] [14] (a.k.a. *regular functions*[15]) in the *λa-calculus* and the *FO transductions* in the *$\lambda\wp$-calculus.*

[14] MSO stands for Monadic Second-Order Logic while FO stands for First-Order Logic, cf. the introduction.
[15] This name is somewhat confusing, since there are multiple classes of string functions that collapse to the single class of regular languages when we consider indicator functions. For example, in-between the sequential functions (Definition 4.4) and the regular (MSO-definable) functions, there is a widely

This goes beyond the encodings of sequential transducers presented in this paper (Theorem 4.1 and Theorem 5.4). But the latter are an important stepping stone, since we do not know how to prove the above claim without using the Krohn–Rhodes decomposition somewhere. To summarize the results of the present paper together with its planned sequel:

calculus	affine	commutative	$\mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$	$\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$
$\lambda\wp$	yes	no	star-free (FO-definable) languages	FO transductions
λa	yes	yes	regular (MSO-definable) languages	MSO transductions

While the connection between non-commutativity and aperiodicity came as a surprise to us, we had more reasons to suspect that affine types should have something to do with transducers. Indeed, the term "linear" itself has been used to describe the *copyless assignment* condition on streaming string transducers (SSTs) [5], a machine model for MSO transductions, e.g. "updates should make a linear use of registers" [15, §5] (in our terminology, the register assignments of SSTs are in fact affine, not strictly linear). Moreover, it seems (informally speaking) that the more sophisticated *single-use-restricted assignments* of streaming *tree* transducers [3] correspond to a form of linearity that incorporates an *additive conjunction*, whereas copyless assignments are purely *multiplicative*; compare with the discussion of §5.3.

7 Related work

We have already mentioned in the introduction several lines of tangentially related research, such as higher-order model checking or the topology of non-commutative proofs. In this section, we discuss a few references that we deemed to be more directly relevant.

Automata as circular proofs. Aside from Hillebrand and Kanellakis's Theorem 1.1, perhaps our most direct precursors in "implicit automata theory" are the works by DeYoung and Pfenning [13] on sequential transducers (their version seems to be equivalent to Definition 4.4) and by Kuperberg, Pinault and Pous [28] characterizing regular languages and deterministic logarithmic space complexity. Both rely on a proofs-as-programs interpretation of *circular*[16] *proof systems* for some variants of linear logic with fixed points.

The Church encoding of strings is obtained by a systematic procedure [12] from the inductive definition $s ::= \varepsilon \mid c_1 \cdot s \mid \ldots \mid c_{|\Sigma|} \cdot s$ ($\Sigma = \{c_1, \ldots, c_{|\Sigma|}\}$). Using fixed points of formulas, one can instead turn it into the recursive type[17] $\mathtt{Str}^\mu_\Sigma = 1 \oplus \mathtt{Str}^\mu_\Sigma \oplus \ldots \oplus \mathtt{Str}^\mu_\Sigma$; this is the definition of the type of strings in [13], and it is also implicitly at work in[18] [28].

So both our approach (following Hillebrand and Kanellakis [24]) and those using fixed point logics morally work because the consumption of strings represented as inductive data types is similar to their traversal by automata. However, while the use of the "right fold" provided by a Church-encoded string involves an "inversion of control" (in programming jargon) that, in the case of the simply typed λ-calculus, has drastic effects on expressive power[19] (contrast Theorem 1.1 with the fact that $\beta\eta$-convertibility of simply typed λ-terms is not elementary recursive [32]), circular proofs seem to give the programmer more degrees of freedom: Kuperberg et al. do not need to add polymorphism to go beyond regular languages.

studied strictly intermediate class called the *rational functions*. (The adjective "rational" is used to refer to regular languages in a French tradition going back to Nivat and Schützenberger.)

[16] These are sometimes called "cyclic" proofs, but in our context, this would create a confusion with an unrelated non-commutative logic, *cyclic linear logic* [50].

[17] Formally, this is expressed as the least fixed point $\mathtt{Str}^\mu_\Sigma = \mu\alpha.\ 1 \oplus \alpha \oplus \ldots \oplus \alpha$.

[18] The left rules given in [28, Figure 1] for A and A^* correspond to $A = 1 \oplus \ldots \oplus 1$ and $A^* = 1 \oplus (A \otimes A^*)$.

[19] To overcome those limits and express any elementary recursive function as a simply typed λ-term, Hillebrand and Kanellakis use an alternative representation of inputs inspired by database theory [24].

Recognizable languages of λ-terms. A modern point of view on Hillebrand and Kanellakis's Theorem 1.1 can be implicitly found in a paper by Terui [48] emphasizing the method of *evaluation in a finite denotational semantics* used to prove it. Along these lines, general notions of recognizable languages of closed λ-terms of a given type (specializing to regular languages for the type of Church-encoded strings) have been proposed, based on finite semantics, in the simply-typed λ-calculus by Salvati [45] and in an infinitary λ-calculus by Melliès [34]. It is plausible that Theorem 1.1 can be extended to give an equivalent syntactic definition for Salvati's recognizable languages: for a simple type B they would be the languages definable by $B[A] \to \texttt{Bool}$. An interesting question would be whether one can give an encoding of *higher-dimensional trees* in the simply typed λ-calculus so that this notion of recognizability coincides with Rogers's automata for those trees [43, 16].

Other implicit automata results. In a recent preprint, Bojańczyk [10] introduces a new class of string-to-string functions that admits several equivalent definitions (see also [11]). One of them uses the simply typed λ-calculus enriched with a ground type of lists and several primitive functions on lists. Strings are represented as lists of characters, which differs from our use of functional encodings in a λ-calculus without any primitive data type.

Using a computational model inspired by denotational semantics of linear logic, Seiller [46] gives a characterization of each level of the k-head two-way non-deterministic automata hierarchy. The lowest level ($k = 1$) corresponds to regular languages, while the union over $k \in \mathbb{N}_{\geq 1}$ gives the complexity class NL (non-deterministic logarithmic space). Something in common with our work is that the representation of strings used by [46] is more or less a semantic version of Church encodings (see [46, §3.2]). There is one main difference with what one usually calls implicit complexity: Seiller's result does not take place inside a syntactically defined programming language (and it is far from obvious how to turn this model into a similarly expressive syntax, because of the previously mentioned inversion of control).

Controlling expressible functions with non-commutativity. The tree-processing programming language of Kodama, Suenaga and Kobayashi [26] uses non-commutative types to force programs to process their input in a depth-first, left-to-right fashion. This allows them to be compiled into a target language that works on a stream of tokens, suggesting a possible connection with nested word automata [4]. The non-commutativity is restricted to arguments of ground type in [26], whereas it is important for our $\lambda\wp$-calculus that it applies at all orders (indeed, since we encode data as functions, higher-order functions are pervasive).

References

1 Samson Abramsky. Temperley–Lieb Algebra: From Knot Theory to Logic and Computation via Quantum Mechanics. In Goong Chen, Louis Kauffman, and Samuel Lomonaco, editors, *Mathematics of Quantum Computation and Quantum Technology*, volume 20074453, pages 515–558. Chapman and Hall/CRC, September 2007. doi:10.1201/9781584889007.ch15.

2 Klaus Aehlig. A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of Automata. *Logical Methods in Computer Science*, 3(3), July 2007. doi:10.2168/LMCS-3(3:1)2007.

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

4 Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):16:1–16:43, 2009. doi:10.1145/1516512.1516518.

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

6 Jean-Marc Andreoli, Gabriele Pulcini, and Paul Ruet. Permutative logic. In C.-H. Luke Ong, editor, *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3634 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2005. doi:10.1007/11538363_14.

7 Andrew Barber. Dual Intuitionistic Linear Logic. Technical report ECS-LFCS-96-347, LFCS, University of Edinburgh, 1996. URL: http://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/.

8 Mikołaj Bojańczyk. The simplest transducer models and their Krohn-Rhodes decompositions. https://www.mimuw.edu.pl/~bojan/slides/transducer-course/krohn-rhodes.html. Slides of a lecture given at FSTTCS '19, accessed on 11-02-2020.

9 Mikołaj Bojańczyk. Automata column: Some Open Problems in Automata and Logic. *ACM SIGLOG News*, 1(2):3–12, October 2014. doi:10.1145/2677161.2677163.

10 Mikołaj Bojańczyk. Polyregular Functions. *CoRR*, abs/1810.08760, October 2018. arXiv:1810.08760.

11 Mikołaj Bojańczyk, Sandra Kiefer, and Nathan Lhote. String-to-String Interpretations With Polynomial-Size Output. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 106:1–106:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.106.

12 Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, January 1985. doi:10.1016/0304-3975(85)90135-5.

13 Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 3–22, 2016. doi:10.1007/978-3-319-47958-3_1.

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

15 Emmanuel Filiot and Pierre-Alain Reynier. Transducers, Logic and Algebra for Functions of Finite Words. *ACM SIGLOG News*, 3(3):4–19, August 2016. doi:10.1145/2984450.2984453.

16 Neil Ghani and Alexander Kurz. Higher dimensional trees, algebraically. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007, Bergen, Norway, August 20-24, 2007, Proceedings*, volume 4624 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2007. doi:10.1007/978-3-540-73859-6_16.

17 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987. doi:10.1016/0304-3975(87)90045-4.

18 Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, Providence, RI, 1989. Proceedings of a Summer Research Conference held June 14–20, 1987. doi:10.1090/conm/092/1003197.

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

20 Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, April 1992. doi:10.1016/0304-3975(92)90386-T.

21 Charles Grellois. *Semantics of linear logic and higher-order model-checking*. PhD thesis, Université Paris 7, April 2016. URL: `https://tel.archives-ouvertes.fr/tel-01311150/`.

22 Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1), January 2007. `doi:10.1145/1182613.1182614`.

23 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible Pushdown Automata and Recursion Schemes. *ACM Transactions on Computational Logic*, 18(3):25:1–25:42, August 2017. `doi:10.1145/3091122`.

24 Gerd G. Hillebrand and Paris C. Kanellakis. 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*, pages 253–263. IEEE Computer Society, 1996. `doi:10.1109/LICS.1996.561337`.

25 Naoki Kobayashi. Model Checking Higher-Order Programs. *Journal of the ACM*, 60(3):1–62, June 2013. `doi:10.1145/2487241.2487246`.

26 Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. *Journal of Functional Programming*, 18(3):333–371, 2008. `doi:10.1017/S0956796807006570`.

27 Kenneth Krohn and John Rhodes. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965. `doi:10.1090/S0002-9947-1965-0188316-1`.

28 Denis Kuperberg, Laureline Pinault, and Damien Pous. Cyclic Proofs and Jumping Automata. In Arkadev Chattopadhyay and Paul Gastin, editors, *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019)*, volume 150 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. `doi:10.4230/LIPIcs.FSTTCS.2019.45`.

29 Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

30 Olivier Laurent. Polynomial time in untyped elementary linear logic. *Theoretical Computer Science*, 813:117–142, April 2020. `doi:10.1016/j.tcs.2019.10.002`.

31 Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*, pages 460–469, Tucson, AZ, USA, November 1983. `doi:10.1109/SFCS.1983.50`.

32 Harry G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, September 1992. `doi:10.1016/0304-3975(92)90020-G`.

33 Damiano Mazza. *Polyadic Approximations in Logic and Computation*. Habilitation à diriger des recherches, Université Paris 13, November 2017. URL: `https://lipn.fr/~mazza/papers/Habilitation.pdf`.

34 Paul-André Melliès. Higher-order parity automata. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, Reykjavik, Iceland, June 2017. IEEE. `doi:10.1109/LICS.2017.8005077`.

35 Paul-André Melliès. Ribbon Tensorial Logic. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*, pages 689–698, Oxford, United Kingdom, 2018. ACM Press. `doi:10.1145/3209108.3209129`.

36 Anca Muscholl and Gabriele Puppis. The Many Facets of String Transducers. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. `doi:10.4230/LIPIcs.STACS.2019.2`.

37 Lê Thành Dũng Nguyễn and Pierre Pradic. Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic, 2020. Full version of this article. URL: `https://hal.archives-ouvertes.fr/hal-02476219`.

38 C.-H. Luke Ong. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90, Seattle, WA, USA, 2006. IEEE. doi:10.1109/LICS.2006.38.

39 Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-communicative linear logic. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 1999. doi:10.1007/3-540-48959-2_21.

40 Jeff Polakow and Frank Pfenning. Relating Natural Deduction and Sequent Calculus for Intuitionistic Non-Commutative Linear Logic. *Electronic Notes in Theoretical Computer Science*, 20:449–466, January 1999. doi:10.1016/S1571-0661(04)80088-4.

41 Christian Retoré. Pomset logic: A non-commutative extension of classical linear logic. In Philippe de Groote, editor, *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, volume 1210 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 1997. doi:10.1007/3-540-62688-3_43.

42 John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3):233–247, November 1993. doi:10.1007/BF01019459.

43 James Rogers. Syntactic Structures as Multi-dimensional Trees. *Research on Language and Computation*, 1(3):265–305, September 2003. doi:10.1023/A:1024695608419.

44 Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, 2009. Translated by Reuben Thomas. doi:10.1017/CBO9781139195218.

45 Sylvain Salvati. Recognizability in the simply typed lambda-calculus. In Hiroakira Ono, Makoto Kanazawa, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 16th International Workshop, WoLLIC 2009, Tokyo, Japan, June 21-24, 2009. Proceedings*, volume 5514 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2009. doi:10.1007/978-3-642-02261-6_5.

46 Thomas Seiller. Interaction Graphs: Non-Deterministic Automata. *ACM Transactions on Computational Logic*, 19(3):21:1–21:24, August 2018. doi:10.1145/3226594.

47 Howard Straubing. First-order logic and aperiodic languages: a revisionist history. *ACM SIGLOG News*, 5(3):4–20, 2018. doi:10.1145/3242953.3242956.

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

49 Igor Walukiewicz. LambdaY-calculus with priorities. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785674.

50 David N. Yetter. Quantales and (noncommutative) linear logic. *The Journal of Symbolic Logic*, 55(1):41–64, March 1990. doi:10.2307/2274953.

51 Noam Zeilberger and Alain Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Logical Methods in Computer Science*, 11(3), September 2015. doi:10.2168/LMCS-11(3:22)2015.