Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic

Lê Thành Dũng Nguyễn, Pierre Pradic

Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic

Lê Thành Dũng Nguyễn
Université publique[1], France
https://nguyentito.eu/
nltd@nguyentito.eu

Pierre Pradic
Department of Computer Science, University of Oxford, United Kingdom
https://www.cs.ox.ac.uk/people/pierre.pradic/
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 to control the expressible functions in a programming language.

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

Keywords and phrases Church encodings, non-commutative linear logic, star-free languages

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.

The remainder of the title page has been left blank to make the 12-page limit "excluding references and the front page(s) (authors, affiliation, keywords, abstract, ...)" easier to check. The main text of the paper thus occupies the pages 2 to 13.

[1] See the manifesto https://pageperso.lif.univ-mrs.fr/~sylvain.sene/affiliation.html (French), archived on 2020–02–12 on the Internet Wayback Machine (https://archive.org/).

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. [14, 32]). 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 [40, 25] concerning multi-head automata, and, most importantly, a remarkable result from 1996 that provides our starting point:

▶ **Theorem 1.1** (Hillebrand & Kanellakis [22, 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:

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

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[2], Hillebrand and Kanellakis's theorem should not be surprising in retrospect: there are strong connections between Church encodings and automata (see e.g. [39, 42, 30]), that have been exploited in particular in *higher-order model checking* for the past 15 years [2, 34, 23, 19, 21, 43]. This is not a mere contrivance: these encodings have been a canonical data representation for λ-calculi for much longer[3].

[2] See e.g. Damiano Mazza's answer to this MathOverflow question: `https://mathoverflow.net/q/296879`

[3] They were introduced for booleans and integers by Church in the 1930s, and later generalized by Böhm and Berarducci [10], see also `http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html`. (Similar ideas appear around the same time in [28].) As for the refined encodings with linear types that we use later, they already appear in Girard's founding paper on linear logic [15, §5.3.3].

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** (classical). *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 [41] for a historical discussion) has led to many equivalent definitions for star-free languages, with the algebraic notion of *aperiodicity* playing a key role:

▶ **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. [41]). *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. Observe that `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[4] logic [15], a system whose "resource-sensitive" nature has been previously exploited in ICC [18, 17]. Not coincidentally, the theme of non-commutativity first appeared in a form of linear logic *ante litteram*, namely the Lambek calculus [26], and resurfaced shortly after the official birth of linear logic: it is already mentioned by Girard in a 1987 colloquium [16].

We shall therefore introduce and use a variant of Polakow and Pfenning's Intuitionistic Non-Commutative Linear Logic [35, 36], 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.*

[4] 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.

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

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

where $\texttt{Str}_\Sigma$ has $|\Sigma|$ arguments of type $(o \multimap o)$. Setting $\texttt{true} = \lambda^\circ x.\, \lambda^\circ y.\, x : \texttt{Bool}$ and $\texttt{false} = \lambda^\circ x.\, \lambda^\circ y.\, y : \texttt{Bool}$ 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 $\texttt{Str}_\Sigma[A] \multimap \texttt{Bool}$ for some* purely affine *type A (that may depend on L).*

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, this result on the computational power of a non-commutative λ-calculus is the first of its kind, despite the age of the subject. Previous works on non-commutative types indeed tend to see λ-terms (or proof nets) as static objects, and to focus on their topological aspects (e.g. [5, 44, 31]), though there is another tradition relating self-dual non-commutativity to process algebras [37, 20].

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 affine λ-calculus* (cf. [1, 44]):

▶ **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" [29, 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* [24].

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), and finally our plans for the next papers in the series (§6).

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)

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

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

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

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

Figure 1 The typing rules of the $\lambda\wp$-calculus.

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 [6]): 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 [35, 36], the $\lambda\wp$-calculus:

- contains two function types instead of four[5];
- is affine instead of linear (all functions can discard arguments) – this seems important to get enough expressive power for our purposes[6].

▶ 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

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

▶ **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:

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

[5] Our '$\to$' and '$\multimap$' are called "intuitionistic functions" and "right ordered functions" in [35]; we have no counterpart for the "linear [commutative] functions" and "left ordered functions" in the $\lambda\wp$-calculus.

[6] 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. [27]. But non-commutativity obstructs the free movement of garbage.

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

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

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

If, for some type A *and some context* $\Gamma \mid \Delta$ *independent of* c*, we have* $\Gamma \mid \Delta \vdash t_c : A \multimap A$ *for all* $c \in \Sigma$*, then* $\Gamma \mid \Delta \vdash w^\dagger(\vec{t_\Sigma}) : A \multimap A$.

Furthermore, for any choice of variables $(f_c)_{c\in\Sigma}$*, there is 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, given by* $w \in \Sigma^* \mapsto w^\dagger(\vec{x}_\Sigma)$.

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, we have* $\overline{w}\, t_{c_1} \dots t_{c_{|\Sigma|}} \longrightarrow_\beta^* w^\dagger(\vec{t_\Sigma})$.

▶ Remark 2.6. For $\Sigma = \{a\}$, η-conversion is necessary to identify $\lambda^{\rightarrow} f.\, f : \mathtt{Str}_\Sigma$ with $\overline{a}$.

▶ Remark 2.7. Our presentation of Church encodings stresses the role of *open terms*. Indeed, it is important to note that when the context Γ of non-affine variables contains $f_c : o \multimap o$ for each $c \in \Sigma$, then any string can be represented as a term of type $o \multimap o$ in that context, and such strings can even be concatenated by function composition. 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

We start by demonstrating how non-commutativity entails aperiodicity.

Proof of Theorem 1.8. The monoid structure is obvious (the identity element is $\lambda^\circ x.\, x$), and it is well-known that in an affine type system, any purely affine type has finitely many $\beta\eta$-equivalence classes of closed inhabitants (here non-commutativity plays no role). Therefore, the remainder of the proof concerns aperiodicity.

Let $t : A \multimap A$; our goal is to show that the sequence $t^n = t \circ \dots \circ t =_\beta \lambda^\circ x.\, t\,(\dots\,(t\,x)\dots)$ 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 $B_1 \multimap \dots \multimap B_m \multimap o$. The base case $m = 0$ being trivial, we assume $m \geq 1$. In this case, t has an η-long β-normal form

$$t = \lambda^\circ x.\, \lambda^\circ y_1. \dots \lambda^\circ y_m.\, z\, u_1 \dots u_k$$

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

From now on we can assume that $z = x$, which entails $k = m$ since the variable x is of type $A = B_1 \multimap \dots \multimap B_m \multimap o$ and we must have $x : A, y_1 : B_1, \dots, y_m : B_m \vdash x\, u_1 \dots u_k : o$. By a straightforward induction, for $n \in \mathbb{N}$, t^n is $\beta\eta$-convertible to

$$\lambda^\circ x.\, \lambda^\circ y_1. \dots \lambda^\circ y_k.\, x\, u_1^{(n)} \dots u_k^{(n)} \quad \text{where} \quad \vec{u}^{(0)} = (y_1, \dots, y_k),\ \vec{u}^{(n+1)} = \vec{u}^{(n)}[\vec{y} := \vec{u}]$$

Here $\vec{u} = (u_1, \dots, u_k)$ and $\vec{u}^{(n)}$ are k-tuples of $\lambda\wp$-terms. The notation $[\vec{y} := \vec{u}]$ denotes a parallel substitution, applied componentwise to $\vec{u}^{(n)}$; in other words

$$\vec{u}_i^{(n+1)} = \vec{u}_i^{(n)}[y_1 := u_1, \dots, y_k := u_k] \quad \text{for } i \in \{1, \dots, k\}$$

Thus, the question becomes: is the sequence of $(\vec{u}^{(n)})_{n\in\mathbb{N}}$ eventually constant modulo $\beta\eta$?

To reach a positive answer, 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 $\forall i,\ y_i \in \mathrm{FV}(u_j) \iff (y_i : B_i) \in \Delta_j$;
- $\Delta_1 \cdot \ldots \cdot \Delta_k$ is an *ordered subsequence* of $(y_1 : B_1) \cdot \ldots \cdot (y_k : B_k)$.

Therefore, for any $i \in \{1,\dots,k\}$, the sequence $((\mu_{\vec{u}})^n(i))_{n\in\mathbb{N}}$ is either non-increasing or non-decreasing as long as it is defined; so at some iteration $n = N_i$, either it becomes undefined or it reaches a fixed point of $\mu_{\vec{u}}$. By taking $N = \max_{1\le i\le k} N_i$, we have $(\mu_{\vec{u}})^N = (\mu_{\vec{u}})^{N+1}$.

Next, let $i \in \{1,\dots,k\}$. Recall that our goal is to prove that $(u_i^{(n)})_{n\in\mathbb{N}}$ is eventually constant modulo $\beta\eta$. The simple case is when $i \notin (\mu_{\vec{u}})^N(\{1,\dots,k\})$: $u_i^{(N)}$ has no free variables, so $u_i^{(N+1)} = u_i^{(N)}[\vec{y} := \vec{u}] = u_i^{(N)}$. 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[7]. 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, $u_i^{(N+n+1)} =_\beta (\lambda^\circ y_i.\, r_i)\, u_i^{(N+n)}$. So we just have to make sure that the sequence of iterates of the $\lambda\wp$-term $\lambda^\circ y_i.\, r_i : B_i \multimap B_i$ is eventually constant modulo $\beta\eta$. Since the type B_i has size strictly smaller than A, the induction hypothesis applies. (It is clear that $\lambda^\circ y_i.\, r_i$ 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).) ◀

After this, we must analyze the form of a closed $\lambda\wp$-term of type $\mathtt{Str}_\Gamma[A] \multimap \mathtt{Bool}$.

▶ **Lemma 3.1.** *Let $\Sigma = \{c_1,\dots,c_{|\Sigma|}\}$ be a finite alphabet, A be a* purely affine *type and $t : \mathtt{Str}_\Gamma[A] \multimap \mathtt{Bool}$ 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 $t =_{\beta\eta} \lambda^\circ s.\, h\,(s\, g_{c_1}\ \dots\ g_{c_{|\Sigma|}})$.*

Proof. By inspection of the normal form of t, see Appendix B.1. ◀

Reusing the notations of this lemma, let us define

$$\varphi : w = w[1]\dots w[n] \in \Sigma^* \mapsto \lambda^\circ x.\, g_{w[1]}\,(\dots\,(g_{w[n]}\,x)) \in \{v \mid v : A \multimap A\}/=_{\beta\eta}$$

φ is a monoid morphism and $\varphi^{-1}(\{v \mid h\, v =_{\beta\eta} \mathtt{true}\})$ is none other than the language defined by the $t : \mathtt{Str}_\Sigma[A] \multimap \mathtt{Bool}$ in the lemma. Since the codomain of φ is finite and aperiodic by Theorem 1.8, L is a star-free language. This proves the soundness part of Theorem 1.7.

[7] 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 $\vec{y} \cap (\mathrm{FV}(\vec{t_1}) \setminus \vec{x}) = \varnothing$.

4 Expressiveness of the $\lambda_{\wp}$-calculus

We now turn to the *extensional completeness* part in Theorem 1.7. To prove it, 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 define the notion of *aperiodic sequential function*, 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*.*

A few straightforward lemmas, whose proof may be found in Appendix B, allow us to deduce immediately the desired extensional completeness result:

▶ **Lemma 4.2.** *If a language* $L \subseteq \Sigma^*$ *is* star-free*, then its* 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 of type* $\mathtt{Str}_{\{1\}}[\mathtt{Bool}] \multimap \mathtt{Bool}$ *that tests whether its input string is non-empty.*

▶ **Lemma 4.4.** *If* $\vdash t : A[T] \multimap B$ *and* $\vdash u : B[U] \multimap C$*, then* $\vdash \lambda^\circ x.\, u\,(t\,x) : A[T[U]] \multimap C$*.*

This composition lemma is also involved in the proof of Theorem 4.1. Our proof strategy will indeed consist in encoding small "building blocks" for aperiodic sequential functions and composing them together, thanks to the *Krohn–Rhodes decomposition* (Theorem 4.8).

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 [38, Chapter V].

▶ **Definition 4.5.** *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* $q_0 = q_I$*,* $q_1 = \delta_1(q_0, w[1])$*, …,* $q_n = \delta_1(q_{n-1}, w[n])$*. Its* output *is obtained as the concatenation* $\delta_2(q_0, w[1]) \cdot \ldots \cdot \delta_2(q_{n-1}, w[n]) \cdot F(q_n)$*.*

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

▶ **Definition 4.6.** *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* $\{\delta_1(-, c) \mid c \in \Sigma\}$ *(where* $\delta_1(-, c)$ *stands for* $q \mapsto \delta_1(q, c)$*).*

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*.*

▶ 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)**.** *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*.*

▶ 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. [9, §1.1]); see [8] for a tutorial containing a proof sketch of this version. In Appendix A, 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 4.4), 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.*

Proof. Let us start by exposing the rough idea of the trick using set-theoretic maps. We reuse the notations of Definition 4.5 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 by the pair (κ_w, ζ) (resp. (ζ, κ_w)) where $\zeta, \kappa_w : \Pi^* \to \Pi^*$ are defined by $\zeta : x \mapsto \varepsilon$ and $\kappa_w : x \mapsto wx$. 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 $\delta_1(-, c)$; we have thanks to the above identities:

- $(h, g) \mapsto (h \circ \kappa_{\delta_2(1,c)}, g \circ \kappa_{\delta_2(2,c)})$ when $\delta_1(-, c) = \mathrm{id}$;
- $(h, g) \mapsto (\kappa_{h(\delta_2(1,c))g(\delta_2(2,c))}, \zeta)$ when $\delta_1(-, c) : q' \mapsto 1$ (note that $h = \zeta$ xor $g = \zeta$);
- $(h, g) \mapsto (\zeta, \kappa_{h(\delta_2(1,c))g(\delta_2(2,c))})$ when $\delta_1(-, c) : q' \mapsto 2$;
- Crucially, the remaining case $\delta_1(-, c) : q \mapsto 3 - q$ is *excluded by aperiodicity.*

Next, we must transpose these ideas to the setting of the $\lambda_{\wp}$-calculus. We define the term of type $\mathtt{Str}_\Sigma[A] \multimap \mathtt{Str}_\Pi$ meant to compute our sequential function as

$$\lambda^{\circ} s.\, \lambda^{\rightarrow} f_{a_1}. \ldots \lambda^{\rightarrow} f_{a_{|\Pi|}}.\, \mathtt{out}\,(s\,\mathtt{trans}_{c_1}\ \ldots\ \mathtt{trans}_{c_{|\Sigma|}})$$

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

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

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 2.7. 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 $\mathtt{Str}_\Sigma$ 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 taking pairs of type $T \otimes T$ (although our $\lambda_{\wp}$-calculus has no actual $\otimes$ connective). Without further ado, let us program:

- $\mathtt{concat} : \lambda^\circ w.\, \lambda^\circ w'.\, \lambda^\circ x.\, w\,(w'\,x) : S \multimap S \multimap o \multimap o = S \multimap S \multimap S = S \multimap T$ plays the roles of both the concatenation operator and of $w \mapsto \kappa_w$ (thanks to currying)
- $\mathtt{zeta} = \lambda^\circ w'.\, \lambda^\circ x.\, x : S \multimap o \multimap o = T$
- $u_q = \delta_2(q,c)^\dagger(\vec{f_\Pi})$ (cf. Definition 2.4) represents the output word $\delta_2(q,c)$ that corresponds to a given input letter $c \in \Sigma$ and state $q \in Q = \{1,2\}$
- case $\delta_1(q,c) = q$: $\mathtt{trans}_c = \lambda^\circ k.\, \lambda^\circ h.\, \lambda^\circ g.\, k\,(\lambda^\circ y.\, h\,(\mathtt{concat}\,u_1\,y))\,(\lambda^\circ z.\, g\,(\mathtt{concat}\,u_2\,z))$ – observe that if we wanted to handle the excluded case $\delta_1(q,c) = 3-q$, we would write a similar term with the occurrences of h and g exchanged $(\lambda^\circ k.\, \lambda^\circ h.\, \lambda^\circ g.\, k\,(\lambda^\circ y.\, g\,\ldots)\,(\lambda^\circ z.\, h\,\ldots))$ violating the non-commutativity requirement (contrast with the proof of Theorem 5.4);
- case $\delta_1(q,c) = 1$: $\mathtt{trans}_c = \lambda^\circ k.\, \lambda^\circ h.\, \lambda^\circ g.\, k\,(\mathtt{concat}\,(\mathtt{concat}\,(h\,u_1)\,(g\,u_2)))\,\mathtt{zeta}$
- case $\delta_1(q,c) = 2$: $\mathtt{trans}_c = \lambda^\circ k.\, \lambda^\circ h.\, \lambda^\circ g.\, k\,\mathtt{zeta}\,(\mathtt{concat}\,(\mathtt{concat}\,(h\,u_1)\,(g\,u_2)))$
- $\mathtt{out} = \lambda^\circ j.\, j\,(\lambda^\circ h.\, \lambda^\circ g.\, \mathtt{concat}\,(h\,v_1)\,(g\,v_2))\,(\lambda^\circ x.\, x)\,\mathtt{zeta}$, 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 terms are well-typed – in particular that they enjoy the requisite non-commutativity conditions – and that they have the expected computational behavior. Note in particular 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 linearity (strictly speaking, affine typing) and non-commutativity, with the latter depending on the former as already mentioned. One could wonder whether linearity 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 *$\lambda\ell$-calculus* (by analogy with Clairambault et al.'s $\lambda\ell\mathbf{Y}$-calculus [11]) – 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: 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 $\lambda\ell$-calculus, the linear 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 $\lambda\ell$-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 $\lambda\ell$-calculus to the simply typed λ-calculus which "forgets linearity".

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 $\lambda\ell$-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 $\lambda\ell$-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)**.** *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* $\lambda\ell$*-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 $\lambda\ell$-term. One can also show that Lemma 4.4 applies to the $\lambda\ell$-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 $(g_q)_{q \in Q} \mapsto (g_{\sigma(q)} \circ \kappa_{\delta_2(\sigma(q),c)})_{q \in Q}$ where $\sigma = (\delta_1(-, c))^{-1}$ – the latter is well-defined because the group assumption means that $\delta_1(-, c)$ is a permutation of Q. The final output is obtained at the end as the concatenation $g_{q_1}(F(q_1)) \ldots g_{q_n}(F(q_n))$ where $Q = \{q_1, \ldots, q_n\}$ (with an arbitrary enumeration of Q).

The elaboration of the corresponding $\lambda\ell$-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 $\delta_1(-, c)$ 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 $A, B ::= \ldots \mid A \,\&\, B$ to the grammar of types and $t, u ::= \ldots \mid \langle t, u\rangle \mid \pi_1\, t \mid \pi_2\, t$ for terms, with the typing rules

$$\frac{\Gamma \mid \Delta \vdash t : A \quad \Gamma \mid \Delta \vdash u : B}{\Gamma \mid \Delta \vdash \langle t, u\rangle : A \,\&\, B} \qquad \frac{\Gamma \mid \Delta \vdash t : A_1 \,\&\, A_2}{\Gamma \mid \Delta \vdash \pi_i\, t : A_i} \qquad \text{(see [35, §4])}$$

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 in the introduction the need to prevent the existence of a $\lambda\wp$-term of type $\mathtt{Bool} \multimap \mathtt{Bool}$ for negation[8]. However, if we use the additive conjunction to define the type $\mathtt{Bool}^{\&} = (o \,\&\, o) \multimap o$, the following are well-typed $\lambda\wp^{\&}$-terms:

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

This admits a straightforward generalization:

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

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

Proof idea. Take $A = \mathtt{Fin}^{\&}(|M|)$ 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 $\mathtt{Tree}_{(2,2,0)} = (o \multimap o \multimap o) \to (o \multimap o \multimap o) \to o \to o$.

We shall not go into the details of tree automata there, 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 $\lambda\ell$-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 $\lambda\ell$-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. [7]).

[8] Actually, the $\lambda\wp$-calculus does admit a term for negation: $\lambda^{\circ} b.\ b\ \mathtt{false}\ \mathtt{true} : \mathtt{Bool}[\mathtt{Bool}] \multimap \mathtt{Bool}$. The heterogeneity of the input and output types means that this $\lambda\wp$-term does not contradict Theorem 1.8 and cannot be iterated by a Church-encoded string.

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 4.4). ◀

This suggests a connection between these λ-definable string functions and automata theory; some partial results in that direction can be found in an old (and partially obsolete) preprint by the first author [33]. 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 [32] 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 linearity and non-commutativity. For string functions, it turns out that commutative linearity suffices to make a difference compared to the simply typed λ-calculus.

▷ Claim 6.2 (to be proved in the 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*[9] [13] (a.k.a. *regular functions*[10]) in the *$\lambda\ell$-calculus* and the *FO transductions* in the *$\lambda\wp$-calculus*.

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
$\lambda\ell$	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 "linearity" itself has been used to describe the *copyless assignment* condition on streaming string transducers [4], a machine model for MSO transductions, e.g. "updates should make a linear use of registers" [14, §5]. 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.

[9] MSO stands for Monadic Second-Order Logic while FO stands for First-Order Logic, cf. the introduction.
[10] 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 4.5) and the regular (MSO-definable) functions, there is a widely 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.)

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 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.
5. Jean-Marc Andreoli, Gabriele Pulcini, and Paul Ruet. Permutative Logic. In *Computer Science Logic*, Lecture Notes in Computer Science, pages 184–199. Springer, Berlin, Heidelberg, August 2005. doi:10.1007/11538363_14.
6. 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/.
7. Mikołaj Bojańczyk. Algebra for trees. To appear in the Handbook of Automata Theory. URL: https://www.mimuw.edu.pl/~bojan/papers/treealgs.pdf.
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. Polyregular Functions. *CoRR*, abs/1810.08760, October 2018. arXiv:1810.08760.
10. 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.
11. Pierre Clairambault, Charles Grellois, and Andrzej Murawski. Linearity in Higher-order Recursion Schemes. *Proceedings of the ACM on Programming Languages*, 2(POPL):39:1–39:29, December 2017. doi:10.1145/3158127.
12. Volker Diekert, Manfred Kufleitner, and Benjamin Steinberg. The Krohn-Rhodes Theorem and Local Divisors. *Fundamenta Informaticae*, 116(1-4):65–77, 2012. doi:10.3233/FI-2012-669.
13. 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.
14. Emmanuel Filiot and Pierre-Alain Reynier. Transducers, Logic and Algebra for Functions of Finite Words. *ACM SIGLOG News*, 3(3):4–19, August 2016. URL: https://dl.acm.org/doi/10.1145/2984450.2984453, doi:10.1145/2984450.2984453.
15. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987. doi:10.1016/0304-3975(87)90045-4.
16. 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.
17. Jean-Yves Girard. Light Linear Logic. *Information and Computation*, 143(2):175–204, June 1998. doi:10.1006/inco.1998.2700.
18. 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.

19 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/`.

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

21 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`.

22 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`.

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

24 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`.

25 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, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.FSTTCS.2019.45`.

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

27 Olivier Laurent. Polynomial time in untyped elementary linear logic. *Theoretical Computer Science*, October 2019. `doi:10.1016/j.tcs.2019.10.002`.

28 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`.

29 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`.

30 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`.

31 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`.

32 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, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.STACS.2019.2`.

33 Lê Thành Dũng Nguyễn. Typed lambda-calculi and superclasses of regular functions. *CoRR*, abs/1907.00467, 2019. `arXiv:1907.00467`.

34 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`.

35 Jeff Polakow and Frank Pfenning. Natural Deduction for Intuitionistic Non-commutative Linear Logic. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Jean-Yves Girard, editors, *Typed Lambda Calculi and Applications*, volume 1581, pages 295–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. `doi:10.1007/3-540-48959-2_21`.

36 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.

37 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.

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

39 Sylvain Salvati. Recognizability in the Simply Typed Lambda-Calculus. In Hiroakira Ono, Makoto Kanazawa, and Ruy de Queiroz, editors, *Logic, Language, Information and Computation*, volume 5514, pages 48–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-02261-6_5.

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

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

42 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.

43 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.

44 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.

A Reminder: the Krohn–Rhodes decomposition theorems, from transformation monoids to sequential transducers

While we have not found a source with a proof for the precise versions of the Krohn–Rhodes theorem for sequential functions that we use, all the material covered in this subsection is well-known among practitioners of automata theory. In other words, we make no claim to originality.

▶ **Definition A.1.** *A* transformation monoid (X, M) *consists of a set* X*, a monoid* M *and a* right action *of* M *on* X *(kept implicit in the notation* (X, M)*, and denoted by* $(x, m) \mapsto x \cdot m$*). It is* finite *when both* X *and* M *are finite.*

Typical transformation monoids are obtained by considering pairs (Q, T) such that Q is the state space of some transducer (Q, δ, q_I, F) and T is its transition monoid, acting on Q via function application.

▶ **Definition A.2.** *Let* (X, M) *and* (Y, N) *be two transformation monoids. Their* wreath product *is a transformation monoid* $(X, M) \wr (Y, N) = (X \times Y, W)$ *where:*

- *the underlying set of the monoid* W *is* $M^Y \times N$*;*
- *the right action of* $(f, n) \in W = M^Y \times N$ *on* $(x, y) \in X \times Y$ *is* $(x, y) \cdot (f, n) = (x \cdot f(y), y \cdot n)$*;*
- *the multiplication on* W *is* $(f, n)(g, k) = ((y \mapsto f(y)g(y \cdot n)), nk)$ *– it is chosen so that the above item is a legitimate monoid action.*

▶ **Proposition A.3.** *The wreath product of transformation monoids is associative up to canonical isomorphism.*

Proof sketch. We give a direct description of $(X, M) \wr (Y, N) \wr (Z, P) = (X \times Y \times Z, W)$:
- the underlying set of the monoid W is $M^{Y\times Z} \times N^Z \times P$ – note that it is canonically isomorphic to $(M^Y \times N)^Z \times P$;
- the right action is $(x, y, z) \cdot (f, g, p) = (x \cdot f(y, z), y \cdot g(z), z \cdot p)$;
- the multiplication on W is $(f, g, p)(f', g', p') = (((y, z) \mapsto f(y, z)f'(y \cdot n, z \cdot p)), (z \mapsto g(z)g'(z \cdot p)), pp')$. ◀

▶ **Definition A.4.** *A transformation monoid* (X, M) strongly divides (Y, N) *if there exists a submonoid* $N' \leq N$*, a surjective morphism* $\varphi : N' \twoheadrightarrow M$ *and a surjection* $s : Y \twoheadrightarrow X$ *such that for all* $y \in Y$ *and* $n' \in N'$*,* $s(y \cdot n') = s(y) \cdot \varphi(n')$*.*

A monoid M divides N *if* M *is the homomorphic image of a submonoid of* N*.*

▶ **Proposition A.5.** *A finite monoid is aperiodic if and only if it there are no non-trivial groups that divide it.*

Proof. Let M be a finite monoid. Suppose that for $x \in M$, there is no $n \in \mathbb{N}$ such that $x^n = x^{n+1}$; then by finiteness, $(x^i)_{i\in\mathbb{N}}$ must be ultimately periodic with period $k \geq 2$, and one can define a surjective morphism from the submonoid generated by x to the cyclic group of order k by sending x to the latter's generator. The converse follows a similar reasoning (recall that every non-trivial group contains a non-trivial cyclic subgroup). ◀

▶ **Theorem A.6** (Krohn–Rhodes with strong divisors [12, Theorem 4.1])**.** *Every finite transformation monoid* (X, M) *strongly divides some wreath product* $(Y_1, N_1) \wr \ldots \wr (Y_n, N_n)$ *where each* (Y_k, N_k) *is either:*
- *the* flip-flop $(Y_k, N_k) = (\{1, 2\}, \{\mathrm{id}_{\{1,2\}}, (x \mapsto 1), (x \mapsto 2)\})$ *(with the action* $x \cdot f = f(x)$ *and the monoid multiplication* $fg = g \circ f$*);*
- *a finite* group *dividing* M *acting on itself by right multiplication.*

In particular, if M is aperiodic*,* (X, M) *strongly divides a wreath product of several copies of the flip-flop transformation monoid.*

▶ Remark A.7. We can also require G above to be a *simple* group. This is the statement given in [12], but group simplicity is not needed for our purposes. (To be more precise, every finite group divides a wreath product of its Jordan–Hölder factors.)

▶ Remark A.8. Let $(Y, N) = (Y_1, N_1) \wr \ldots \wr (Y_n, N_n)$. In both the flip-flop and group cases, the action of N_k on Y_k is *faithful*, i.e. two distinct elements of N_k act differently on at least one element of Y_k. Furthermore, the wreath product of faithful transformation monoids is faithful. Therefore, one can safely identify N with a submonoid of $Y \to Y$.

Now let us relate this wreath product operation to sequential functions. This is sufficient to derive Theorem 4.8 and Theorem 5.2 as corollaries of Theorem A.6.

▶ **Proposition A.9.** *Let* (Q, δ, q_I, F) *be a sequential transducter with transition monoid* T *describing a function* $f : \Sigma^* \to \Pi^*$*. Suppose that* (Q, T) *strongly divides some faithful transformation monoid* $(X, M) \wr (Y, N)$*. Then there is an alphabet* Ξ *and transducers*

$$(X, \delta_X, x_I, F_X) : \Sigma^* \to \Xi^* \qquad \text{and} \qquad (Y, \delta_Y, y_I, F_Y) : \Xi^* \to \Pi^*$$

with respective transition monoid T_X *and* T_Y *corresponding to sequential functions* $f_X : \Sigma^* \to \Xi^*$ *and* $f_Y : \Xi^* \to \Pi^*$ *such that* $f = f_X \circ f_Y$*. Moreover, there are injective homomorphisms* $T_X \hookrightarrow M$ *and* $T_Y \hookrightarrow N$*.*

Proof. Let (Q, δ, q_I, F) be the transducer under scrutiny. Let $K \subseteq M^Y \times N$ such that $\varphi : K \twoheadrightarrow T$, $s : X \times Y \twoheadrightarrow Q$ be the maps witnessing that (Q, T) strongly divides $(X, M) \wr (Y, N)$. We choose a pair (x_I, y_I) such that $s(x_I, y_I) = q_I$ and, for each $a \in \Sigma$, we choose an element $(g_a, n_a) \in M^Y \times N$ which is mapped by φ to $\delta_1(-, a) \in T$. Set $\Xi = (\Sigma \uplus \{*\}) \times Y$, $(x_I, y_I) = s^{-1}(x, y)$ and

$$\begin{array}{lcllcl} F_Y(y) & = & (*, y) & F_X(x) & = & \epsilon \\ \delta_Y(y, a) & = & (y \cdot m_a, y) & \delta_X(x, (a, y)) & = & (x \cdot g_a(y), \delta_2(s(x), a)) \\ & & & \delta_X(x, (*, y)) & = & (x, F(s(x, y))) \end{array}$$

We leave checking that this defines transducers with the expected properties to the reader. ◀

This generalizes to n-fold wreath products in the expected way.

▶ **Proposition A.10.** *Let T be the transition monoid of a sequential transducer with state space Q computing the function $f : \Sigma^* \to \Pi^*$. Suppose that (Q, T) strongly divides some wreath product $(X, M) = (X_1, M_1) \wr \ldots \wr (X_n, M_n)$ of faithful transformation monoids. Then f admits a decomposition $f = f_1 \circ \ldots \circ f_n$ (with $f_i : \Xi_i^* \to \Xi_{i-1}^*$, $\Xi_0 = \Pi$ and $\Xi_n = \Sigma$) such that for each $i \in \{1, \ldots, n\}$, f_i is computed by a sequential transducer whose transformation monoid embeds in M_i and with state space X_i.*

Proof. By induction starting from $n = 1$.

- For $n = 1$, let $\varphi : K \twoheadrightarrow T$ and $s : X \twoheadrightarrow Q$ be the maps witnessing that (Q, T) strongly divides (X, M). Let x_I be such that $s(x_I) = q_I$, and, for each $a \in \Sigma$, pick an element $m_a \in K$ such that $\varphi(m_a) = \delta_1(-, a)$. Then, letting (Q, δ, q_I, F) being the transducer under scrutiny, a suitable transducer (X, δ', x_I, F') is defined by setting $\delta'(x, a) = (x \cdot m_a, \delta_2(s(x), a))$ and $F'(x) = F(s(x))$.
- For $n > 1$, use Proposition A.9 and the induction hypothesis.

◀

Proof of Theorems 4.8 and 5.2. Let (Q, δ, q_I, F) be a transducter computing a certain sequential function $f : \Sigma^* \to \Pi^*$ and let T be its transition monoid. By Theorem A.6, there is a transformation monoid (Y, N) which can be written as a wreath product $(Y, N) = (Y_1, N_1) \wr \ldots \wr (Y_k, N_k)$ such that (Q, T) strongly divides (Y, N), and the (Y_i, N_i) are either flip-flops or groups (the latter case being ruled out for Theorem 4.8, thanks to Proposition A.5). By applying Proposition A.10, we may obtain transducers $\mathcal{T}_i$ implementing sequential functions $f_i : \Xi_i^* \to \Xi_{i+1}^*$ such that $\Xi_0 = \Sigma$, $\Xi_k = \Pi$ and $f = f_{k-1} \circ \ldots \circ f_0$. Furthermore, we know that the state space of $\mathcal{T}_i$ is Y_i and that the corresponding transition monoid T_i embeds into N_i. Recalling that "being aperiodic" and "being a finite subgroup" are properties stable under homomorphic embeddings, we know that either Y_i has cardinality 2 and T_i is aperiodic with two states or T_i is a group (a trivial group if T was aperiodic), thus we may conclude. ◀

B Omitted proofs

B.1 Proof of Lemma 3.1

The β-normal η-long form of a closed $\lambda\wp$-term t of type $\mathtt{Str}_\Sigma[A] \multimap (o \multimap o \multimap o) \multimap o)$ (by definition of $\mathtt{Bool}$) is

$$t =_{\beta\eta} \lambda^\circ s.\, \lambda^\circ x.\, \lambda^\circ y.\, z\, t_1 \ldots t_n \quad \text{where} \quad \varnothing \mid s : \mathtt{Str}_\Sigma[A], x : o, y : o \vdash z\, t_1 \ldots t_n : o$$

If $z \in \{x, y\}$, then $n = 0$ and t is a constant function from strings to booleans: the statement of Lemma 3.1 is true with $g_c = \lambda^\circ a.\, a$ and $h = \lambda^\circ f.\, \lambda^\circ x.\, \lambda^\circ y.\, z$ (the latter is the constant

function equal to either `true` or `false` depending on whether $z = x$ or $z = y$) or, said explicitly, $t =_{\beta\eta} \lambda^\circ s.\, (\lambda^\circ f.\, \lambda^\circ x.\, \lambda^\circ y.\, z)\, (s\, (\lambda^\circ a.\, a)\, \dots\, (\lambda^\circ a.\, a))$.

In the remaining case $z = s$, one can take $h = \lambda^\circ f.\, \lambda^\circ x.\, \lambda^\circ y.\, f\, t_{|\Sigma|+1}\, \dots\, t_n$ and, for $i \in \{1, \dots, |\Sigma|\}$, $g_{c_i} = t_i$.

B.2 Proof of Lemma 4.2

Since L is star-free, $L = \varphi^{-1}(P)$ for some $\varphi \in \mathrm{Hom}(\Sigma^*, M)$ and $P \subseteq M$, where M is an aperiodic monoid. Here is a sequential transducer computing χ_L: $Q = M$, $\delta(m, c) = (m\varphi(c), \varepsilon)$ and $F(m) = 1$ if $m \in P$ and $F(m) = \varepsilon$ otherwise. Its transition monoid is $\varphi(\Sigma^*) \subseteq M$, which is aperiodic.

B.3 Proof of Lemma 4.3

The $\lambda\wp$-term in question is $\lambda^\circ s.\, s\, (\lambda^\circ x.\, \mathtt{true})\, \mathtt{false}$.

B.4 Proof of Lemma 4.4

The lemma follows from the more usual stability of typing judgments under type substitution. We write $\Gamma[A]$ and $\Delta[B]$ for the obvious extension of Notation 1.2 to contexts.

▶ **Lemma B.1.** *If $\Gamma \mid \Delta \vdash t : A$, then, for every type B, we have $\Gamma[B] \mid \Delta[B] \vdash t : A[B]$.*

Proof. Routine induction on the typing derivation. ◀

Picturing Lemma B.1 as an admissible typing rule (dashed inference line), we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset \mid \emptyset \vdash t : A[T] \multimap B}{\emptyset \mid \emptyset \vdash t : A[T[U]] \multimap B[U]} \qquad \dfrac{}{\emptyset \mid x : A[T[U]] \vdash x : A[T[U]]}}{\emptyset \mid x : A[T[U]] \vdash t\, x : B[U]} \qquad \emptyset \mid \emptyset \vdash u : B[U] \multimap C}{\emptyset \mid x : A[T[U]] \vdash u\, (t\, x) : C}}{\emptyset \mid \emptyset \vdash \lambda^\circ x.\, u\, (t\, x) : A[T[U]] \multimap C}$$