



HAL
open science

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

Lê Thành Dũng Nguyễn, Pierre Pradic

► **To cite this version:**

Lê Thành Dũng Nguyễn, Pierre Pradic. Implicit automata in typed λ -calculi I: aperiodicity in a non-commutative logic. 2020. hal-02476219v1

HAL Id: hal-02476219

<https://hal.science/hal-02476219v1>

Preprint submitted on 12 Feb 2020 (v1), last revised 13 Feb 2023 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1 Implicit automata in typed λ -calculi I: 2 aperiodicity in a non-commutative logic

3 Lê Thành Dũng Nguyễn 

4 Université publique¹, France

5 <https://nguyentito.eu/>

6 nltd@nguyentito.eu

7 Pierre Pradic

8 Department of Computer Science, University of Oxford, United Kingdom

9 <https://www.cs.ox.ac.uk/people/pierre.pradic/>

10 pierre.pradic@cs.ox.ac.uk

11 — Abstract —

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

18 **2012 ACM Subject Classification** Theory of computation \rightarrow Algebraic language theory; Theory of
19 computation \rightarrow Linear logic

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

21 **Acknowledgements** The initial trigger for this line of work was twofold: the serendipitous rediscovery
22 of Hillebrand and Kanellakis’s theorem by Damiano Mazza, Thomas Seiller and the first author, and
23 Mikołaj Bojańczyk’s suggestion to the second author to look into connections between transducers
24 and linear logic. Célia Borlido proposed looking at star-free languages at the Topology, Algebra and
25 Categories in Logic 2019 summer school.

26 During its long period of gestation, this work benefited from discussions with many other people:
27 Pierre Clairambault, Amina Doumane, Marie Fortin, Jérémy Ledent, Paolo Pistone, Lorenzo Tortora
28 de Falco, Noam Zeilberger and others that we may have forgotten to mention.

29
30 The remainder of the title page has been left blank to make the 12-page limit “excluding
31 references and the front page(s) (authors, affiliation, keywords, abstract, ...)” easier to
32 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 $\mathbf{Str}_\Sigma[A] \rightarrow \mathbf{Bool}$ 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: $A, B ::= o \mid A \rightarrow B$, and use the *Church encodings* of booleans and strings:

$$\mathbf{Bool} = o \rightarrow o \rightarrow o \qquad \mathbf{Str}_\Sigma = (o \rightarrow o) \rightarrow \dots \rightarrow (o \rightarrow o) \rightarrow o \rightarrow o$$

with $|\Sigma|$ arguments of type $(o \rightarrow o)$, where Σ is a finite alphabet. Moreover, given any other chosen type A , one can form the type $\mathbf{Str}_\Sigma[A]$ 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 \mathbf{Str}_Σ can also be seen as a term of type $\mathbf{Str}_\Sigma[A]$. (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 $\mathbf{Str}_\Sigma[A] \rightarrow \mathbf{Bool}$ in the simply typed λ -calculus defines a predicate on strings, i.e. a language $L \subseteq \Sigma^*$.

Although little-known², 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 longer³.

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].

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

74 ► **Theorem 1.3** (classical). *A language $L \subseteq \Sigma^*$ is regular if and only if for some finite monoid*
 75 *M , some subset $P \subseteq M$ and some monoid morphism $\varphi \in \text{Hom}(\Sigma^*, M)$, $L = \varphi^{-1}(P)$.*

76 Similarly, the seminal work of Schützenberger, Petrone, McNaughton and Papert in the
 77 1960s (see [41] for a historical discussion) has led to many equivalent definitions for star-free
 78 languages, with the algebraic notion of *aperiodicity* playing a key role:

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

81 ► **Theorem 1.5** (cf. [41]). *For a language $L \subseteq \Sigma^*$, the following conditions are equivalent:*
 82 ■ *L is defined by some star-free regular expression: $E, E' ::= \emptyset \mid \{a\} \mid E \cup E' \mid E \cdot E' \mid E^c$*
 83 *where a can be any letter in Σ and E^c denotes the complement of E ($\llbracket E^c \rrbracket = \Sigma^* \setminus \llbracket E \rrbracket$);*
 84 ■ *$L = \varphi^{-1}(P)$ for some finite and aperiodic monoid M , some subset $P \subseteq M$ and some*
 85 *monoid morphism $\varphi \in \text{Hom}(\Sigma^*, M)$;*
 86 ■ *L is recognized by a deterministic finite automaton whose transition monoid is aperiodic;*
 87 ■ *L is definable in first-order logic.*

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

91 **Non-commutative affine types** Monoids appear in typed λ -calculi when one looks at the
 92 functions from a type A to itself, i.e. at the (closed) terms of type $A \rightarrow A$. At first glance, it
 93 seems difficult indeed to enforce the aperiodicity of such monoids via a type system. For
 94 instance, one needs to rule out $\text{not} = \lambda b. \lambda x. \lambda y. b y x : \text{Bool} \rightarrow \text{Bool}$ since it has period two.
 95 Observe that not essentially *exchanges* the two arguments of b ; to exclude it, we are therefore
 96 led to require functions to *use their arguments in the same order that they are given in*.

97 It is well-known that in order to make such a *non-commutative* λ -calculus work – in
 98 particular to ensure that non-commutative λ -terms are closed under β -reduction – one
 99 needs to make the type system *affine*, that is, to restrict the duplication of data. This is
 100 achieved by considering a type system based on Girard’s linear⁴ logic [15], a system whose
 101 “resource-sensitive” nature has been previously exploited in ICC [18, 17]. Not coincidentally,
 102 the theme of non-commutativity first appeared in a form of linear logic *ante litteram*, namely
 103 the Lambek calculus [26], and resurfaced shortly after the official birth of linear logic: it is
 104 already mentioned by Girard in a 1987 colloquium [16].

105 We shall therefore introduce and use a variant of Polakow and Pfenning’s Intuitionistic
 106 Non-Commutative Linear Logic [35, 36], making a distinction between two kinds of function
 107 arrows: $A \multimap B$ and $A \rightarrow B$ are, respectively, the types of affine functions and non-affine
 108 functions from A to B . Accordingly:

109 ► **Definition 1.6.** *A type is said to be purely affine if it does not contain the ‘ \multimap ’ 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.

4 Aperiodicity in a non-commutative logic

110 In our system that we call the λ_{\wp} -calculus, the types of Church encodings become

$$111 \quad \mathbf{Bool} = o \multimap o \multimap o \quad \mathbf{Str}_{\Sigma} = (o \multimap o) \rightarrow \dots \rightarrow (o \multimap o) \rightarrow (o \multimap o)$$

112 where \mathbf{Str}_{Σ} has $|\Sigma|$ arguments of type $(o \multimap o)$. Setting $\mathbf{true} = \lambda^{\circ}x. \lambda^{\circ}y. x : \mathbf{Bool}$ and
113 $\mathbf{false} = \lambda^{\circ}x. \lambda^{\circ}y. y : \mathbf{Bool}$ for the rest of the paper, we can now state our main result:

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

116 However, if we use the *commutative* variant of the λ_{\wp} -calculus instead, then what we get is
117 the class of regular languages (Theorem 5.1), just as in Hillebrand and Kanellakis’s theorem.

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

123 **Proof strategy** As usual in implicit computational complexity, the proof of Theorem 1.7
124 consists of a *soundness* part – “every λ_{\wp} -definable language is star-free” – and an *extensional*
125 *completeness* part – the converse implication. In our case, soundness is a corollary of the
126 following property of the purely affine fragment of the λ_{\wp} -calculus – what one might call the
127 *planar affine λ -calculus* (cf. [1, 44]):

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

131 Extensional completeness turns out here to be somewhat deeper than the “programming
132 exercise of limited theoretical interest” [29, p. 137] that one generally finds in ICC. Indeed, we
133 have only managed to encode star-free languages in the λ_{\wp} -calculus by relying on a powerful
134 tool from semigroup theory: the *Krohn–Rhodes decomposition* [24].

135 **Plan of the paper** After having defined the λ_{\wp} -calculus in §2, we prove Theorem 1.7:
136 soundness is treated in §3 and extensional completeness in §4. Then we discuss the analogous
137 results for the commutative variant of the λ_{\wp} -calculus and its extension with additives (§5),
138 and finally our plans for the next papers in the series (§6).

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

143 **2 Preliminaries: the λ_{\wp} -calculus and Church encodings**

144 The terms and types of the λ_{\wp} -calculus are defined by the respective grammars

$$145 \quad A, B ::= o \mid A \rightarrow B \mid A \multimap B \quad t, u ::= x \mid tu \mid \lambda^{\rightarrow}x. t \mid \lambda^{\circ}x. t$$

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

$$148 \quad (\lambda^{\rightarrow}x. t) u \longrightarrow_{\beta} t\{x := u\} \quad (\lambda^{\circ}x. t) u \longrightarrow_{\beta} t\{x := u\}$$

$$\begin{array}{c}
\frac{}{\Gamma \uplus \{x : A\} \mid \emptyset \vdash x : A} \quad \frac{\Gamma \mid \Delta \vdash t : A \rightarrow B \quad \Gamma \mid \emptyset \vdash u : A}{\Gamma \mid \Delta \vdash tu : B} \quad \frac{\Gamma \uplus \{x : A\} \mid \Delta \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\rightarrow} x. t : A \rightarrow B} \\
\\
\frac{}{\Gamma \mid x : A \vdash x : A} \quad \frac{\Gamma \mid \Delta \vdash t : A \multimap B \quad \Gamma \mid \Delta' \vdash u : A}{\Gamma \mid \Delta \cdot \Delta' \vdash tu : B} \quad \frac{\Gamma \mid \Delta \cdot (x : A) \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\circ} x. t : A \multimap B} \\
\\
\frac{\Gamma \mid \Delta \vdash t : A}{\Gamma \mid \Delta' \vdash t : A} \text{ when } \Delta \text{ is a subsequence of } \Delta'
\end{array}$$

■ **Figure 1** The typing rules of the λ_{\wp} -calculus.

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

150 The typing judgements make use of dual contexts (a common feature originating in [6]):
 151 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
 152 form $x : B$ (x being a variable and B a type), and Δ is an *ordered list* of bindings – this
 153 order is essential for non-commutativity. The typing rules are given in Figure 1, where $\Delta \cdot \Delta'$
 154 denotes the *concatenation of the ordered lists Δ and Δ'* . For both Γ, Γ', \dots and Δ, Δ', \dots we
 155 require each variable to appear at most once on the left of a colon.

156 ► **Remark 2.1.** Unlike Polakow and Pfenning’s system [35, 36], the λ_{\wp} -calculus:

- 157 ■ contains two function types instead of four⁵;
- 158 ■ is affine instead of linear (all functions can discard arguments) – this seems important to
 159 get enough expressive power for our purposes⁶.

160 ► **Remark 2.2.** Morally, the non-affine variables “commute with everything”. More formally,
 161 one could translate the λ_{\wp} -calculus into a non-commutative version of Intuitionistic Affine
 162 Logic whose exponential modality ‘!’ incorporates the customary rules

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

164 ► **Proposition 2.3.** *The λ_{\wp} -calculus enjoys subject reduction and admits normal forms (that
 165 is, every well-typed λ_{\wp} -term is convertible to a β -normal η -long one).*

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

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

171 ► **Definition 2.4.** *Let Σ be a finite alphabet, $w = w[1] \dots w[n] \in \Sigma^*$ be a string, and for each
 172 $c \in \Sigma$, let t_c be a λ_{\wp} -term. We abbreviate the family $(t_c)_{c \in \Sigma}$ as \vec{t}_{Σ} , and define the λ_{\wp} -term
 173 $w^{\dagger}(\vec{t}_{\Sigma}) = \lambda^{\circ} x. t_{w[1]} (\dots (t_{w[n]} x) \dots)$.*

⁵ Our ‘ \rightarrow ’ 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 λ_{\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.

6 Aperiodicity in a non-commutative logic

174 Given a total order on the alphabet $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$, the Church encoding of any string
175 $w \in \Sigma^*$ is $\bar{w} = \lambda^\rceil f_{c_1}. \dots \lambda^\rceil f_{c_{|\Sigma|}}. w^\dagger(\vec{f}_\Sigma)$.

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

177 **► Proposition 2.5.** *We reuse the notations of the above definition.*

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

180 *Furthermore, for any choice of variables $(f_c)_{c \in \Sigma}$, there is a bijection between the strings
181 over Σ and the λ_{\wp} -terms u such that $\{f_c : o \multimap o \mid c \in \Sigma\} \mid \emptyset \vdash u : o \multimap o$ and considered up
182 to $\beta\eta$ -conversion, given by $w \in \Sigma^* \mapsto w^\dagger(\vec{f}_\Sigma)$.*

183 *It follows from the above that $w \in \Sigma^* \mapsto \bar{w}$ is a bijection from Σ^* to the set of closed
184 λ_{\wp} -terms of type \mathbf{Str}_Σ modulo $\beta\eta$. Finally, we have $\bar{w} t_{c_1} \dots t_{c_{|\Sigma|}} \xrightarrow{\beta}^* w^\dagger(\vec{t}_\Sigma)$.*

185 **► Remark 2.6.** For $\Sigma = \{a\}$, η -conversion is necessary to identify $\lambda^\rceil f. f : \mathbf{Str}_\Sigma$ with \bar{a} .

186 **► Remark 2.7.** Our presentation of Church encodings stresses the role of *open terms*. Indeed,
187 it is important to note that when the context Γ of non-affine variables contains $f_c : o \multimap o$
188 for each $c \in \Sigma$, then any string can be represented as a term of type $o \multimap o$ in that context,
189 and such strings can even be concatenated by function composition. This gives us a kind of
190 *purely affine type of strings*, which will allow us in §4.2 to encode sequential transducers as
191 λ_{\wp} -terms of type $\mathbf{Str}_\Sigma[A] \multimap \mathbf{Str}_\Pi$ for some purely affine type A (compare Theorem 1.7).

3 Proof of soundness

192 We start by demonstrating how non-commutativity entails aperiodicity.

194 **Proof of Theorem 1.8.** The monoid structure is obvious (the identity element is $\lambda^x x$),
195 and it is well-known that in an affine type system, any purely affine type has finitely many
196 $\beta\eta$ -equivalence classes of closed inhabitants (here non-commutativity plays no role). Therefore,
197 the remainder of the proof concerns aperiodicity.

198 Let $t : A \multimap A$; our goal is to show that the sequence $t^n = t \circ \dots \circ t =_\beta \lambda^x x. t(\dots (tx)\dots)$
199 is eventually constant modulo $\beta\eta$. We shall do so by *induction on the size of A* . The type A
200 is *purely affine* by assumption, and can therefore be written as $B_1 \multimap \dots \multimap B_m \multimap o$. The
201 base case $m = 0$ being trivial, we assume $m \geq 1$. In this case, t has an η -long β -normal form

$$202 \quad t = \lambda^x x. \lambda^{\circ} y_1. \dots \lambda^{\circ} y_m. z u_1 \dots u_k$$

203 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
204 omit the non-affine context Γ which will always be empty during this proof). The abstraction
205 rule only allows introducing $\lambda^{\circ} y_i$ when $(y_i : B_i)$ is on the right, so by then Δ must have been
206 entirely emptied out by previous abstractions. This means that $\lambda^{\circ} y_1. \dots \lambda^{\circ} y_m. z u_1 \dots u_k$ is
207 a closed term, so in particular it contains no free occurrence of x : t is a constant function
208 from A to A . So in the case $z = y_i$ the sequence of iterations stabilizes from $n = 1$.

209 From now on we can assume that $z = x$, which entails $k = m$ since the variable x is of type
210 $A = B_1 \multimap \dots \multimap B_m \multimap o$ and we must have $x : A, y_1 : B_1, \dots, y_m : B_m \vdash x u_1 \dots u_k : o$.
211 By a straightforward induction, for $n \in \mathbb{N}$, t^n is $\beta\eta$ -convertible to

$$212 \quad \lambda^x x. \lambda^{\circ} y_1. \dots \lambda^{\circ} y_k. x u_1^{(n)} \dots u_k^{(n)} \quad \text{where} \quad \vec{u}^{(0)} = (y_1, \dots, y_k), \quad \vec{u}^{(n+1)} = \vec{u}^{(n)}[\vec{y} := \vec{u}]$$

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

$$215 \quad \vec{u}_i^{(n+1)} = \vec{u}_i^{(n)}[y_1 := u_1, \dots, y_k := u_k] \quad \text{for } i \in \{1, \dots, k\}$$

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

217 To reach a positive answer, let us define the partial function $\mu_{\bar{u}} : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$
 218 by $\mu_{\bar{u}}(i) = j \iff y_i \in \text{FV}(u_j)$. ($\text{FV}(u)$ denotes the set of free variables of u .) The
 219 relation on the right-hand side of the equivalence is indeed a partial function because of
 220 the affinity 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 $n \in \mathbb{N}$,
 221 $\text{FV}(u_i^{(n)}) = \{y_j \mid (\mu_{\bar{u}})^n(j) = i\}$.

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

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

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

229 Next, let $i \in \{1, \dots, k\}$. Recall that our goal is to prove that $(u_i^{(n)})_{n \in \mathbb{N}}$ is eventually
 230 constant modulo $\beta\eta$. The simple case is when $i \notin (\mu_{\bar{u}})^N(\{1, \dots, k\})$: $u_i^{(N)}$ has no free
 231 variables, so $u_i^{(N+1)} = u_i^{(N)}[\bar{y} := \bar{u}] = u_i^{(N)}$. For the remainder of the proof we assume
 232 otherwise, that is, we take i in the range of $(\mu_{\bar{u}})^N$.

233 First, $\bar{u}^{(n+1)} = \bar{u}[\bar{y} := \bar{u}^{(n)}]$ because parallel substitution is associative⁷. Thus,

$$234 \quad \forall n \in \mathbb{N}, u_i^{(N+n+1)} = u_i \left[y_j := u_j^{(N+n)} \text{ for } j \in \{1, \dots, k\} \text{ such that } \mu_{\bar{u}}(j) = i \right]$$

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

$$238 \quad u_i^{(N+n+1)} = r_i[y_i := u_i^{(N+n)}] \quad \text{where} \quad r_i = u_i \left[y_j := u_j^{(N)} \text{ for } j \neq i \text{ s.t. } \mu_{\bar{u}}(j) = i \right]$$

239 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
 240 the sequence of iterates of the λ° -term $\lambda^\circ y_i. r_i : B_i \multimap B_i$ is eventually constant modulo $\beta\eta$.
 241 Since the type B_i has size strictly smaller than A , the induction hypothesis applies. (It is
 242 clear that $\lambda^\circ y_i. r_i$ is closed, but one should check that it is well-typed; to do so, one convenient
 243 observation is that the $u_j^{(N)}$ are closed (because $j \notin (\mu_{\bar{u}})^N(\{1, \dots, k\})$) and well-typed (as
 244 closed subterms of a reduct of the N -fold composition t^N .) ◀

245 After this, we must analyze the form of a closed λ° -term of type $\text{Str}_\Gamma[A] \multimap \text{Bool}$.

246 ► **Lemma 3.1.** *Let $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$ be a finite alphabet, A be a purely affine type and
 247 $t : \text{Str}_\Gamma[A] \multimap \text{Bool}$ be a closed λ° -term. Then there exist some closed λ° -terms $g_c : A \multimap A$
 248 for $c \in \Gamma$ and $h : (A \multimap A) \multimap \text{Bool}$ such that $t =_{\beta\eta} \lambda^\circ s. h(s g_{c_1} \dots g_{c_{|\Sigma|}})$.*

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

250 Reusing the notations of this lemma, let us define

$$251 \quad \varphi : w = w[1] \dots w[n] \in \Sigma^* \mapsto \lambda^\circ x. g_{w[1]}(\dots (g_{w[n]} x)) \in \{v \mid v : A \multimap A\} / =_{\beta\eta}$$

252 φ is a monoid morphism and $\varphi^{-1}(\{v \mid h v =_{\beta\eta} \text{true}\})$ is none other than the language defined
 253 by the $t : \text{Str}_\Sigma[A] \multimap \text{Bool}$ in the lemma. Since the codomain of φ is finite and aperiodic by
 254 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 $\vec{y} \cap (\text{FV}(\vec{t}_1) \setminus \vec{x}) = \emptyset$.

255 4 Expressiveness of the $\lambda\wp$ -calculus

256 We now turn to the *extensional completeness* part in Theorem 1.7. To prove it, the most
 257 convenient way that we have found is to take a detour through automata that compute an
 258 output string instead of a single bit (acceptance/rejection). We will define the notion of
 259 *aperiodic sequential function*, and then establish that:

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

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

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

266 ► **Lemma 4.3.** *There exists a $\lambda\wp$ -term of type $\text{Str}_{\{1\}}[\text{Bool}] \multimap \text{Bool}$ that tests whether its
 267 input string is non-empty.*

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

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

272 4.1 Reminders on automata theory

273 Sequential transducers are among the simplest models of automata with output. They are
 274 deterministic finite automata which can append a word to their output at each transition,
 275 and at the end, they can add a suffix to the output depending on the final state. The
 276 definition is classical; a possible reference is [38, Chapter V].

277 ► **Definition 4.5.** *A sequential transducer with input alphabet Σ and output alphabet Π
 278 consists of a set of states Q , a transition function $\delta : Q \times \Sigma \rightarrow Q \times \Pi^*$, an initial state
 279 $q_I \in Q$, and a final output function $F : Q \rightarrow \Pi^*$. We abbreviate $\delta_i = \pi_i \circ \delta$ for $i \in \{1, 2\}$,
 280 where $\pi_1 : Q \times \Pi^* \rightarrow Q$ and $\pi_2 : Q \times \Pi^* \rightarrow \Pi^*$ are the projections of the product.*

281 *Given an input string $w = w[1] \dots w[n] \in \Sigma^*$, the run of the transducer over w is the
 282 sequence of states $q_0 = q_I, q_1 = \delta_1(q_0, w[1]), \dots, q_n = \delta_1(q_{n-1}, w[n])$. Its output is obtained
 283 as the concatenation $\delta_2(q_0, w[1]) \cdot \dots \cdot \delta_2(q_{n-1}, w[n]) \cdot F(q_n)$.*

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

286 ► **Definition 4.6.** *The transition monoid of a sequential transducer is the submonoid of
 287 $Q \rightarrow Q$ (endowed with reverse function composition: $fg = g \circ f$) generated by the maps
 288 $\{\delta_1(-, c) \mid c \in \Sigma\}$ (where $\delta_1(-, c)$ stands for $q \mapsto \delta_1(q, c)$).*

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

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

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

298 ► **Remark 4.9.** This is not the standard way to state this theorem, though one may find it in
 299 the literature, usually without proof (e.g. [9, §1.1]); see [8] for a tutorial containing a proof
 300 sketch of this version. In Appendix A, we show how Theorem 4.8 follows from the more
 301 usual statement on wreath products of monoid actions.

302 4.2 Encoding aperiodic sequential transducers

303 Thanks to the Krohn–Rhodes decomposition and to the fact that the string functions
 304 definable in the $\lambda\wp$ -calculus (as specified by Theorem 4.1) are closed under composition (by
 305 Lemma 4.4), the following entails Theorem 4.1, thus concluding our completeness proof.

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

309 **Proof.** Let us start by exposing the rough idea of the trick using set-theoretic maps. We
 310 reuse the notations of Definition 4.5 and assume w.l.o.g. that the set of states is $Q = \{1, 2\}$.
 311 Suppose that at some point, after processing a prefix of the input, the transducer has arrived
 312 in state 1 (resp. 2) and in the meantime has outputted $w \in \Pi^*$. We can represent this by the
 313 pair (κ_w, ζ) (resp. (ζ, κ_w)) where $\zeta, \kappa_w : \Pi^* \rightarrow \Pi^*$ are defined by $\zeta : x \mapsto \varepsilon$ and $\kappa_w : x \mapsto wx$.
 314 Some key observations are

$$315 \quad \zeta \circ \kappa_w = \zeta \quad \kappa_w \circ \kappa_{w'} = \kappa_{ww'} \quad \kappa_w(w')\zeta(w'') = \zeta(w'')\kappa_w(w') = ww''$$

316 Now, consider an input letter $c \in \Sigma$; how to encode the corresponding transition $\delta(-, c)$ as a
 317 transformation on the pair encoding the current state and output history? It depends on the
 318 state transition $\delta_1(-, c)$; we have thanks to the above identities:

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

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

$$325 \quad \lambda^\circ s. \lambda^\top f_{a_1}. \dots \lambda^\top f_{a_{|\Pi|}}. \text{out} (s \mathbf{trans}_{c_1} \dots \mathbf{trans}_{c_{|\Sigma|}})$$

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

$$327 \quad \Gamma \mid \emptyset \vdash \mathbf{trans}_c : A \multimap A \quad (\text{for all } c \in \Sigma) \quad \Gamma \mid \emptyset \vdash \text{out} : (A \multimap A) \multimap (o \multimap o)$$

328 In the presence of this non-affine context Γ , the type $S = o \multimap o$ morally serves as a purely
 329 affine type of strings, as mentioned in Remark 2.7. Moreover this “contextual encoding of
 330 strings” supports concatenation (by function composition), leading us to represent the maps
 331 ζ and κ_w as open terms of type $T = S \multimap S$ that use non-affinely the variables f_a for $a \in \Pi$.

332 We shall take the type A , at which the input \mathbf{Str}_Σ is instantiated, to be $A = T \multimap T \multimap S$,
 333 which is indeed purely affine as required by the theorem statement. This can be seen morally
 334 as a type of continuations taking pairs of type $T \otimes T$ (although our $\lambda\wp$ -calculus has no actual
 335 \otimes connective). Without further ado, let us program:

336 ■ $\text{concat} : \lambda^\circ w. \lambda^\circ w'. \lambda^\circ x. w (w' x) : S \multimap S \multimap o \multimap o = S \multimap S \multimap S = S \multimap T$ plays the
337 roles of both the concatenation operator and of $w \mapsto \kappa_w$ (thanks to currying)

338 ■ $\text{zeta} = \lambda^\circ w'. \lambda^\circ x. x : S \multimap o \multimap o = T$

339 ■ $u_q = \delta_2(q, c)^\dagger(\vec{f}_\Pi)$ (cf. Definition 2.4) represents the output word $\delta_2(q, c)$ that corresponds
340 to a given input letter $c \in \Sigma$ and state $q \in Q = \{1, 2\}$

341 ■ case $\delta_1(q, c) = q$: $\text{trans}_c = \lambda^\circ k. \lambda^\circ h. \lambda^\circ g. k (\lambda^\circ y. h (\text{concat } u_1 y)) (\lambda^\circ z. g (\text{concat } u_2 z))$ –
342 observe that if we wanted to handle the excluded case $\delta_1(q, c) = 3 - q$, we would write a simi-
343 lar term with the occurrences of h and g exchanged ($\lambda^\circ k. \lambda^\circ h. \lambda^\circ g. k (\lambda^\circ y. g \dots) (\lambda^\circ z. h \dots)$)
344 violating the non-commutativity requirement (contrast with the proof of Theorem 5.4);

345 ■ case $\delta_1(q, c) = 1$: $\text{trans}_c = \lambda^\circ k. \lambda^\circ h. \lambda^\circ g. k (\text{concat } (\text{concat } (h u_1) (g u_2))) \text{zeta}$

346 ■ case $\delta_1(q, c) = 2$: $\text{trans}_c = \lambda^\circ k. \lambda^\circ h. \lambda^\circ g. k \text{zeta} (\text{concat } (\text{concat } (h u_1) (g u_2)))$

347 ■ $\text{out} = \lambda^\circ j. j (\lambda^\circ h. \lambda^\circ g. \text{concat } (h v_1) (g v_2)) (\lambda^\circ x. x) \text{zeta}$, where $v_q = F(q)^\dagger(\vec{f}_\Pi)$ represents
348 the output suffix for state $q \in \{1, 2\}$, assuming w.l.o.g. that the initial state is 1 (also,
349 here $\lambda^\circ x. x$ represents κ_ε since the latter is the identity on Π^*)

350 We leave it to the reader to check that these terms are well-typed – in particular that
351 they enjoy the requisite non-commutativity conditions – and that they have the expected
352 computational behavior. Note in particular that in functional programming terms, the use
353 of continuations turns the “right fold” of the Church-encoded input string into a “left fold”,
354 and the latter fits with the left-to-right processing of a sequential transducer. ◀

355 5 Regular languages in extensions of the λ_{\wp} -calculus

356 5.1 The commutative case

357 The λ_{\wp} -calculus adds two restrictions to the simply typed λ -calculus, namely linearity
358 (strictly speaking, affine typing) and non-commutativity, with the latter depending on the
359 former as already mentioned. One could wonder whether linearity by itself would be enough
360 to characterize star-free languages. We now show that it is not the case.

361 The *commutative* variant of the λ_{\wp} -calculus – let us call this variant the *$\lambda\ell$ -calculus* (by
362 analogy with Clairambault et al.’s $\lambda\ell\mathbf{Y}$ -calculus [11]) – has the same grammar of types and
363 terms as the λ_{\wp} -calculus (cf. §2). The typing rules are also given by Figure 1, but their
364 interpretation differs from the previous one as follows: Δ, Δ' stand for *sets* of bindings $x : A$,
365 $\Delta \cdot \Delta'$ denotes the *disjoint union* of sets, and one must read “subset” instead of “subsequence”.
366 In other words, the main difference is that in the $\lambda\ell$ -calculus, the linear context Δ does not
367 keep track of the *ordering* of variables.

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

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

372 **Proof.** Soundness is a consequence of Hillebrand and Kanellakis’s Theorem 1.1, by a simple
373 translation from the $\lambda\ell$ -calculus to the simply typed λ -calculus which “forgets linearity”.

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

$$378 \quad v_{mp}[i] = v_m[j_1] \vee \dots \vee v_m[j_k] \text{ where } \{j_1, \dots, j_k\} = \{j \in M \mid jp = i\}$$

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

381 Let $\mathbf{ttt} = \lambda^o x. \mathbf{true} : \mathbf{Bool} \multimap \mathbf{Bool}$ and $\mathbf{fff} = \lambda^o x. x : \mathbf{Bool} \multimap \mathbf{Bool}$. This makes the
 382 type $B = \mathbf{Bool} \multimap \mathbf{Bool}$ into a kind of type of booleans that supports a disjunction of type
 383 $B \multimap B \multimap B$ (by function composition) and a type-cast function of type $B \multimap \mathbf{Bool}$ (by
 384 applying to \mathbf{false}). (Of course B has other closed inhabitants besides \mathbf{ttt} and \mathbf{fff} , but we
 385 only use those two.) Using this type and the “iteration+continuations” recipe of the proof of
 386 Lemma 4.10, one can define a $\lambda\ell$ -term of type $\mathbf{Str}_\Sigma[A] \multimap \mathbf{Bool}$ that decides the language L
 387 with $A = B \multimap \dots \multimap B \multimap \mathbf{Bool}$ (with $|M|$ arguments of type B). ◀

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

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

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

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

401 **Proof sketch.** First, by Theorem 4.1, we can already encode aperiodic sequential functions,
 402 since every well-typed $\lambda\wp$ -term is also a well-typed $\lambda\ell$ -term. One can also show that
 403 Lemma 4.4 applies to the $\lambda\ell$ -calculus. By the general Krohn–Rhodes theorem, we just need
 404 to handle the case of a sequential transducer whose transition monoid is a group.

405 The idea, in terms of set-theoretic maps as in our explanation of the proof of Lemma 4.10
 406 (whose notations we borrow here), is as follows. The current state $q \in Q$ and output history
 407 $w \in \Pi^*$ is represented by a Q -indexed family $(g_{q'})_{q' \in Q}$ of functions such that $g_q = \kappa_w$ and
 408 for $q' \neq q$, $g_{q'} = \zeta$. The transition $\delta(-, c)$ is represented by $(g_q)_{q \in Q} \mapsto (g_{\sigma(q)} \circ \kappa_{\delta_2(\sigma(q), c)})_{q \in Q}$
 409 where $\sigma = (\delta_1(-, c))^{-1}$ – the latter is well-defined because the group assumption means that
 410 $\delta_1(-, c)$ is a permutation of Q . The final output is obtained at the end as the concatenation
 411 $g_{q_1}(F(q_1)) \dots g_{q_n}(F(q_n))$ where $Q = \{q_1, \dots, q_n\}$ (with an arbitrary enumeration of Q).

412 The elaboration of the corresponding $\lambda\ell$ -term is left to the reader. Keep in mind that
 413 the reason this term will not be well-typed for the $\lambda\wp$ -calculus is that the inversions in the
 414 permutation $\delta_1(-, c)$ correspond to violations of non-commutative typing. ◀

415 5.2 Extension with additive pairs

416 Let’s look at what happens if we add the *additive conjunction* connective of linear logic to the
 417 $\lambda\wp$ -calculus. The $\lambda\wp^{\&}$ -calculus is obtained by adding $A, B ::= \dots \mid A \& B$ to the grammar of
 418 types and $t, u ::= \dots \mid \langle t, u \rangle \mid \pi_1 t \mid \pi_2 t$ for terms, with the typing rules

$$419 \frac{\Gamma \mid \Delta \vdash t : A \quad \Gamma \mid \Delta \vdash u : B}{\Gamma \mid \Delta \vdash \langle t, u \rangle : A \& B} \quad \frac{\Gamma \mid \Delta \vdash t : A_1 \& A_2}{\Gamma \mid \Delta \vdash \pi_i t : A_i} \quad (\text{see [35, §4]})$$

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

421 Recall that we discussed in the introduction the need to prevent the existence of a λ_{\wp} -term
 422 of type $\text{Bool} \multimap \text{Bool}$ for negation⁸. However, if we use the additive conjunction to define
 423 the type $\text{Bool}^{\&} = (o \& o) \multimap o$, the following are well-typed $\lambda_{\wp^{\&}}$ -terms:

$$424 \quad \text{true}^{\&} = \lambda^{\circ} p. \pi_1 p \quad \text{false}^{\&} = \lambda^{\circ} p. \pi_2 p \quad \text{not}^{\&} = \lambda^{\circ} b. \lambda^{\circ} p. b \langle \pi_2 p, \pi_1 p \rangle$$

425 This admits a straightforward generalization:

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

430 ► **Corollary 5.6.** *Every regular language can be defined by a closed $\lambda_{\wp^{\&}}$ -term of type*
 431 *$\text{Str}_{\Sigma}[A] \multimap \text{Bool}$ for some purely affine type A – we consider ‘&’ as a linear connective and*
 432 *therefore allow it in A .*

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

436 Similarly, one could show that the addition of the *additive disjunction* ‘ \oplus ’ of linear logic
 437 to the λ_{\wp} -calculus would be sufficient to encode all regular languages.

438 **5.3 On regular and first-order tree languages: a discussion**

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

443 We shall not go into the details of tree automata there, but the knowledgeable reader
 444 may check that Proposition 5.5 can be used to encode all *regular tree languages* over
 445 $(a : 2, b : 2, x : 0)$ as closed $\lambda_{\wp^{\&}}$ -terms of type $\text{Tree}_{(2,2,0)}[A] \multimap \text{Bool}$ for purely affine A .
 446 Predictably, this fails for the λ_{\wp} -calculus without additive connectives. More noteworthy
 447 is the failure of the trick used to prove Theorem 5.1 for the commutative λ_{ℓ} -calculus when
 448 one replaces strings with trees. Thus, it seems (though this remains conjectural) that *the*
 449 *additives of linear logic might be required to express some regular tree languages.*

450 We believe that this is no accident and that some fundamental difficulty of automata
 451 theory is being manifested here. Indeed, if we had a characterization of regular tree languages
 452 in the λ_{ℓ} -calculus, we could expect that moving to the λ_{\wp} -calculus would yield the *first-order*
 453 *tree languages*, which are the commonly accepted counterpart of star-free languages for
 454 trees. (Recall from Theorem 1.5 that definability in first-order logic is among the equivalent
 455 definitions of star-free languages.) However, while Theorem 1.5 demonstrates that star-free
 456 languages are well-understood, the situation is quite different for first-order tree languages:
 457 there is no known algebraic characterization, and neither is there any known algorithm to
 458 decide whether a tree automaton recognizes a first-order language (see e.g. [7]).

⁸ Actually, the λ_{\wp} -calculus does admit a term for negation: $\lambda^{\circ} b. b \text{ false true} : \text{Bool}[\text{Bool}] \multimap \text{Bool}$. The heterogeneity of the input and output types means that this λ_{\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 $\mathbf{Str}_\Sigma[A] \rightarrow \mathbf{Str}_\Pi$. Indeed, an immediate consequence of this “regular = λ -definable” result is:

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

Proof idea. Let $u : \mathbf{Str}_\Pi[B] \rightarrow \mathbf{Bool}$ and $t : \mathbf{Str}_\Sigma[A] \rightarrow \mathbf{Str}_\Pi$ 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 $\mathbf{Str}_\Sigma[A[B]] \rightarrow \mathbf{Bool}$ (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 $\mathbf{Str}_\Sigma[A] \multimap \mathbf{Str}_\Pi$, 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	$\mathbf{Str}_\Sigma[A] \multimap \mathbf{Bool}$	$\mathbf{Str}_\Sigma[A] \multimap \mathbf{Str}_\Pi$
$\lambda\wp$	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.)

494 — References

- 495 1 Samson Abramsky. Temperley–Lieb Algebra: From Knot Theory to Logic and Computation
496 via Quantum Mechanics. In Goong Chen, Louis Kauffman, and Samuel Lomonaco, editors,
497 *Mathematics of Quantum Computation and Quantum Technology*, volume 20074453, pages
498 515–558. Chapman and Hall/CRC, September 2007. doi:10.1201/9781584889007.ch15.
- 499 2 Klaus Aehlig. A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of
500 Automata. *Logical Methods in Computer Science*, 3(3), July 2007. doi:10.2168/LMCS-3(3:
501 1)2007.
- 502 3 Rajeev Alur and Loris D’Antoni. Streaming Tree Transducers. *Journal of the ACM*, 64(5):1–55,
503 August 2017. doi:10.1145/3092842.
- 504 4 Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In *IARCS*
505 *Annual Conference on Foundations of Software Technology and Theoretical Computer Science*
506 (*FSTTCS 2010*), pages 1–12, 2010. doi:10.4230/LIPIcs.FSTTCS.2010.1.
- 507 5 Jean-Marc Andreoli, Gabriele Pulcini, and Paul Ruet. Permutative Logic. In *Computer Science*
508 *Logic*, Lecture Notes in Computer Science, pages 184–199. Springer, Berlin, Heidelberg, August
509 2005. doi:10.1007/11538363_14.
- 510 6 Andrew Barber. Dual Intuitionistic Linear Logic. Technical report ECS-LFCS-96-347,
511 LFCS, University of Edinburgh, 1996. URL: [http://www.lfcs.inf.ed.ac.uk/reports/96/
512 ECS-LFCS-96-347/](http://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/).
- 513 7 Mikołaj Bojańczyk. Algebra for trees. To appear in the Handbook of Automata Theory. URL:
514 <https://www.mimuw.edu.pl/~bojan/papers/treealgs.pdf>.
- 515 8 Mikołaj Bojańczyk. The simplest transducer models and their Krohn-Rhodes decom-
516 positions. [https://www.mimuw.edu.pl/~bojan/slides/transducer-course/krohn-rhodes.
517 html](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.
- 518 9 Mikołaj Bojańczyk. Polyregular Functions. *CoRR*, abs/1810.08760, October 2018. arXiv:
519 1810.08760.
- 520 10 Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ -programs on
521 term algebras. *Theoretical Computer Science*, 39:135–154, January 1985. doi:10.1016/
522 0304-3975(85)90135-5.
- 523 11 Pierre Clairambault, Charles Grellois, and Andrzej Murawski. Linearity in Higher-order
524 Recursion Schemes. *Proceedings of the ACM on Programming Languages*, 2(POPL):39:1–39:29,
525 December 2017. doi:10.1145/3158127.
- 526 12 Volker Diekert, Manfred Kufleitner, and Benjamin Steinberg. The Krohn-Rhodes Theorem and
527 Local Divisors. *Fundamenta Informaticae*, 116(1-4):65–77, 2012. doi:10.3233/FI-2012-669.
- 528 13 Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and
529 two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254,
530 April 2001. doi:10.1145/371316.371512.
- 531 14 Emmanuel Filiot and Pierre-Alain Reynier. Transducers, Logic and Algebra for Functions
532 of Finite Words. *ACM SIGLOG News*, 3(3):4–19, August 2016. URL: [https://dl.acm.org/
533 doi/10.1145/2984450.2984453](https://dl.acm.org/doi/10.1145/2984450.2984453), doi:10.1145/2984450.2984453.
- 534 15 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987.
535 doi:10.1016/0304-3975(87)90045-4.
- 536 16 Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors,
537 *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages
538 69–108. American Mathematical Society, Providence, RI, 1989. Proceedings of a Summer
539 Research Conference held June 14–20, 1987. doi:10.1090/conm/092/1003197.
- 540 17 Jean-Yves Girard. Light Linear Logic. *Information and Computation*, 143(2):175–204, June
541 1998. doi:10.1006/inco.1998.2700.
- 542 18 Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular
543 approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, April
544 1992. doi:10.1016/0304-3975(92)90386-T.

- 545 19 Charles Grellois. *Semantics of linear logic and higher-order model-checking*. PhD thesis,
546 Université Paris 7, April 2016. URL: <https://tel.archives-ouvertes.fr/tel-01311150/>.
- 547 20 Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational*
548 *Logic*, 8(1), January 2007. doi:10.1145/1182613.1182614.
- 549 21 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible
550 Pushdown Automata and Recursion Schemes. *ACM Transactions on Computational Logic*,
551 18(3):25:1–25:42, August 2017. doi:10.1145/3091122.
- 552 22 Gerd G. Hillebrand and Paris C. Kanellakis. On the Expressive Power of Simply Typed and
553 Let-Polymorphic Lambda Calculi. In *Proceedings of the 11th Annual IEEE Symposium on*
554 *Logic in Computer Science*, pages 253–263. IEEE Computer Society, 1996. doi:10.1109/LICS.
555 1996.561337.
- 556 23 Naoki Kobayashi. Model Checking Higher-Order Programs. *Journal of the ACM*, 60(3):1–62,
557 June 2013. doi:10.1145/2487241.2487246.
- 558 24 Kenneth Krohn and John Rhodes. Algebraic theory of machines. I. Prime decomposition
559 theorem for finite semigroups and machines. *Transactions of the American Mathematical*
560 *Society*, 116:450–464, 1965. doi:10.1090/S0002-9947-1965-0188316-1.
- 561 25 Denis Kuperberg, Laureline Pinault, and Damien Pous. Cyclic Proofs and Jumping Automata.
562 In Arkadev Chattopadhyay and Paul Gastin, editors, *39th IARCS Annual Conference on*
563 *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019)*,
564 volume 150 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:14,
565 Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/
566 LIPIcs.FSTTCS.2019.45.
- 567 26 Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*,
568 65(3):154–170, 1958.
- 569 27 Olivier Laurent. Polynomial time in untyped elementary linear logic. *Theoretical Computer*
570 *Science*, October 2019. doi:10.1016/j.tcs.2019.10.002.
- 571 28 Daniel Leivant. Reasoning about functional programs and complexity classes associated with
572 type disciplines. In *24th Annual Symposium on Foundations of Computer Science (FOCS*
573 *1983)*, pages 460–469, Tucson, AZ, USA, November 1983. doi:10.1109/SFCS.1983.50.
- 574 29 Damiano Mazza. *Polyadic Approximations in Logic and Computation*. Habilitation à diriger
575 des recherches, Université Paris 13, November 2017. URL: [https://lipn.fr/~mazza/papers/
576 Habilitation.pdf](https://lipn.fr/~mazza/papers/Habilitation.pdf).
- 577 30 Paul-André Melliès. Higher-order parity automata. In *2017 32nd Annual ACM/IEEE*
578 *Symposium on Logic in Computer Science (LICS)*, pages 1–12, Reykjavik, Iceland, June 2017.
579 IEEE. doi:10.1109/LICS.2017.8005077.
- 580 31 Paul-André Melliès. Ribbon Tensorial Logic. In *Proceedings of the 33rd Annual ACM/IEEE*
581 *Symposium on Logic in Computer Science - LICS '18*, pages 689–698, Oxford, United Kingdom,
582 2018. ACM Press. doi:10.1145/3209108.3209129.
- 583 32 Anca Muscholl and Gabriele Puppis. The Many Facets of String Transducers. In Rolf
584 Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical*
585 *Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings*
586 *in Informatics (LIPIcs)*, pages 2:1–2:21, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-
587 Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2019.2.
- 588 33 Lê Thành Dũng Nguyễn. Typed lambda-calculi and superclasses of regular functions. *CoRR*,
589 abs/1907.00467, 2019. arXiv:1907.00467.
- 590 34 C.-H. Luke Ong. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In
591 *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90, Seattle,
592 WA, USA, 2006. IEEE. doi:10.1109/LICS.2006.38.
- 593 35 Jeff Polakow and Frank Pfenning. Natural Deduction for Intuitionistic Non-commutative
594 Linear Logic. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Jean-Yves Girard,
595 editors, *Typed Lambda Calculi and Applications*, volume 1581, pages 295–309. Springer Berlin
596 Heidelberg, Berlin, Heidelberg, 1999. doi:10.1007/3-540-48959-2_21.

- 597 **36** Jeff Polakow and Frank Pfenning. Relating Natural Deduction and Sequent Calculus for
598 Intuitionistic Non-Commutative Linear Logic. *Electronic Notes in Theoretical Computer
599 Science*, 20:449–466, January 1999. doi:10.1016/S1571-0661(04)80088-4.
- 600 **37** Christian Retoré. Pomset logic: A non-commutative extension of classical linear logic. In
601 Philippe de Groote, editor, *Typed Lambda Calculi and Applications, Third International
602 Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4,
603 1997, Proceedings*, volume 1210 of *Lecture Notes in Computer Science*, pages 300–318. Springer,
604 1997. doi:10.1007/3-540-62688-3_43.
- 605 **38** Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.
606 Translated by Reuben Thomas. doi:10.1017/CB09781139195218.
- 607 **39** Sylvain Salvati. Recognizability in the Simply Typed Lambda-Calculus. In Hiroakira Ono,
608 Makoto Kanazawa, and Ruy de Queiroz, editors, *Logic, Language, Information and Com-
609 putation*, volume 5514, pages 48–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
610 doi:10.1007/978-3-642-02261-6_5.
- 611 **40** Thomas Seiller. Interaction Graphs: Non-Deterministic Automata. *ACM Transactions on
612 Computational Logic*, 19(3):21:1–21:24, August 2018. doi:10.1145/3226594.
- 613 **41** Howard Straubing. First-order logic and aperiodic languages: a revisionist history. *ACM
614 SIGLOG News*, 5(3):4–20, 2018. URL: <https://dl.acm.org/doi/10.1145/3242953.3242956>,
615 doi:3242953.3242956.
- 616 **42** Kazushige Terui. Semantic Evaluation, Intersection Types and Complexity of Simply Typed
617 Lambda Calculus. In *23rd International Conference on Rewriting Techniques and Applications
618 (RTA'12)*, pages 323–338, 2012. doi:10.4230/LIPIcs.RTA.2012.323.
- 619 **43** Igor Walukiewicz. LambdaY-calculus with priorities. In *34th Annual ACM/IEEE Symposium
620 on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages
621 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785674.
- 622 **44** Noam Zeilberger and Alain Giorgetti. A correspondence between rooted planar maps and
623 normal planar lambda terms. *Logical Methods in Computer Science*, 11(3), September 2015.
624 doi:10.2168/LMCS-11(3:22)2015.

625 **A** Reminder: the Krohn–Rhodes decomposition theorems, from 626 transformation monoids to sequential transducers

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

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

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

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

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

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

645 **Proof sketch.** We give a direct description of $(X, M) \wr (Y, N) \wr (Z, P) = (X \times Y \times Z, W)$:
 646 ■ the underlying set of the monoid W is $M^{Y \times Z} \times N^Z \times P$ – note that it is canonically
 647 isomorphic to $(M^Y \times N)^Z \times P$;
 648 ■ the right action is $(x, y, z) \cdot (f, g, p) = (x \cdot f(y, z), y \cdot g(z), z \cdot p)$;
 649 ■ the multiplication on W is
 650 $(f, g, p)(f', g', p') = ((y, z) \mapsto f(y, z)f'(y \cdot n, z \cdot p)), (z \mapsto g(z)g'(z \cdot p)), pp'$. ◀

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

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

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

657 **Proof.** Let M be a finite monoid. Suppose that for $x \in M$, there is no $n \in \mathbb{N}$ such that
 658 $x^n = x^{n+1}$; then by finiteness, $(x^i)_{i \in \mathbb{N}}$ must be ultimately periodic with period $k \geq 2$, and
 659 one can define a surjective morphism from the submonoid generated by x to the cyclic group
 660 of order k by sending x to the latter's generator. The converse follows a similar reasoning
 661 (recall that every non-trivial group contains a non-trivial cyclic subgroup). ◀

662 ► **Theorem A.6** (Krohn–Rhodes with strong divisors [12, Theorem 4.1]). *Every finite transfor-*
 663 *mation monoid (X, M) strongly divides some wreath product $(Y_1, N_1) \wr \dots \wr (Y_n, N_n)$ where*
 664 *each (Y_k, N_k) is either:*

- 665 ■ *the flip-flop $(Y_k, N_k) = (\{1, 2\}, \{\text{id}_{\{1, 2\}}, (x \mapsto 1), (x \mapsto 2)\})$ (with the action $x \cdot f = f(x)$*
 666 *and the monoid multiplication $fg = g \circ f$);*
- 667 ■ *a finite group dividing M acting on itself by right multiplication.*

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

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

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

677 Now let us relate this wreath product operation to sequential functions. This is sufficient
 678 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 transducer with transition monoid T*
describing a function $f : \Sigma^ \rightarrow \Pi^*$. Suppose that (Q, T) strongly divides some faithful*
transformation monoid $(X, M) \wr (Y, N)$. Then there is an alphabet Ξ and transducers

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

679 *with respective transition monoid T_X and T_Y corresponding to sequential functions $f_X : \Sigma^* \rightarrow$*
 680 *Ξ^* and $f_Y : \Xi^* \rightarrow \Pi^*$ such that $f = f_X \circ f_Y$. Moreover, there are injective homomorphisms*
 681 *$T_X \hookrightarrow M$ 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 $\varphi : K \rightarrow T$, $s : X \times Y \rightarrow Q$ 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

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

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

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

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

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

691 ■ For $n = 1$, let $\varphi : K \rightarrow T$ and $s : X \rightarrow Q$ be the maps witnessing that (Q, T)
 692 strongly divides (X, M) . Let x_I be such that $s(x_I) = q_I$, and, for each $a \in \Sigma$, pick
 693 an element $m_a \in K$ such that $\varphi(m_a) = \delta_1(-, a)$. Then, letting (Q, δ, q_I, F) being
 694 the transducer under scrutiny, a suitable transducer (X, δ', x_I, F') is defined by setting
 695 $\delta'(x, a) = (x \cdot m_a, \delta_2(s(x), a))$ and $F'(x) = F(s(x))$.

696 ■ For $n > 1$, use Proposition A.9 and the induction hypothesis.

697 ◀

698 **Proof of Theorems 4.8 and 5.2.** Let (Q, δ, q_I, F) be a transducer computing a certain
 699 sequential function $f : \Sigma^* \rightarrow \Pi^*$ and let T be its transition monoid. By Theorem A.6,
 700 there is a transformation monoid (Y, N) which can be written as a wreath product $(Y, N) =$
 701 $(Y_1, N_1) \wr \dots \wr (Y_k, N_k)$ such that (Q, T) strongly divides (Y, N) , and the (Y_i, N_i) are either flip-
 702 flops or groups (the latter case being ruled out for Theorem 4.8, thanks to Proposition A.5). By
 703 applying Proposition A.10, we may obtain transducers \mathcal{T}_i implementing sequential functions
 704 $f_i : \Xi_i^* \rightarrow \Xi_{i+1}^*$ such that $\Xi_0 = \Sigma$, $\Xi_k = \Pi$ and $f = f_{k-1} \circ \dots \circ f_0$. Furthermore, we know that
 705 the state space of \mathcal{T}_i is Y_i and that the corresponding transition monoid T_i embeds into N_i .
 706 Recalling that “being aperiodic” and “being a finite subgroup” are properties stable under
 707 homomorphic embeddings, we know that either Y_i has cardinality 2 and T_i is aperiodic with
 708 two states or T_i is a group (a trivial group if T was aperiodic), thus we may conclude. ◀

709 B Omitted proofs

710 B.1 Proof of Lemma 3.1

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

$$713 \quad t =_{\beta\eta} \lambda^{\circ\beta\eta} \lambda^{\circ} s. \lambda^{\circ} x. \lambda^{\circ} y. z t_1 \dots t_n \quad \text{where } \emptyset \mid s : \mathbf{Str}_\Sigma[A], x : o, y : o \vdash z t_1 \dots t_n : o$$

714 If $z \in \{x, y\}$, then $n = 0$ and t is a constant function from strings to booleans: the statement
 715 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

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

718 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
 719 $i \in \{1, \dots, |\Sigma|\}$, $g_{c_i} = t_i$.

720 B.2 Proof of Lemma 4.2

721 Since L is star-free, $L = \varphi^{-1}(P)$ for some $\varphi \in \text{Hom}(\Sigma^*, M)$ and $P \subseteq M$, where M is
 722 an aperiodic monoid. Here is a sequential transducer computing $\chi_L: Q = M$, $\delta(m, c) =$
 723 $(m\varphi(c), \varepsilon)$ and $F(m) = 1$ if $m \in P$ and $F(m) = \varepsilon$ otherwise. Its transition monoid is
 724 $\varphi(\Sigma^*) \subseteq M$, which is aperiodic.

725 B.3 Proof of Lemma 4.3

726 The λ^φ -term in question is $\lambda^\circ s. s (\lambda^\circ x. \text{true}) \text{false}$.

727 B.4 Proof of Lemma 4.4

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

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

731 **Proof.** Routine induction on the typing derivation. ◀

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

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