Université Paris XIII (dite Sorbonne Paris Nord)
École Doctorale Sciences, Technologies, Santé Galilée

Implicit automata in linear logic and categorical transducer theory

Automates implicites en logique linéaire et théorie catégorique des transducteurs

Thèse de doctorat *présentée par*

Lê Thành Dũng (Tito) Nguyễn
Laboratoire d'Informatique de Paris Nord

pour l'obtention du grade de Docteur en Informatique

soutenue le 3 décembre 2021 *devant le jury composé de* :

Patrick Baillot	Directeur de recherche CNRS Université de Lille	Examinateur
Stefano Guerrini	Professeur Université Paris XIII	Directeur de thèse
Anca Muscholl	Professeure Université de Bordeaux	Examinatrice
Daniela Petrişan	Maîtresse de conférences Université de Paris	Examinatrice
Damien Pous	Directeur de recherche CNRS École normale supérieure de Lyon	Rapporteur
Thomas Seiller	Chargé de recherche CNRS Université Paris XIII	Co-encadrant
Igor Walukiewicz	Directeur de recherche CNRS Université de Bordeaux	Rapporteur

Abstract. This thesis aims to establish new connections between two distinct fields within theoretical computer science. The first is automata theory, whose objects of study are models of computation with limited memory. The second is the theory of programming languages, in particular of the λ-calculus upon which the functional paradigm is based.

It is known that the functions from strings to booleans definable by programs written in the simply typed λ-calculus, using Church encodings, correspond exactly to regular languages. Taking this as our starting point, we give several other characterizations of automata-theoretic classes of languages and string-to-string (or even tree-to-tree) functions using λ-calculi with linear types. To prove those results, we develop some connections between denotational semantics – which we use to evaluate λ-terms – and categorical automata theory; in particular, we show that monoidal closed categories, which appear in the semantics of linear logic, also provide a basis to generalize some constructions on automata that morally involve function spaces. Our investigations into linear λ-calculi also lead us to introduce and study a new transducer model that computes string-to-string functions with polynomial growth.

Résumé. Cette thèse cherche à établir de nouveaux liens entre deux domaines distincts au sein de l'informatique théorique : d'une part la théorie des automates, dont les objets d'étude sont des modèles de calcul à mémoire limitée, et d'autre part celle des langages de programmation, en particulier du λ-calcul qui sert de base au paradigme fonctionnel.

Il est connu que les fonctions des mots vers les booléens définissables par des programmes écrits en λ-calcul simplement typé, en utilisant les codages de Church, correspondent exactement aux langages rationnels. Partant de là, nous employons des λ-calculs à types linéaires pour caractériser plusieurs autres classes de langages et de fonctions sur les mots (voire sur les arbres) provenant de la théorie des automates. Afin de démontrer ces résultats, nous tissons des liens entre les sémantiques dénotationnelles – qui nous servent à évaluer les λ-termes – et la théorie catégorique des automates ; en particulier, nous montrons que les catégories monoïdales closes, qui apparaissent en sémantique de la logique linéaire, fournissent également une base pour généraliser quelques constructions sur les automates qui font moralement intervenir des espaces de fonctions. Nos recherches sur les λ-calculs linéaires nous mènent également à introduire et étudier un nouveau modèle de transducteurs qui calcule des fonctions sur les mots à croissance polynomiale.

Un jour vous verrez la serveuse *automate*
S'en aller cultiver ses tomates
Au soleil…

Starmania on semigrape varieties

When Randolph Carter was thirty he lost the key of the gate of dreams. Prior to that time he had made up for the prosiness of life by nightly excursions to strange and ancient cities beyond space, and lovely, unbelievable garden lands across ethereal seas […]
He had read much of things as they are, and talked with too many people. Well-meaning philosophers had taught him to look into the *logical relations* of things, and analyse the processes which shaped his thoughts and fancies.

H. P. Lovecraft, *The Silver Key*

Il en est qui, face à cela, se contentent de hausser les épaules d'un air désabusé et de parier qu'il n'y a rien à tirer de tout cela, sauf des rêves. Ils oublient, ou ignorent, que notre science, et toute science, serait bien peu de chose, si depuis ses origines elle n'avait été nourrie des rêves et des visions de ceux qui s'y adonnent avec passion.

Alexander Grothendieck, *Esquisse d'un programme*

Contents

CHAPTER 1

Introduction

Theoretical computer science (TCS) was born in the 1930s when several formalisms for general-purpose computation were introduced. Arguably the two most important ones were:

- *Turing machines*, idealized mechanical devices arising from Alan Turing's conceptual analysis of how human beings carry out concrete pen-and-paper calculations,[1]
- and Alonzo Church's λ-*calculus*, a somewhat more abstract formal system which is recognized today as the first *(high-level) programming language*, that is, a means of specifying algorithms while abstracting away from the details of the hardware meant to execute them.

It turns out that a function (i.e. a deterministic input-output relation) can be computed by a Turing machine if, and only if, it can be defined in the λ-calculus; and other models of computation were also shown to be equivalent in this sense. This led to a rigorous way to capture the intuitive concept of "which tasks can be solved by computers": a function is *computable*[2] when it can be defined in any of those equivalent models. That two starkly different approaches – through "machines" and through "programs" – converge to the same definition of computability is a testament to the robustness of this definition.

But this also planted the seeds of the diversity of contemporary TCS. This manuscript deals with two subfields that are on different sides of the machine/program dichotomy:

- *Automata theory* is concerned with a wide variety of machine models which often be seen as restricted variants of Turing machines, with far less computational power. The most famous example, covered in most undergraduate computer science curricula, is *finite automata* (also known as *finite state machines*) which can find patterns in text using a low amount of memory (they are used for instance to implement the Unix `grep` command).
- *Programming language theory* is an area aptly described by its name. Among its objects of study are certain languages similar to the λ-calculus, called *typed* λ-*calculi*. They are statically typed functional languages, like the "real world" languages Haskell and OCaml.

The work that we present here contains a modest contribution to "pure" automata theory, but it is mostly about drawing connections between those two areas. At an impressionistic level, the main ideas that we defend – or, to be pedantic, the *thesis* of the *dissertation* that you are reading – are the following:

- Automata are relevant to natural questions on the expressive power of typed λ-calculi; we demonstrate this by proving equivalences in the same sense as above.
- Conversely, the tools of programming language theory (in particular categorical semantics) can provide conceptual insights on automata.

[1] Turing's seminal paper [Tur37] uses the word "computer" to refer to a person performing calculations, as was the historical usage. Indeed, computer science precedes the construction of computers in the modern sense (Konrad Zuse's Z3 was finished in 1941), although there are some precursors, for instance the 18th century Jacquard looms that could be "programmed" using punch cards. The first design for a general-purpose computer is Charles Babbage's Analytical Engine from the 19th century – for which Ada Lovelace wrote the first computer program in history – but it was not built at the time.

[2] Or historically, *recursive*, although this can be a bit misleading with respect to the current usage in programming practice. Likewise, computability theory is also called recursion theory.

Before we can explain the precise contributions behind those slogans, we must first recall some more background in order to explain what distinguishes our work from related topics such as *higher-order model checking* or *implicit computational complexity*.

1.1. Background

1.1.1. On the landscape of TCS today: algorithms/complexity vs "logic in CS". The two themes that we have mentioned until now are both part of what is sometimes called "Volume B" of theoretical computer science, or "European" TCS. The other half of the subject, "Volume A", is concerned essentially with *algorithmics* and *computational complexity* – and it is often called simply "theory" in North American computer science departments (despite the fact that a lot of interesting work on Volume B subjects is done in the programming language research groups of some of those departments). To give an illustration of this cultural divide, consider for instance Boaz Barak's *Introduction to Theoretical Computer Science* (https://introtcs.org): at the time of writing, out of 23 chapters, a single one is dedicated to finite automata, and the λ-calculus only takes up two sections in the chapter on computability, while most other chapters are focused on complexity. For further historical discussion and an explanation of the "Volume A/B" terminology, we refer to Moshe Vardi's column in *Communications of the ACM* [Var15] and to the interesting comments at https://archive.md/v6iYu that it elicited in response.

Complexity theory focuses mainly on the "resources" necessary to carry out computations. Two central examples are *time complexity* and *space complexity*: how long does it take for an algorithm to execute, and how much memory does it need to consume during that execution, depending on the size of its input? Typically, the Millennium Prize problem "P vs NP" (see [Aar17] for a comprehensive survey) is about time complexity: roughly speaking, a function is in P if there exists an algorithm that solves it "reasonably quickly" (to be precise, the running time must be bounded by some polynomial function of the input size).

To define formally what this means, one must fix a model of computation with a measure of execution time. There is an *invariance thesis* due to Slot and van Emde Boas [SE88] which claims that all "reasonable" models of computation can simulate each other with a low overhead in time and space complexity (polynomial in time and constant in space). As a consequence, the aforementioned class P of tractable problems should not depend on the considered model, which is an argument in favor of its canonicity, similarly to the multiple equivalent definitions of computability. But those "reasonable" models tend to be ultimately *machines* (in fact, theoreticians customarily use Turing machines for this purpose): it is difficult to define the running time of a program in a high-level language without referring to an (idealized) machine that executes it.[3]

So, in our earlier machines vs programs dichotomy, complexity theory and automata theory are arguably on the same side, with the difference being the kind of restrictions

[3] In the λ-calculus, there is a seemingly natural intrinsic measure of time complexity, namely the number of β-reduction steps. But there are two issues: first, this number depends on the reduction strategy chosen, and second, it is far from clear whether a each β-reduction step can be simulated efficiently, since a naive implementation would need to duplicate arbitrarily large data for a single step. A surprising positive result – that a certain reduction strategy satisfies the time invariance thesis – was proved a few years ago [AL16]; we refer to that paper for further discussion of the questions related to complexity invariance for the λ-calculus.

We should also mention that complexity theory papers do not describe their algorithms directly as Turing machines, but as pseudocode that could *in principle* be complied into a machine. (In the same way most mathematics could "in principle" be translated into Bourbaki's foundational system, though this would be completely intractable in practice [Mat02].) By sticking to an imperative programming paradigm and a limited feature set, the complexity cost of each primitive operation in such pseudocode can be easily understood, circumventing the second problem mentioned for the λ-calculus.

considered on machine models (quantitative vs qualitative). Furthermore, questions concerning "what can be computed by given means" play an important role in both fields. For instance, P vs NP is about whether two classes of functions are equal – in other words, whether polynomial time algorithms are "powerful enough" to compute all NP functions. Most people expect the answer to be negative, so the problem is to *separate* those classes. Separation results are also fundamental for automata; the main difference is that while such results for complexity classes tend to be either trivial or utterly out of reach of current mathematics ($\mathsf{P} \neq \mathsf{NP}$ is the latter), proving non-trivial separations is something ordinary in automata theory – and we will do this in Chapter 3. And sometimes a qualitative constraint and a quantitative bound yield the same class: for instance, multi-head finite automata are equivalent to logarithmic space Turing machines (see [HKM11] for a survey).

All this to say that the Volume A/B distinction, grouping automata with programming languages rather than with complexity, is not entirely self-evident from a purely technical point of view. In order to give an impression of unity, Volume B is sometimes called "logic in computer science"; it corresponds indeed to the scope of various CS conferences[4] with "logic" in their name, and of the journal *Logical Methods in Computer Science*. However, we shall be mostly uninterested[5] in the logical aspects here, although we will explain them briefly in this introduction for the sake of broad cultural interest. Instead, we will emphasize another commonality of both main currents of Volume B which, we believe, explains to a large extent the success of the research programme presented in this dissertation:

their *algebraic* and *compositional* perspective on computation.

1.1.2. **Automata on strings and monoids.** We limit our presentation of automata theory here to the kind that lends itself to such algebraic methods. For instance, we exclude the above-mentioned multi-head automata from our scope, precisely because of their equivalence with a standard space complexity class, which signals to us that their study is better left to complexity theory.[6] The spirit of what lies *within* our scope, though, is perhaps best illustrated, at an elementary level, on the well-known *regular languages*.

What is called a *language*[7] in TCS is a set of *strings* (or *finite words*); a string is a finite sequence of letters taken in a finite *alphabet*. Given an alphabet Σ, we write Σ^* for the set of *all* strings over Σ; thus, a language is a *subset* of Σ^* (for instance, $\{aa, abb, ba\} \subseteq \{a, b\}^*$). A language $L \subseteq \Sigma^*$ can also be presented a function that maps each string in Σ^* to either "yes" or "no" (L is the set of strings mapped to "yes" by the corresponding function); such functions are called *decision problems* in computability theory and complexity theory (and *decidability* is sometimes used as a synonym for *computability* for such problems).

[4] Of course, the mere existence of those conferences confirms the sociological reality of "Volume B". But this should be nuanced by the observation that their audience generally consists of two clusters that rarely attend each other's talks.

[5] Not to say that these connections with logic are uninteresting *per se* – quite the contrary! – but they simply fall out of the scope of this dissertation. The "linear logic" part of the title has been chosen to avoid including a Greek letter: "linear λ-calculi" would have been more accurate. This is why no mention of Aristotle, Frege or the crisis of foundations and Gödel's incompleteness theorems had been made before the present sentence. (Nor did we talk about the Ariane 5 maiden flight, a common argument for the necessity of certified software and thus of logic; by the way, the final sequence in *Koyaanisqatsi* draws from a similar incident something quite different, namely a statement against industrial civilization.)

[6] Thus, we are narrowing our focus to ignore the uses of automata as a qualitative approach to usual time/space complexity classes, advocated in e.g. [Aub15].

[7] A different use of "language" that will sometimes occur in this manuscript is as an abbreviation for "programming language" when the context makes this clear. The connection is that the set of source codes without syntax errors in a given programming language is a set of strings. This is why some old textbooks on compilers dedicate several chapters to the kind of formal language theory that uses automata.

For instance, P and NP are classes of decision problems, that is, any element of P or NP is a function $\Sigma^* \to \{\text{yes}, \text{no}\}$ for some finite alphabet Σ; equivalently, they can be defined as classes of languages. Therefore, formally speaking, the class of regular languages is the same type of object as P or NP; more than that, a language is regular if and only if it is recognized[8] by some constant-space Turing machine.[9] But it is arguably not very well-behaved as a complexity class.[10] Still, the notion of regularity can be said to be canonical, just like computability and polynomial time complexity: there are many automata models that can recognize exactly the regular languages.

The algebraic point of view on these equivalent machine models starts from the following observation: many of them have a notion of "behavior of an automaton on a string", which determines whether the automaton returns "yes" when given that string as input, but contains more information. We give some examples below; prior knowledge of the keywords is not necessary to get the rough idea.

- The behavior of a *deterministic finite automaton* (DFA) on a word describes how its internal memory is updated when it reads this word from left to right. The possible values for the contents stored in memory are represented by a finite set of *states* Q, and a behavior of the automaton is described by a *function* $Q \to Q$.
- Just like DFA, *non-deterministic finite automata* (NFA) process their input from left to right, but they can "choose" between different updates, so that each initial state may lead to multiple final states. Therefore, their behavior is best represented by a *binary relation* on their finite set of states.
- *Two-way deterministic finite automata* (2DFA) – which are mostly the same thing as constant-space Turing machines – can move their reading head back and forth over the input, but they still admit a well-known notion of behavior which is a bit more complicated to describe (see e.g. [Bir89]). We will not describe it formally here, but Figure 1.1.1 should give a vague impression of the combinatorics involved.

The second crucial observation is that these behaviors enjoy a *compositionality* property. Compositionality is the idea that "the parts determine the whole"; its concrete incarnation here is that if we know the behaviors of an automaton on two strings (say, *abb* and *aa*), then we can deduce its behavior on their *concatenation* (*abbaa* in this example). For DFA and NFA, input concatenation is reflected at the level of behaviors by composition of functions and of relations, respectively; there is also an associative composition operation for 2DFA behaviors, illustrated in Figure 1.1.1. To put it succinctly: for a fixed automaton,

- the set of possible behaviors admits a *monoid* structure;
- the map from input strings in Σ^* to the corresponding behaviors is a *monoid morphism*.

[8] To recognize a language is to compute the corresponding yes/no function.

[9] Some more results along these lines: Turing machines using $o(\log \log n)$ space can only recognize regular languages, but there are non-regular languages that can be recognized with $O(\log \log n)$ space [SJL65]. Concerning time complexity, every *single-tape* Turing machine that runs in time $o(n \log n)$ recognizes a regular language [Kob85].

[10] To go below logarithmic space complexity, the usual approach is to use *circuit complexity*. From its point of view, regular languages are strange: they are not closed under uniform AC^0 (i.e. $\mathsf{ALOGTIME}$) reductions, but there exist NC^1-complete regular languages. That said, there is a substantial body of research at the intersection of algebraic automata theory and circuit complexity, see [Pin21a, Chapter 14].

There are also parts of automata theory that do not fit neatly into either algebraic language theory or complexity theory. An important example is *pushdown automata* which recognize *context-free languages*. In theory, the syntax of most real-world programming languages (cf. Footnote 7) is context-free (an (in)famous counterexample is that parsing Perl is undecidable [Keg08]), and students are taught to write compilers using parser generators based on context-free grammars. In practice… well, to quote [Bes+10]: "Parsing is considered a solved problem. Unfortunately, this view is naïve, rooted in the widely believed myth that programming languages exist."

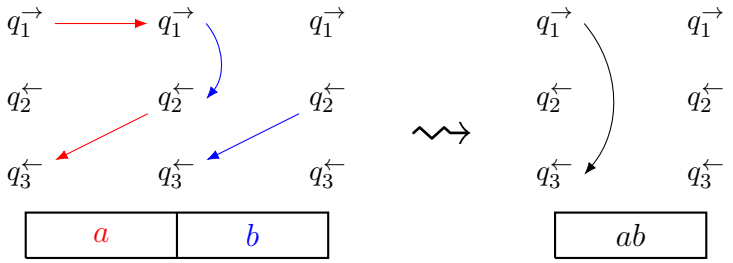


FIGURE 1.1.1. The behavior of a 2DFA with 3 states on the inputs a and b determines, by monoid multiplication, its behavior on the concatenation ab.

The third and last important point is that a fixed automaton has *finitely many* possible behaviors (despite having infinitely many potential inputs). This strong limitation finally brings us to the remarkably concise algebraic definition of regular languages:

Definition 1.1.1. A language $L \subseteq \Sigma^*$ is *regular* when there exist a *finite monoid* M, a subset $P \subseteq M$ and a morphism $\varphi : \Sigma^* \to M$ such that $L = \varphi^{-1}(P)$.

(Σ^* endowed with concatenation is the so-called *free monoid* on the set Σ, with the empty string as its identity element.) For example, the set of words in $\{a, b\}^*$ that contain an even number of as and an odd number of bs is regular: take $M = (\mathbb{Z}/(2))^2$ and $P = \{(0, 1)\}$.

Let us forget about machine models for a moment and focus on the above definition in itself. It opens up the possibility of studying correspondences between subclasses of regular languages and classes of finite monoids: what languages can one define when the possible values for the target monoid (M above) are limited? This is part of the field of *algebraic language theory*, and a lot of research has been dedicated to such correspondences[11] since the 1960s. The initial and most famous is Schützenberger's theorem, whose history is recapitulated in [Str18], and which we shall take as the definition of star-free languages.[12]

Definition 1.1.2. A monoid M is said to be *aperiodic* when for every $x \in M$ there is some $n \in \mathbb{N}$ such that $x^n = x^{n+1}$. A language $L \subseteq \Sigma^*$ is *star-free* if and only if there exist a *finite aperiodic monoid* M, a subset $P \subseteq M$ and a morphism $\varphi : \Sigma^* \to M$ such that $L = \varphi^{-1}(P)$.

As an example, the monoid $(\mathbb{Z}/(2))^2$ is *not* aperiodic because $n \not\equiv n + 1 \pmod 2$. This is consistent with (but not sufficient by itself to imply) the fact that the aforementioned language defined via parity conditions is not star-free.

In this dissertation, the only classes of languages that we consider are regular and star-free languages. But they are by no means the only sensible possibilities: for instance, there are recent results on a concrete algorithmic problem [AJP21] whose statements involve somewhat exotic classes of finite monoids, and whose proofs required new developments in algebraic language theory of potential independent interest [AP21].

1.1.3. **Transducers, trees, etc.** Coming back to machine models now, we have seen that the "monoid of behaviors" philosophy allows one to prove that various automata only recognize regular languages. This implies for instance that 2DFA are no more expressive

[11] A fundamental meta-result indicating what kind of correspondences to look for is *Eilenberg's variety theorem*. It states that *varieties* of regular languages correspond to *pseudovarieties* of finite monoids (here "(pseudo)variety" means "well-behaved class" in a precise technical sense).

[12] What Schützenberger proved is that the algebraic definition we give is equivalent to *regular expressions* without iteration star (hence "star-free") but with complementation. Regular expressions are originally a syntactic means to describe regular languages; they have been extended into a powerful tool for text processing tasks, supported in the standard libraries of many general-purpose programming languages.

than (one-way) DFA, a result that would be non-trivial without this algebraic point of view. But we will now discuss an extension that makes them no longer equivalent: instead of merely returning a yes/no answer, we allow them to produce an *output string*, so that they compute *string functions* $\Sigma^* \to \Gamma^*$ instead of languages. Such automata with output are called *transducers*. The respective transducer counterparts to the three automata models mentioned previously are:

- *deterministic finite transducers*, which compute *sequential functions*;
- *(functional) non-deterministic finite transducers*, which compute *rational functions*;
- *two-way deterministic finite transducers*, which compute *regular functions*;

and there is a strict hierarchy of function classes (surveyed in [FR16; MP19])

$$\text{sequential functions} \subsetneq \text{rational functions} \subsetneq \text{regular functions}$$

Those classical transduction classes have *linear growth*: input strings of length n are mapped to output strings of length $O(n)$. In this manuscript, we will also encounter the exponentially growing *HDT0L transductions* (see [FMS14; FR21]) and various recently investigated classes with polynomial growth [Boj18; DFG20] – including a new one that we introduce and study in detail in Chapter 3. All those transductions also collapse to regular languages if we ask them to produce a single output bit. In fact, a slightly stronger property satisfied by all functions under consideration is the following:

Definition 1.1.3. We say that $f : \Sigma^* \to \Gamma^*$ *preserves regular languages (by preimage)* when $f^{-1}(L) \subseteq \Sigma^*$ is regular for every regular $L \subseteq \Gamma^*$.

This preservation property hints that those function classes and the devices that compute them can be studied algebraically. Indeed, while the monoids of behaviors of transducers tend to be infinite (since they have to record enough information to determine an output string of unbounded length), they generally admit finite quotients on which we may apply the powerful tools of finite monoid theory[13] such as Ramsey's theorem for additive colorings[14] (used in Chapter 3) and the somewhat related[15] Factorization Forest Theorem [Sim90]. Let us also mention that a fundamental theorem called the Krohn–Rhodes decomposition [KR65] can equivalently be stated either on sequential transducers – this is the version applied in Chapter 7 – or on finite monoids.

The differences in expressive power between the transducer models listed above are orthogonal to the considerations of algebraic language theory. One can combine both: there exist suitable notions of aperiodic sequential/rational/regular functions. Another name for aperiodic regular functions is *first-order transductions*; we will explain the "first-order" part in the subsection on logic-automata connections (§1.3.1).

But before that, there is another source of diversity in automata theory that cannot go unmentioned: the choice of input structure. For instance, there exist *tree automata*

[13]The area is generally called finite *semigroup* theory instead – a semigroup is like a monoid, but does not necessarily have an identity element. There is not much technical difference since an identity can be adjoined freely to a semigroup. (However, subsemigroups are often more useful than submonoids.)

[14]Ramsey's theorem is a celebrated result from the early history of combinatorics on monochromatic cliques in edge-colored complete graphs. It has spawned a whole subfield called Ramsey theory, with many ties to other areas of mathematics (for instance set theory and the geometry of infinite-dimensional vector spaces, as can be seen in the recent PhD thesis [Ran18]). The special case useful for automata theory considers edge colors taken in a finite monoid and required to be "additive" i.e. compatible in some way with a given order on the vertices. This "additive Ramsey theorem" turns out to be equivalent in logical strength (in the sense of reverse mathematics) to fundamental results on automata over infinite words [Koł+19].

[15]It is said to be analogous to Ramsey's theorem in both [Sim90] and [Pin21a, Chapter 18] but there is no consensus on whether the analogy is truly relevant (according to personal communication from Charles Paperman). Unlike in Ramsey's theorem, the monoid structure is essential for factorization forests.

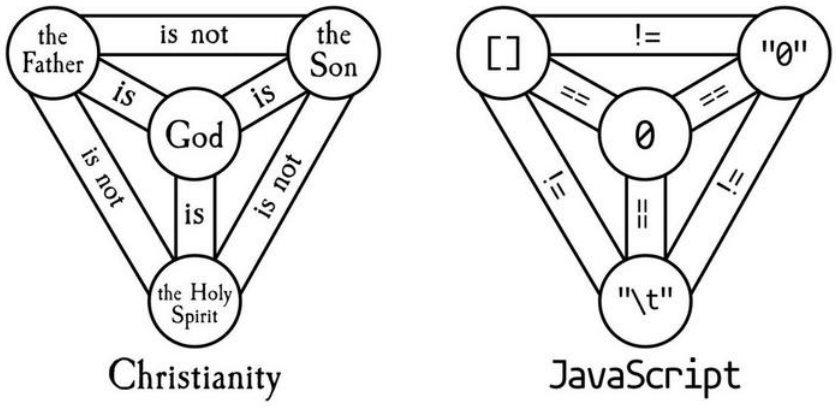


FIGURE 1.1.2. The Holy Trinity of JavaScript (widespread Internet meme).

taking trees as output. The main "practical" motivation for such devices is to process hierarchically structured data, such as HTML documents.[16] Here we must note a difference with computability and complexity, which are mostly indifferent to input encodings: all data is ultimately serialized as string of bits in real computers, and in particular a tree could very well be given as text with matching delimiters (such as HTML tags). The issue with automata is that they are too weak to recover the structure of a tree from its string representation.[17] Therefore, tree automata do not reduce to string automata; furthermore, they have a rich algebraic theory of their own (the structures involved are more complicated than monoids, see e.g. [Pin21a, Chapter 22]). And of course, there are also tree transducers.

There is a more radical departure from computability/complexity in the theory of automata taking *infinite structures* as inputs [Pin21a, Chapters 6 and 8]. While there exist meaningful notions of computability over infinite data, the languages of infinite words or trees recognized by automata models are generally *uncomputable*.[18] One major impetus for studying such seemingly strange devices is their connection with logic, that we shall describe in §1.3.1. For now, let us just say that logic provides alternative characterizations of various classes of languages and transductions, demonstrating that those classes are somehow natural. One could even say that logic, computation and algebra are three tightly entangled aspects of the kind of automata theory featured in this dissertation.

The other half of "logic in CS" or "Volume B" has a celebrated logic/computation/algebra trinity[19] of its own, and it is finally time for us to properly introduce this topic. We shall start by focusing on the logical and computational sides, and cover the algebra later.

[16] In the 2000s, many tree automata papers purported to be motivated by XML processing. At the time, XML seemed poised to take over the Web, with the standardization of various formats (XHTML, SVG, …) and technologies (XSLT, XPath/XQuery, …). But the trend faded away: HTML5 renounced the rigidity of XHTML and JSON emerged as a more lightweight alternative for most data serialization purposes.

[17] As a famous StackOverlow answer puts it: "Parsing HTML with regex summons tainted souls into the realm of the living" (regex = *reg*ular *ex*pressions, cf. Footnote 12 – though "regex" is sometimes used (e.g. in [Sch19]) to refer to the extended regular expressions with backreferences supported by Perl and by the PCRE library). Formally speaking, the Dyck language of well-bracketed words using the letters '`(`' and '`)`' is not regular. It is also TC^0-complete (see [BC89] for membership; completeness is folklore), so these format conversions are not completely innocuous either from the point of view of circuit complexity.

[18] It is worth noting that several questions around computability for transductions between infinite words have been tackled by recent research [Dav+20; FW21].

[19] Robert Harper, a programming languages researcher, coined the term "computational trinitarianism" in a 2011 blog post; see `https://ncatlab.org/nlab/show/computational+trinitarianism` (and for another trinity in computing, see Figure 1.1.2; tip: use the strict equality operator `===` instead!).

1.1.4. **From proofs-as-programs to implicit complexity.** An influential principle in contemporary programming language theory is the *proofs-as-programs correspondence*. The idea is that, in certain logical systems, formal proofs admit a "normalization procedure" that can be seen as the execution of a program – or, perhaps more accurately, as the evaluation of an *expression*, i.e. a piece of code that denotes a value. According to this analogy, a proof is thus a expression[20] and the *formula* that it proves is the *type* of the expression: a label that tells us what kind of result its evaluation yields. A remarkable empirical fact is that this manifests as several concrete isomorphisms between proof systems and (theoretical) programming languages that were originally designed independently.

The first instance of this correspondence, due to Howard[21] [How80], was the relation between intuitionistic propositional logic[22] and the *simply typed λ-calculus*. The latter is obtained by adding a *type system*[23] on top of the (untyped) λ-calculus that we mentioned in the very beginning. To illustrate this, consider two λ-terms (i.e. expressions in the λ-calculus) t and u. One can always combine them into a new λ-term $t\,u$, the *application* of t to u, whose intended intuitive meaning is that the function t is fed the argument u. But even if this is allowed, it might not make sense: can you expect to get a meaningful result if you give a string as argument to a function that expects an integer?[24] Type systems classify λ-terms to avoid such situations. In this example, the simply typed λ-calculus requires t to have a type of the form $\sigma \to \tau$ (notation: $t : \sigma \to \tau$) and u to have the type σ in order for the application to be *well-typed*; moreover, in this case, $t\,u : \tau$. The logical counterpart of this typed application is the deduction rule "if $A \Rightarrow B$ and A then B" (whose traditional name is *modus ponens*); application can be seen as a way to combine a proof of $A \Rightarrow B$ with a proof of A to get a proof of B.[25]

Here it is important to understand that the λ-term $t\,u$ is just a piece of source code that says "apply t to u"; it is not in itself the *result* of this application. To get the result, one must

[20]The usual slogan is "a proof is a program" but in real-world programming languages, expressions are merely parts of programs and can almost never constitute whole programs by themselves. Arguably, a Haskell program could be seen as ultimately being an expression of type `IO ()` but even then, this is too narrow for us: we want to state a correspondence with logic that works for all formulas/types.

[21]Since Howard was inspired by an earlier remark by Curry, the proofs-as-programs correspondence is also known as the *Curry–Howard* correspondence; another name is *formulae-as-types* (from the title of [How80]).

[22]Intuitionistic logic is a variant of classical logic that rejects the law of excluded middle "A or (not A)". What this restriction achieves is that intuitionistic logic is constructive; this means that, for instance, in intuitionistic arithmetic, a proof of $\forall x \in \mathbb{N}.\, \exists y \in \mathbb{N}.\, P(x, y)$ (implicitly) provides a way to "construct", given some x, a y such that $P(x, y)$. The notion of construction is not precisely defined, but concretely, the function that maps x to y is *computable*. More generally, the *Brouwer–Heyting–Kolmogorov interpretation*, introduced in the 1930s, informally explains the meaning of intuitionistic formulas in terms of constructions; and long before the proofs-as-programs correspondence, the theory of *realizability* gave this idea a formal incarnation with computable functions as constructions. (Another famous computational interpretation of intuitionistic arithmetic, Gödel's *Dialectica intepretation* [Göd58], also predates proofs-as-programs; it will be discussed again in §1.1.8 and Footnote 59.) Therefore, Howard's contribution was not the mere idea of relating proofs and computation; it was the stronger observation that syntactic proofs are *isomorphic* to typed λ-terms, while the BHK interpretation only saw them as indirect linguistic descriptions of "constructions".

[23]In this manuscript, type systems are always *static*: the type of each expression is determined without evaluating it. This is an usual meaning of "type system" in academic circles. However, programmers often speak of "dynamically typed" languages; "types" in this context are labels attached to values during program execution. (There is also a somewhat orthogonal "strongly typed" vs "weakly typed" distinction, referring to the type-casts allowed or implicitly performed: the type system of C is often considered static but weak.) For a discussion of this polysemy of the word "type" in computing, see [Pet15].

[24]In some weakly typed languages, you might get a result using implicit conversions, whose behavior might be unexpected: see `https://www.destroyallsoftware.com/talks/wat` and Figure 1.1.2.

[25]Observe that this is very close to the BHK interpretation (Footnote 22) of implication: "a proof of $A \Rightarrow B$ is a construction that turns proofs of A into proofs of B".

execute this code. In the untyped λ-calculus, the execution might get stuck in an infinite loop, but this never happens for λ-terms that are well-typed according to the simply typed λ-calculus. Thus, the type system ensures *termination* by ruling out all non-terminating λ-terms; but it also excludes some well-behaved λ-terms along the way, and the price to pay is the loss of Turing-completeness[26] (i.e. the ability to express all computable functions). From a logical point of view, this is an acceptable price: termination is highly desirable since it entails *consistency*, that is, the absence of contradiction in a logic.[27]

Such a termination guarantee might even come with quantitative time complexity bounds. For instance, Hillebrand et al. [HKM96] show that simply typed λ-terms operating over certain data encodings and returning booleans can compute all functions[28] in the complexity class ELEMENTARY (i.e. those with a time complexity bounded by a tower of exponentials), and only those. This result illustrates the type-theoretic approach to *implicit computational complexity* (ICC), a well-established field concerned with machine-free characterizations of complexity classes via high-level programming languages.

1.1.5. **Linearity in types and in automata.** Many works in implicit computational complexity have taken inspiration from *linear logic* [Gir87] to design more sophisticated type systems; see [Péc20, §1.2.4] for a survey. The key concept of linear logic is that a proof that "A linearly implies B" must use A "exactly once" to deduce B. From a computational point of view, this has a clear meaning: a function is linear when it does not duplicate nor discard its argument. Since duplication is a major source of complexity in the λ-calculus, controlling it can lead to complexity bounds, hence the relevance to ICC.

An important point for us is that linearity has a counterpart in the old theme of *restricting the copying power* of automata models (see e.g. [ERS80]). The latter is manifested in one of the alternative definitions of regular functions (recall from §1.1.3 that these are the functions computed by two-way transducers): *copyless*[29] *streaming string transducers* (SSTs) [AČ10]. An SST is roughly speaking an automaton whose internal memory consists of a state (in a finite set) and some string-valued registers, and its transitions are copyless when they compute new register values without duplicating the old ones. The term "linear" itself has been used to describe this condition, e.g. "updates should make a linear use of registers" [FR16, §5], although strictly speaking one should speak of *affineness*: duplication is forbidden, but not erasure. One also finds the terminology *single use restriction* in various tree transducer models [EM99; BE00; BD20] where it sometimes stands for more sophisticated restrictions that mere copylessness. And more recently, copylessness has been shown to play an important role in the theory of automata over infinite alphabets [BS20].

[26] A programming language whose programs always terminate cannot be Turing-complete – in particular, it cannot define *its own interpreter*! (See [BP16b], which also counter-intuitively exhibits a self-interpreter for a terminating language that operates on typed representations of programs; the impossibility result assumes that the interpreter takes a string containing raw source code as input.) Most real-world statically typed languages choose to be Turing-complete and to allow non-termination; the type system is still useful in that case, e.g. to detect whole classes of bugs during compilation, following Milner's dictum that "well-typed programs do not go wrong" [Mil78] (but one could argue that infinite loops are one way of "going wrong"…).

[27] Normalization of proofs, the logical counterpart of program execution, was first investigated by Gentzen in the 1930s for the purpose of proving the consistency of Peano arithmetic, and his main result (the so-called *Hauptsatz*) was a termination guarantee. Nowadays, terminating programming languages can be found in proof assistants based on the proofs-as-programs paradigm.

[28] This does *not* mean that a given algorithm with elementary complexity must admit a direct implementation in the simply typed λ-calculus. Instead, what must exist is a λ-term computing the same function from inputs to outputs, with potentially different inner workings. (The nuance is admittedly vague since the notion of "equality of algorithms" does not have a proper mathematical definition.)

[29] The adjective "copyless" does not appear in the original paper [AČ10] but is nowadays commonly used to distinguish them from the later *copyful SSTs* [FR21].

Let us briefly digress to review some other uses for linear logic. For theoreticians, it has proved to be an invaluable tool in the fine-grained study[30] of the (non-linear) λ-calculus and its variants, thanks to translations of intuitionistic logic into linear logic. It also led to the invention of *proof nets* – graph-like representations of proofs first introduced in Girard's seminal paper on linear logic [Gir87] – and inspired another way to build proofs known as *deep inference* [Gug07]; both are innovative approaches to structural proof theory.[31] On the programming side, linear types are especially useful for resource management.[32] The last application that we will mention – our list is far from exhaustive – concerns neither proofs nor programs, but linguistics; the relevant keyword for this line of work is "categorial grammar" (*not* "categorical"!), see [MR12]. It started out with the Lambek calculus [Lam58], which is actually a linear λ-calculus *ante litteram*, predating the official birth of linear logic by three decades. An interesting feature of the Lambek calculus is that its type system enforces not only linearity but also *non-commutativity*: roughly speaking, functions must use their arguments in the same order that they are given in.[33] As we shall see later, one of the main results of this dissertation involves a non-commutative λ-calculus.[34]

1.1.6. **Towards implicit automata: computation on Church encodings.** We have now given enough background to explain one of the main ambitions of this manuscript: to provide for automata what (type-theoretic) implicit complexity has done for complexity classes. One expected benefit would be, typically, to turn the analogy between linear types and copyless / single use automata into precise technical connections. There are some previous works tackling this theme of "implicit automata", but not many (see §1.3.3). Our main inspiration is a remarkable yet little-known[35] result by Hillebrand and Kanellakis: *a language can be defined by a simply typed λ-term taking* Church-encoded strings *as inputs if and only if it is regular* [HK96, Theorem 3.4].

[30] Melliès has coined the term "micrologic" [Mel17b] to describe the quest for the "elementary particles" of logic. This stated ambition should perhaps not be taken too literally, but linear logic sometimes feels like a significant step in that direction, e.g. intuitionistic connectives are broken into "more primitive" linear connectives. Let us give a concrete example of a technical breakthrough achieved through this approach: the proof of the time invariance thesis for the (untyped) λ-calculus (cf. Footnote 3) in [AL16] relies on the *linear substitution calculus*, which is heavily inspired by linear logic, and whose role is roughly to decompose evaluation (β-reduction) into finer steps.

[31] Actually, both have forerunners in category theory: proof nets are related to the free compact closed categories from [KL80], as explained in e.g. [Mel18, §1]; and for deep inference vs categorical logic, see [Hug04].

[32] The most noteworthy industrial success in this vein is perhaps the Rust systems programming language, whose notion of *ownership* depends on linearity: each value can only have one "owner" at a time. The academically-minded reader might want to read the recent column [Jun+21] on Rust. It also covers the formalization efforts surrounding it, taking place in a framework for program verification called *separation logic*; and separation logic itself is based on a variant of linear logic (namely the *logic of bunched implications* [OP99]). Other applications of linearity to resource management can be found in [Bak92; Mun18] (with a historical overview in [Mun18, §3]). Finally, let us note that version 9.0.1 of the Glasgow Haskell Compiler, released in February 2021, added support for the `LinearTypes` extension (based on [Ber+18]).

[33] In versions of linear logic that admit a tensor product, this makes the formulas/types $A \otimes B$ and $B \otimes A$ non-isomorphic in general, hence "non-commutativity". On λ-terms or proof nets, non-commutativity corresponds to a topological *planarity* condition; to our knowledge, this was first remarked in [Gir89b, §II.9]. For more recent works pursuing this idea, see e.g. [APR05; Abr07; Mel18]; there is an especially noteworthy connection with bijective and enumerative combinatorics [ZG15]. Finally, there is another tradition that studies extensions of commutative linear logic with a self-dual non-commutative connective, starting with pomset logic [Ret97] (defined using proof nets) and continuing with BV [Gug07] (whose proof system was the first based on deep inference); see [NS22] for a recent overview.

[34] Non-commutative types have also been used to restrict the expressive power of a domain-specific programming language in [KSK08].

[35] At least this was the case when I started my PhD, see e.g. Damiano Mazza's answer to this MathOverflow question: `https://mathoverflow.net/q/296879` – maybe this theorem has become more famous since then!

To explain this statement, we must first remark that many λ-calculi, including Church's original λ-calculus, do not feature integers or strings as primitive data types. But such data can still be represented indirectly as functions,[36] and one standard way to do so is by using *Church encodings*. For instance, $n \in \mathbb{N}$ is represented, morally, as the n-fold iterator $f \mapsto f \circ \cdots (n \text{ times}) \cdots \circ f$ (this is a *higher-order function*: a function that takes another function as its argument). This encoding of natural numbers was used by Church to define computable functions $\mathbb{N} \to \mathbb{N}$ in the λ-calculus; the idea was later generalized by Böhm and Berarducci [BB85] to encodings of all "algebraic data types" (to use functional programming jargon), a class that includes strings and trees.

In some terminating typed λ-calculi, most conceivable functions can still be programmed over these encodings.[37] This is very far from being the case for the simply typed λ-calculus, as Hillebrand and Kanellakis's theorem shows. Consistently with this, the characterization of ELEMENTARY in the simply typed λ-calculus [HKM96] that we mentioned earlier uses a somewhat unusual (though entirely justified) representation for the inputs: in this perspective, the fact that Church encodings only allow regular languages to be recognized is an obstruction to be overcome in order to capture an interesting complexity class. We prefer to see this fact as a feature rather than a bug: it is our starting point to relate λ-calculi to automata. (Church encodings also play an important role in another well-established topic at the interface between λ-calculi and automata, *higher-order model checking*, see §1.3.4.)

Since "λ-term taking Church encodings as inputs" can be ambiguous, let us give a more technical statement (which is not necessary to follow most of the rest of the introduction).

Theorem 1.1.4 (Hillebrand & Kanellakis [HK96, Theorem 3.4])**.** *A language $L \subseteq \Sigma^*$ can be defined by a simply typed λ-term[38] of type $\mathtt{Str}_\Sigma[\tau] \to \mathtt{Bool}$ for some simple type τ (that may depend on L) if and only if it is a regular language.*

Some explanations are in order. A type of the simply typed λ-calculus, or *simple type* for short, is either the *base type* $\mathbb{o}$ or a *function type* $\sigma \to \tau$ where σ and τ are types; in other words, types are syntax trees with binary nodes $\to$ and leaves $\mathbb{o}$. We write $\mathtt{Bool}$ and $\mathtt{Str}_\Sigma$ to denote the types for the Church encodings of booleans and strings respectively: they have the property that the values[39] of type $\mathtt{Bool}$ (resp. $\mathtt{Str}_\Sigma$) are in canonical bijection with $\{\mathtt{true}, \mathtt{false}\}$ (resp. Σ^*). We use the notation $\sigma[\tau]$ for the substitution of all occurrences of $\mathbb{o}$ in σ by τ; since the base type $\mathbb{o}$ is morally "generic" – it is not a primitive data type! – every λ-term of type σ can also be given the type $\sigma[\tau]$. In particular, each string $w \in \Sigma^*$ has an encoding $\overline{w} : \mathtt{Str}_\Sigma$ (recall that $t : \tau$ means "t has type τ"); according to the previous remark, $\overline{w} : \mathtt{Str}_\Sigma[\tau]$ for every simple type τ. Thus, any λ-term $t : \mathtt{Str}_\Sigma[\tau] \to \mathtt{Bool}$ defines a function $\Sigma^* \to \{\mathtt{true}, \mathtt{false}\}$, that corresponds to a language: given an input $w \in \Sigma^*$, the application $t\ \overline{w}$ has type $\mathtt{Bool}$ and thus evaluates to either $\mathtt{true}$ or $\mathtt{false}$.

The main ambiguity in our initial informal statement is that it could be interpreted without the type substitution in the input. We discuss this, as well as some open questions related to definability over Church encodings, in Section 1.3.5. But the results of this dissertation will always involve this kind of substitution.

[36] For practical programming, it can sometimes be useful to represent data as functions. For instance, *difference lists* are a functional representation of lists used by Haskell programmers, that supports a more efficient concatenation operation than usual linked lists. More philosophically, this idea of "data as functions" is also a sort of converse to "code is data", one of the pillars of computing since the universal Turing machine and the von Neumann stored-program architecture. (Or, as a pithy soundbite: Lisp is dual to Haskell.)

[37] For example, this is the case for the polymorphic λ-calculus a.k.a. System F, see e.g. [Wad07].

[38] A technical detail: one must consider a *closed* λ-term, i.e. without free variables. This also applies to the following paragraph: "any *closed* λ-term $t : \mathtt{Str}_\Sigma[\tau] \to \mathtt{Bool}$ defines a function $\Sigma^* \to \{\mathtt{true}, \mathtt{false}\}$".

[39] The precise technical meaning of "value" here is "β-normal η-long term"; the important point is that every simply typed λ-term evaluates to exactly one value.

1.1.7. **Implicit complexity meets denotational semantics.** We will now introduce an important tool used to prove Theorem 1.1.4: the naive reading of the simply typed λ-calculus as a way to define set-theoretic functions. One can assign to each simple type τ a set $[\![\tau]\!]$ called its *denotation*, and to each term $t : \tau$ a denotation $[\![t]\!] \in [\![\tau]\!]$, in such a way that (among other properties) $[\![\sigma \to \tau]\!] = [\![\tau]\!]^{[\![\sigma]\!]}$ and $[\![t\,u]\!] = [\![t]\!]([\![u]\!])$. The second equation makes sense because when $t\,u : \tau$ is well-typed, there must exist (according to the type system) some σ such that $t : \sigma \to \tau$ and $u : \sigma$, and then $[\![t]\!] \in [\![\tau]\!]^{[\![\sigma]\!]}$ is a function from $[\![\sigma]\!]$ to $[\![\tau]\!]$ while $[\![u]\!] \in [\![\sigma]\!]$ so $[\![t]\!]([\![u]\!])$ is a well-defined element of $[\![\tau]\!]$.

This set-theoretic interpretation obeys an important *invariance* property: if two λ-terms evaluate to the same value, then they have the same denotation. This means that computing the denotation of a term is somewhat related to evaluating it; what is particularly useful for us is that when the type of a term t is simple enough, one can "read" the value of t from $[\![t]\!]$. Thus, a conceivable strategy to compute this value is to compute $[\![t]\!]$ by operating on functions between sets. This is made possible by the *compositionality* of the interpretation (compare with our discussion of monoids of behaviors in §1.1.2): the denotation of the application $t\,u$ is determined by the denotation of its subterms t and u (plus some other similar properties, since application is not the only way to build λ-terms).

The readers who followed the technical details in the statement of Hillebrand and Kanellakis's theorem may be interested in the proof sketch that we give below leveraging the above ideas; others can safely skip to the subsequent discussion.

Proof idea for Theorem 1.1.4. We prove the non-trivial direction ("only if"): the language L defined by a simply typed λ-term $t : \mathtt{Str}_\Sigma[\tau] \to \mathtt{Bool}$ must be regular.

The idea is that $[\![\mathtt{Str}_\Sigma[\tau]]\!]$ can be given a monoid structure, by using as monoid multiplication the denotation of a λ-term that computes the concatenation of two encoded strings. One then checks that $\varphi : w \in \Sigma^* \mapsto [\![\overline{w}]\!] \in [\![\mathtt{Str}_\Sigma[\tau]]\!]$ is a monoid morphism by compositionality. We have $L = \varphi^{-1}(\{x \mid [\![t]\!](x) = [\![\mathtt{true}]\!]\})$, assuming that $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$; this can be shown to hold when the denotation of the base type is of cardinality at least 2. (We are free to choose the set $[\![\mathfrak{o}]\!]$, and this determines inductively, by $[\![\sigma \to \kappa]\!] = [\![\kappa]\!]^{[\![\sigma]\!]}$, the denotation of all other types.) Finally, in order to apply Definition 1.1.1 and deduce the regularity of L, we want $[\![\mathtt{Str}_\Sigma[\tau]]\!]$ to be finite, which is the case when $[\![\mathfrak{o}]\!]$ is finite. □

The same paper [HK96] contains a more sophisticated application of this set-theoretic interpretation: characterizations of k-EXPTIME and k-EXPSPACE for all $k \in \mathbb{N}$ (including P = 0-EXPTIME) in an extension of the simply typed λ-calculus with constants. In general, one can also use other *denotational semantics* – interpretations of terms and types that satisfies the invariance and compositionality properties mentioned previously – for this kind of strategy. The technique using denotations to compute values[40] is called *semantic evaluation*. It has also been succesfully applied, for example, to study the complexity of evaluating simply typed λ-terms [Ter12] and to implicit computational complexity in other typed λ-calculi such as System T or PCF[41] (see e.g. [Kri12] and the references therein).

However, it should be noted that most of the works in ICC based on linear logic do not use denotational semantics. The influential methodology introduced by Girard's Light Linear Logic [Gir98] is to control the complexity of syntactic evaluation thanks to combinatorial

[40] There is a loosely related idea in type theory, applied in very different contexts, called *normalization by evaluation*; see for example [Abe13]. In this broad family of ideas, we should also mention the application to ICC of realizability semantics with bounded-complexity programs as realizers [DH11; HL21].

[41] Both are extensions of the simply typed λ-calculus (though System T was originally formulated as an axiomatic theory in [Göd58]), supporting natural numbers as a primitive data type. But PCF does not correspond to a consistent constructive logic: it is Turing-complete, and therefore necessarily non-terminating.

invariants (deemed "geometric"[42]) *that do not depend on types*. This means that those works take place in settings where untyped proofs/terms terminate;[43] in fact, the underlying untyped language may suffice to get an ICC result, as in [Lau20] for instance. This is rather uncommon in the world of proofs-as-programs: usually, types are the reason for termination and thus the primary cause of any complexity bound. And semantic evaluation is a way to study this causation, since semantic interpretations are made possible by typing.

Denotational semantics is not just useful for its application to ICC: it is also, in itself, an important subfield of programming language theory. Originally, its goal was to provide a way to rigorously define the meaning of programs, but these days, for "big" enough programming languages, one generally prefers to specify what a program formally means by describing (a mathematical model for) its execution – an approach named *operational semantics*.[44] Still, the denotational approach is well-suited to study minimalistic languages such as the typed λ-calculi considered in this dissertation, and interpreting programs in rich mathematical structures can yield valuable insights. Those insights may then sometimes inform the design of new logical systems. Linear logic itself is a famous historical example, since Girard came up with it by analyzing denotational models[45] for λ-calculi that he had built earlier.[46] More recently, Voevodsky's investigations into topological semantics of dependent type theory (published in [KL21] posthumously) inspired *homotopy type theory* [Uni13]; it is completely unrelated to our work, but still worth mentioning as undoubtedly one of the most significant

[42]French and Italian proof theorists who are familiar with linear logic are fond of speaking of "geometry of computation" (or similar expressions) to describe this kind of analysis of proofs/programs. The choice of words might be influenced by the visual intuitions associated with proof nets, but it is unclear whether pure mathematicians would consider this to be truly geometric. Why not "combinatorics of interaction" instead? The initial aim of "geometry" was to suggest something less *ad hoc* than operational semantics [Gir89b, §III.1]; one should perhaps be careful not to reenact Bourbaki's historical contempt towards discrete math… (the problem was not just French: Whitehead allegedly said that "combinatorics is the slums of topology" [Cam11]).

[43] Recalling that termination implies consistency, when we put types on top of such an untyped language, the logic that we get is consistent no matter what the type system looks like. Light Linear Logic was indeed designed to avoid a logical inconsistency, to wit, Russell's paradox in naive set theory (see also [Ter04]).

[44]Most real-world programming languages do not have any mathematically precise specification at all: either there is no specification except for how the reference implementation behaves, or there is a standard written in natural language that can be ambiguous. (For instance, you can test your knowledge of fine details in the C++17 standard on `https://cppquiz.org/`.) Standard ML was the first "realistic" programming language designed directly with a formal semantics [MTH90], and it was very influential for this reason. From [MHR20, §6.3]: "While Michael Gordon and Milner had written, but not published, a denotational semantics for LCF/ML in the late 1970s, that approach was abandoned for the definition of Standard ML – instead, an operational approach was adopted. […] These operational approaches had several advantages over denotational techniques" (a list of such advantages follows). A major application of formal semantics of real-world languages in the 21st century is certified compilation: to convince a proof assistant that a compiler is correct, the meaning of the source code that it takes must be rigorously specified. Some examples of certified compilers are CakeML [Kum+14] (for a subset of Standard ML) and CompCert [Ler09] (for the C language – see also [Kre15] for a formalized version of the C standard).

[45]We use "model" as a synonym for "semantics" when there is no risk of confusion (e.g. with the relational structures of §1.3.1 which are models in the traditional sense i.e. Tarskian ($\neq$ denotational) semantics).

[46]To quote [Gir87, §III]: "For logic, computer science is the first real field of application since the applications to general mathematics have been too isolated. The applications have a feedback to the domain of pure logic by stressing neglected points, shedding new light on subjects that one could think of as frozen into desperate staticism […]. Linear logic is an illustration of this point: everything has been available to produce it since a very long time; in particular, retrospectively, the syntactic restriction on structural rules seems so obviously of interest that one can hardly understand the delay of fifty years in its study. Computer science prompted this subject through semantics: […]". For a firsthand account of the birth of linear logic and its very early history, see [Gir87, §V] where the curious idea of a "Taylor expansion" of λ-terms is mentioned. This, together with further semantic motivations (in particular vector space semantics, see [Ker18] for recent developments on the subject), would later give rise to *differential linear logic* [ER03; Ehr18] where one can take "derivatives" of proofs/terms extending the analogy between linear types and linear algebra.

developments in type theory since the start of the millennium (it includes in particular a new logical treatment of equality[47] and a proposal for new foundations of mathematics).

1.1.8. **Categorical semantics.** But how to define precisely what is a denotational semantics for, say, the simply typed λ-calculus? A first attempt could be to consider that the denotation of a type τ should be a set $[\![\tau]\!]$ equipped with some additional structure, while a λ-term $t : \tau$ would have as its denotation an element $[\![t]\!] \in [\![\tau]\!]$. For the interpretation of types to be compositional, $[\![\sigma \to \tau]\!]$ should be determined by $[\![\sigma]\!]$ and $[\![\tau]\!]$; the natural option would be to take the set of structure-preserving functions between them. This fits with the set-theoretic semantics that we saw previously, as well as with several important denotational models, such as the partially ordered sets[48] used for semantic evaluation in [Ter12; Kri12]. However, some semantics do not fit into this mould: their interpretation of function types does not consist of mere input-output maps. For example, in *game semantics*[49] (see e.g. [CC21] for a modern account), a denotation $[\![t]\!] \in [\![\sigma \to \tau]\!]$ records some information on *how* the term t processes an input of type σ to yield an output of type τ.

This echoes earlier developments in structuralist mathematics (told for instance in the book [Cor04]). In the early-mid 20th century, the Bourbaki group had the ambition to rebuild mathematics with a notion of *structure* at its center. They conceived of structures as collections of sets with auxiliary data, plus maps between those sets compatible with the data; this encompassed various algebraic structures (monoids, groups, rings, …) and morphisms between them, topological spaces and continuous functions, etc.[50] But it turned out later to be extremely fruitful to generalize this further: a *category* consists of a collection of *objects* that are not necessarily sets[51], and of a collection of *morphisms* that are not necessarily set-theoretic maps. Instead, the desirable properties of sets and maps become axioms included in the definition of categories (each morphism has a source and a target that are both objects, two morphisms whose source and target match can be composed, …).

It makes sense, then, to use categories (with additional properties) for the denotational semantics of typed λ-calculi. Hence the "computational trinity" of programming languages:

$$\begin{array}{ccccc} \textbf{formulas} & \longleftrightarrow & \textbf{types} & \longleftrightarrow & \textbf{objects} \\ \textbf{proofs} & \longleftrightarrow & \textbf{programs} & \longleftrightarrow & \textbf{morphisms} \end{array}$$

[47] The key to this new perspective on equality is the *Univalence Axiom*; the very vague idea is that this makes equality the same thing as a certain notion of "equivalence" so that, for example, two isomorphic algebraic structures are equal. It is validated by Voevodsky's semantics, but at first, there was no known way of computing with this new axiom. This problem was solved by *cubical type theories* (see [Mör21]), which, interestingly, were also reverse-engineered from a constructive (cf. Footnote 22) denotational semantics! By the way, these semantics have a reasonable claim at being "geometric" (cf. Footnote 42) since they are close to topological spaces up to homotopy. (Some would argue that the recent normalization proof for cartesian cubical type theory [SA21] is also geometric for different reasons, namely its reliance on Grothendieck topoi.)

[48] More precisely, we are speaking of a kind of posets called *Scott domains* (that can also be seen as exotic topological spaces); morphisms between them are not just monotone, but also commute with directed suprema. Domains are famous for providing the first non-trivial model for the *untyped* λ-calculus [Sco70]; the same paper also presents a conceptual analysis of computation that led to their definition. Hyland has argued that in retrospect, Scott domains are not at all the simplest conceivable such model [Hyl10, §3].

[49] An important historical precursor to game semantics is Berry and Curien's model of sequential algorithms [BC82]. It led Berry to argue against set-based definitions of denotational models proposed by previous authors and in favor of Lambek's categorical proposal [Ber81].

[50] Algebraic structures are the product of an axiomatic approach to algebra pursued in the early 20th century by German algebraists (the central figure being Emmy Noether) – which we now take for granted when we work e.g. with arbitrary monoids defined as sets endowed with an associative binary operation and a unit. Point-set topology was also invented around the same time.

[51] Well, if one uses ZF(C) as a foundation of mathematics, everything is technically a set (e.g. the natural number 42 is a set!). But morally, one is not interested in the set-theoretic elements of an object in a category.

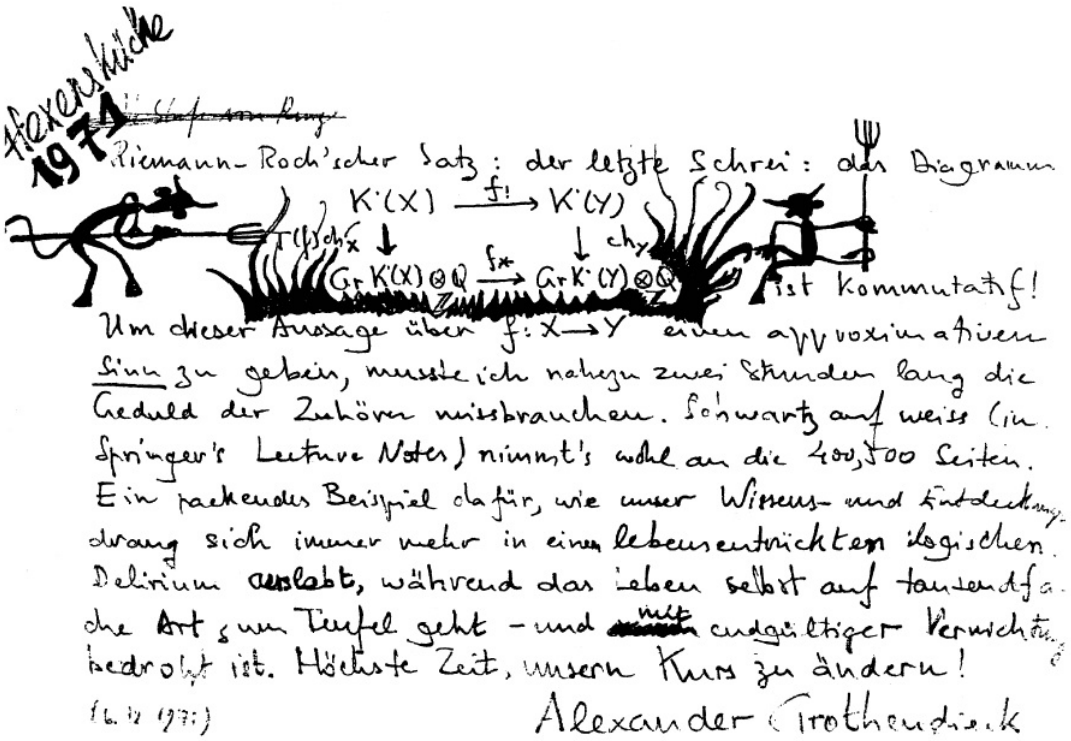


FIGURE 1.1.3. A commutative diagram hand-drawn by A. Grothendieck.
Translation of the letter: "Witches' Kitchen 1971. Riemann–Roch Theorem: The final cry: The diagram [...] is commutative! To give an approximate sense to the statement about $f : X \to Y$, I had to abuse the listeners' patience for almost two hours. Black on white (in Springer lecture notes) it probably takes about 400, 500 pages. A gripping example of how our thirst for knowledge and discovery indulges itself more and more in a logical delirium far removed from life, while life itself is going to Hell in a thousand ways and is under the threat of final extermination. High time to change our course!"
(Image and translation both from http://math.stanford.edu/~vakil/11-245/)

The idea was first advocated by Lambek[52] [Lam72], who showed that *cartesian closed categories* provide denotational models for the simply typed λ-calculus (with product types[53]); it has been widely adopted as the "right" way to conceive of denotational semantics, despite the frightening reputation of category theory as an esoteric subject full of scary diagrams (aptly surrounded by the flames of Hell in Figure 1.1.3).[54]

From a modern point of view, a semantics is thus primarily understood as a category that supports internal operations reflecting those of the programming language being interpreted – for example, the application of a λ-term to another is modeled as (a special case of) composition of morphisms. The mapping $[\![-]\!]$ from types/terms to denotations comes later:

[52]This is why computational trinitarianism is also called the *Curry–Howard–Lambek correspondence*.

[53]Here there is a slight historical mismatch between syntax and semantics: "cartesian" refers to a categorical interpretation of product types $\sigma \times \tau$, which are *not* present in the most common version of the simply typed λ-calculus. *Multicategories* (also known as *colored operads*) can be used to provide a semantics for simple types without products: they handle "multimorphisms" with multiple arguments $A_1, \ldots, A_n$ directly, instead of turning them into morphisms with a single argument $A_1 \times \cdots \times A_n$ using products (see [Hyl17] or [Pel17, Chapter 1]). We will use multicategories in Chapter 6.

[54]This has not prevented some categorical concepts – most famously *monads*, see e.g. [Pet18, §2.1] – from being incorporated into programming practice, especially in the Haskell language and its library ecosystem.

it is the only map[55] that "preserves" the aforementioned operations, e.g. $[\![t\ u]\!] = [\![t]\!] \circ [\![u]\!]$ – this corresponds to compositionality of the semantics. To ensure the other important property, invariance (if t and u evaluate to the same value, then $[\![t]\!] = [\![u]\!]$), one asks that certain equations hold inside the category (for example, composition must be associative[56]).

This means that a category may turn out to provide denotational models for some programming language despite having been defined and studied for *a priori* unrelated reasons; it suffices to exhibit the appropriate structure on the category. Valeria de Paiva's *Dialectica categories* [Pai89] are an example of this (whose application in our work[58] will be discussed in §1.2.3). They originated as a categorical account of Gödel's "Dialectica" functional interpretation [Göd58], a kind of translation from one logical system to another with a computational flavor.[59] It was then noticed that Dialectica categories form a semantics of linear logic – and at a time when linear logic was still very young, this provided an argument in favor of its relevance (no pun intended [Avr14]).

1.1.9. **Automata and categories.** An emphasis on intrinsic algebraic structure is present both in semantics and in the approach to automata behaviors we saw in §1.1.2. In the latter, one starts by coming up with the target monoid; the morphism from strings to behaviors is then uniquely determined by the images of the input alphabet letters (this is what "free monoid" (§1.1.2) means). We could even argue that this is an instance of denotational semantics if the letters are seen as instructions – strings would then be simple sequential programs. Indeed, monoids and categories with a single object are the same thing![60]

More generally, for an object X in a category $\mathcal{C}$, the set $\mathsf{Hom}_{\mathcal{C}}(X, X)$ of morphisms from X to X has a canonical monoid structure (given by composition of morphisms in $\mathcal{C}$). This is particularly relevant to us: thanks to this, our descriptions of the behaviors of various automata models in §1.1.2 can be recast in categorical terms. (By FinSet (resp. FinRel) we mean the category of finite sets and functions (resp. relations) between them.)

- Monoids of behaviors of *deterministic* finite automata are of the form $\mathsf{Hom}_{\mathsf{FinSet}}(X, X)$.

[55] Up to natural isomorphism, if $[\![-]\!]$ is seen as a functor from the syntactic category (whose objects and morphisms are types and terms respectively) to the category of denotations.

[56] This is part of the definition of a category. But programming languages with *side effects* (mutable state, exceptions, input/output…) have a non-associative composition that reflects their sensitivity to the *order of evaluation*. A major focus of the "French school" has been to gain a subtle understanding of such situations using the notion of *polarity*, starting with Girard's work on classical logic [Gir91]. Indeed, one widespread proofs-as-programs counterpart of classical logic (though not the only possible one[57]) is a family of λ-calculi with the side effect of non-local control flow (using "call with current continuation" or similar operators) [Gri90; Par92; CH00]. For modern accounts of polarity and non-associativity, see [Zei09; Mun13]. Let us also mention *classical realizablity* [Kri09] which attempts to give a computational content to axioms in set theory (e.g. the Axiom of Choice [FG20; Kri20]) using side effects.

[57] For radically different alternatives, see e.g. [Alc19, Part II] or [Str11, Chapter 2].

[58] In fact Dialectica-like categorical structures have already appeared in quite varied and unexpected contexts. In "applied category theory", a recent example is [DLP21]; we also highly recommend reading Jules Hedges's *Lenses for philosophers* (`https://archive.md/R72aj`) for an explanation of the connections with *lenses* from functional programming (see e.g. [Cla+20]) and with categorical approaches to game theory and microeconomics [Gha+18]. More in line with the topic of this dissertation, Dialectica-like semantics of linear logic have also been built out of automata on infinite words [PR19] and infinite trees [Rib20].

[59] For a modern account of the Dialectica translation in terms of side effects, following the paradigm described in Footnote 56, see [Péd15] (as well as [KP21] for a connection with differential LL (cf. Footnote 46)).

[60] This observation leads to working with finite categories as a generalization of finite monoids, which can be useful in the context of algebraic language theory, as demonstrated in the (very long) paper [Til87]. The recent work [AP21] that we cited at the end of §1.1.2 relies on the Delay Theorem from [Til87], whose statement involves categories. However, all this is quite different from the kind of connections between category theory and automata that we investigate in this manuscript. Also, feel free to add "locally small" in front of "categories" if you want to be pedantic about sets vs proper classes.

- Monoids of behaviors of *nondeterministic* finite automata are of the form $\mathsf{Hom}_{\mathsf{FinRel}}(X, X)$.
- For *two-way* automata (2DFA), Hines [Hin03] has shown that this pattern can be continued by considering the category $\mathsf{Int}(\mathsf{PFSet})$ (PFSet = *partial* functions on finite sets), defined using the generic Int-construction on traced monoidal categories [JSV96]. It is noteworthy that the not-quite-trivial combinatorics of this automata model can be recovered from categorical "abstract nonsense", which moreover is also related to the so-called *geometry of interaction* semantics of linear logic – we will say more about this connection in §1.4.3.

This observation almost suffices by itself to establish category theory as a language to uniformly describe these different kinds of automata. (Such *unification* and *generalization* is typical of the role played by categories in mathematics.) There remains to take care of a few details, such as the fact that automata have initial and final states – here, morphisms whose source and target do not coincide have a role to play. This has been done by Colcombet and Petrişan [CP17a; CP20], who provide a suitable framework in which one can say that a DFA (resp. NFA, 2DFA) is *an automaton over the category* FinSet (resp. FinRel, $\mathsf{Int}(\mathsf{PFSet})$).

One benefit of this framework is that it allows one to relate the expressive power of various automata models using *functors*, i.e. "morphisms between categories". Let us illustrate this with an example from [CP20, §3.3]. There is a functor $\mathcal{P} : \mathsf{FinRel} \to \mathsf{FinSet}$ that sends any object (finite set) X to its powerset $\mathcal{P}(X)$ and any morphism (relation) $R \subseteq X \times Y$ to the function $S \in \mathcal{P}(X) \mapsto \{y \in Y \mid \exists x \in S.\, (x, y) \in R\} \in \mathcal{P}(Y)$. It induces a translation between the corresponding automata, that is, from DFA to NFA: a *determinization* procedure. As the reader might already have guessed, this is none other than the "powerset construction" covered in undergraduate theory of computation courses! (It is a bit more efficient, in terms of state space explosion, than the determinization through monoids sketched in §1.1.2.)

A large focus of [CP20] is on developing a general automata minimization theorem that can be instantiated on various categories (applications may be found in [CP17b; CPS21]). This kind of use of category theory to understand the essence of various constructions on automata – such as determinization or minimization – and to generalize them to other settings has a long history, see for instance [Hee+19] and the many references therein; it is also among the main interests of the "coalgebra community" (`https://www.coalg.org/`).

1.2. Contributions

Since this document is a PhD thesis, in order to fulfill one of its functions, there must come a moment (much to my chagrin) when I – the author of these lines – argue that I have made some modest contribution to the sum of human knowledge. In this section, I will therefore summarize the portion of my research that I have included in this dissertation.[61] It has been carried out in collaboration with **Pierre Pradic**, and all chapters except for this introduction are based on jointly written material (either published papers[62] [NP20; NNP21] or unpublished drafts) that I have slightly reworked for the sake of a consistent narrative.

[61] I have also been working during my PhD on a mostly unrelated topic, which will not be covered here: applying graph algorithms to tackle old questions on proof nets and deep inference [Ngu20; NS22].

[62] One of the authors of [NNP21], C. Noûs (`https://www.cogitamus.fr/camilleen.html`), is fictional (see also [PS20]). They symbolize the fundamental dependence of any piece of research on the wider community of scientists beyond its handful of coauthors, as well as an opposition to metrics devised by bureaucrats, among other things. It is because of this distaste for bean-counting that I will not guesstimate here which percentage of the work presented here is mine, or worse, indicate which theorems I have personally proven and which ones are due to my coauthor in this manuscript; in any case, since many ideas exposed here emerged from our conversations, this would not be very meaningful. That said, it can sometimes be appropriate to delineate individual contributions in a joint work – for instance, in case of significant differences in seniority between coauthors, or to counteract bias against (or the invisibilization of) under-represented minorities. An example of such authorship statements done right, in my opinion, can be found in [Rin21] (that PhD thesis is also interesting scientifically: it applies univalence (Footnote 47) to practical proof engineering).

Without further ado, let me drop the first-person pronoun and get back to the actual scientific content.[63] (Henceforth, "we" stands for either "Pradic and I" or "the narrator and you, dear reader, together on a wonderful(?) journey" depending on the context.) We do not indicate in this section where our claims are supported in the body of the dissertation; this will be the role of the outline (§1.5).

1.2.1. **A research programme: implicit automata in typed λ-calculi.** As already announced in §1.1.6, a substantial portion of the results of this dissertation consists of characterizing automata-theoretic classes of languages and functions using typed λ-calculi, in the same way that implicit computational complexity (ICC) characterizes complexity classes. There have previously been a few results of this kind, or more than a few if one considers a broad definition of ICC (cf. §1.3.2) that includes non-type-theoretic approaches; we cover this related work in §1.3.3. However, it seems that most of the previous papers on "implicit automata" do not mention the connection with ICC (with the exception of [Sei18]), and therefore that we are the first to give this topic the name that it deserves.

Beyond such superficial naming considerations, our awareness of this historical context inspired us to put forth a new recipe for type-theoretic implicit automata results: *consider functions operating on Church encodings in a λ-calculus with a substructural (e.g. linear / affine / non-commutative / …) and monomorphic type system.* For the reader who is unfamiliar with this technical term, "monomorphic" can be replaced by "admitting a translation into the simply typed λ-calculus (with product and sum types)" in this context while preserving most of the intended meaning; thanks to Hillebrand and Kanellakis's Theorem 1.1.4 (cf. §1.1.6) this condition ensures that we are more likely to get something corresponding to an automata model than a complexity class. This draws upon two distinct traditions within implicit complexity: while Theorem 1.1.4 comes from the study of the simply typed λ-calculus and its extensions by semantic means, previous works on ICC taking inspiration from linear logic mostly did not use semantic evaluation (as we said in §1.1.7).

There are many possible variations on this theme: one can obviously vary the typed λ-calculus under consideration, but the same type system can also be used to impose different constraints. Given the diversity of automata-theoretic function classes (especially concerning transducers, cf. §1.1.3), one could expect these variations to yield characterizations of many different classes. This is why we believe our approach to implicit automata to be worthy[64] of a systematic investigation that the present manuscript has merely begun. Our first results, described in §1.2.2, have not even exhausted the low-hanging fruits! The work-in-progress section (§1.4) should hopefully convince the reader that much remains to be done.

Furthermore, we have come to consider some of our equivalence results about definability of languages or functions as surface-level manifestations of deeper structural connections. As we announced from the start, this is made possible by the emphasis on compositionality, embodied by algebraic tools, that is prevalent in both automata theory and programming languages. What this means concretely is that we express these connections in the language of category theory, which leads us to bring ideas from denotational semantics into categorical automata theory along the way (§1.2.3). Our exploration of the expressive power of typed λ-calculi has also fed back into "pure" automata theory, suggesting relevant objects of study for the latter (§1.2.4). All this has been a welcome surprise: ICC has not had much influence on (nor structural insight about) "mainstream" complexity theory (cf. the end of §1.3.2).

[63] Adrien Douady, former member of the Bourbaki group : "Le but de ce travail est de munir son auteur du grade de docteur ès-sciences mathématiques et l'ensemble $H(X)$ des sous-espaces analytiques compacts de X d'une structure d'espace analytique" (from the introduction to [Dou66]).

[64] Such a turn of phrase implicitly assumes that research in pure mathematics – or scientific research in general – is something worthwhile, which is of course highly debatable (see Figure 1.1.3 and [Gro16]).

1.2.2. **Equivalences in expressive power.** Let us now present our concrete results, starting with one that concerns languages. It takes place in a fragment – that we call the *$\lambda\ell\wp$-calculus* – of Intuitionistic Non-Commutative Linear Logic [Pol01] ('$\wp$' as in "planar", see e.g. [Abr07; ZG15]). As the name indicates, it features non-commutative linear types in the vein of the Lambek calculus (cf. §1.1.5). We have the following:

Theorem 1.2.1 (variant of [NP20, Theorem 1.7])**.** *A language $L \subseteq \Sigma^*$ can be defined by a $\lambda\ell\wp$-term of type $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{NBool}$ for some* purely linear *type τ (that may depend on L) if and only if it is a* star-free *language (cf. Definition 1.1.2 in §1.1.2).*

This mirrors Theorem 1.1.4, with several additional linearity conditions. First of all, observe that the type $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{NBool}$ uses a *linear* function arrow '$\multimap$', indicating that the argument $\mathtt{Str}_\Sigma[\tau]$ is "used only once", rather than the non-linear arrow '$\to$'. We also have a linearity requirement on τ:

Definition 1.2.2. A type is *purely linear* when it does not contain '$\to$'.

In the case of the $\lambda\ell\wp$-calculus, purely linear types are generated by '$\multimap$' from the base type $\mathbb{o}$. There are also other less visible amendments to the pattern of Theorem 1.1.4 in order to take linearity into account: $\mathtt{Str}_\Sigma$ now denotes the type of *linearized* Church-encoded strings [Gir87, §5.3.3], while $\mathtt{NBool}$ is a purely linear type of booleans that works in a linear non-commutative setting. Writing the types with a left-associative convention for function arrows ($\sigma \multimap \tau \multimap \kappa = \sigma \multimap (\tau \multimap \kappa)$) as usual, we have

$$\mathtt{Str}_\Sigma = \underbrace{(\mathbb{o} \multimap \mathbb{o}) \to \cdots \to (\mathbb{o} \multimap \mathbb{o})}_{|\Sigma| \text{ times}} \to \mathbb{o} \to \mathbb{o}$$

$$\mathtt{NBool} = ((\mathbb{o} \multimap \mathbb{o}) \multimap (\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o}) \multimap (\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o}$$

(how these encodings work will be explained in Chapter 5 and Chapter 7 respectively).

Aside from the connection to algebraic language theory, let us point out two less obvious novelties compared to previous work in ICC. First, star-free languages might be, as far as we know, the smallest class of languages ever characterized in ICC: they are a strict subclass of both regular languages and uniform AC^0. Second, our proof for the "if" direction of Theorem 1.2.1 is somewhat deeper than the "programming exercise of limited theoretical interest" [Maz17, §4.2.1] that is common for analogous proofs in ICC: we had to use the Krohn–Rhodes decomposition theorem (mentioned in §1.1.3).

We also show that when the $\lambda\ell\wp$-calculus is replaced by its commutative counterpart in Theorem 1.2.1, we get regular languages instead of star-free languages. This shows that non-commutative types are crucial to get the upper bound on computational power that we want. The key lemma for this upper bound is a property of the *planar λ-calculus*, that is, the purely linear fragment of the $\lambda\ell\wp$-calculus. It seems to be the first result exhibiting this kind of difference between the planar λ-calculus and its commutative version, namely the linear λ-calculus. Indeed, we recently discovered[65] [Das+21] a fact that runs counter to this: normalizing planar λ-terms is P-complete, just like for general linear λ-terms. (The proof – not included in this dissertation – relies heavily on the above type $\mathtt{NBool}$.)

The linearity restriction, by itself (without non-commutativity), does not make any difference compared to the simply typed λ-calculus concerning definable *languages* over Church encodings. However, if we look at *functions*, there is much more to say. While characterizing the class of string-to-string functions corresponding to $\mathtt{Str}_\Gamma[\tau] \to \mathtt{Str}_\Sigma$ is still an open problem in the simply typed λ-calculus (we shall come back to this in §1.3.5), we have managed to prove several equivalences with transduction classes for the *$\lambda\ell^{\oplus\&}$-calculus*.

[65] Joint work with Anupam Das, Damiano Mazza and Noam Zeilberger.

The latter is basically a version of Dual Intuitionistic Linear Logic [Bar96] incorporating additive connectives (i.e. categorical products and coproducts).

Theorem 1.2.3. *We have the following implicit characterizations of transductions in the $\lambda\ell^{\oplus\&}$-calculus, in the style of Theorems 1.1.4 and 1.2.1:*

$\mathtt{Str}_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$	regular string functions
$\mathtt{Str}_\Gamma[\tau] \to \mathtt{Str}_\Sigma$	comparison-free polyregular functions
$\mathtt{Tree}_{\mathbf{\Gamma}}[\tau] \multimap \mathtt{Tree}_{\mathbf{\Sigma}}$	regular tree functions

where Γ and Σ are finite alphabets, $\mathbf{\Gamma}$ and $\mathbf{\Sigma}$ are ranked alphabets and τ is a purely linear *type that may be chosen depending on the function that one wants to define.*

The first row above is not too surprising, given the resemblance between copyless streaming string transducers – that compute regular functions – and linear types that we already discussed in §1.1.5. (Although, as we shall see in §1.2.3, there is slightly more than meets the eye.) *Comparison-free polyregular functions* will be the subject of Section 1.2.4. As for the class of *regular tree functions*, it is is obtained by generalizing the definition for strings based on monadic second-order logic (cf. §1.3.1).

There is also an automata model adapting copyless SSTs to trees, namely *bottom-up ranked tree transducers* (BRTTs) [AD17].[66] However, it is conjectured that some regular functions *cannot* be computed by *copyless* BRTTs. Instead, a more sophisticated linearity condition, called the *single use restriction* in [AD17], is imposed on BRTTs in order to characterize the regular tree functions. The additional flexibility thus afforded, compared to copyless BRTTs, turns out to correspond directly to an important feature of linear type systems: the *additive conjunction* or categorical product, denoted by the '&' in $\lambda\ell^{\oplus\&}$.

This is one reason for working in a calculus with additive connectives. What happens if we drop them? We prove that in this case, Theorem 1.2.3 *stops working even for strings*, but it does so for uninteresting reasons which do not apply anymore if we consider *affine* instead of linear types.[67] This leaves many questions as to the expressiveness of affine λ-calculi without additives; another natural question in line with Theorem 1.2.3 is the possibility of characterizing first-order transductions (recall from §1.1.3 that those are to regular functions what star-free languages are to regular languages). We have tentative answers to all those questions; we do not develop them in this manuscript because they have not yet been written up rigorously, but we will state our claims in §1.4.2. Let us say for now that we have found a situation where, if our claims are correct, the presence or absence of additive connectives makes a true difference concerning expressive power for meaningful reasons; such a phenomenon has not been observed before in implicit complexity to our knowledge (though it is known that additives increase the complexity of proof normalization [MT03]).

1.2.3. **Semantic evaluation and contributions to categorical transducer theory.** Our proof of Theorem 1.2.1 uses purely syntactic methods, i.e. combinatorial manipulations of $\lambda\ell_\wp$-terms. We could perhaps tackle Theorem 1.2.3 syntactically as well, by taking inspiration from the proof techniques used in a recent work by Gallot, Lemay and Salvati [GLS20]: they show that a tree transducer model containing linear λ-terms computes regular tree functions.

But the strategy that we present here for the characterization of regular string functions in Theorem 1.2.3 – which we developed while being unaware of [GLS20] – is arguably more conceptual. (The price to pay is that there are more technical prerequisites.) It is expressed

[66] As we shall explain in the beginning of Chapter 6, a better choice in hindsight would have been to work with *macro tree transducers* [EV85; EM99].

[67] In general, ICC results using affine types tend to also work with linear types if one dedicates part of the output to collect the "garbage" that could not be discarded during the computation; see for example [Lau20].

in a categorical framework for automata, the same as the one introduced by Colcombet and Petrişan (§1.1.9) up to insignificant details. Consider:

- some category $\mathcal{C}$ corresponding to a transducer model, such that the functions computed by $\mathcal{C}$-automata are exactly the regular functions;
- a "syntactic category" $\mathcal{L}$ whose objects are *purely linear* types and whose morphisms are $\lambda\ell^{\oplus\&}$-terms, such that $\mathcal{L}$-automata can be shown to be equally expressive as $\lambda\ell^{\oplus\&}$-terms of type $\mathtt{Str}_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$ (for purely linear τ) through syntactic means.

Our goal is to show that $\mathcal{L}$-automata are at most as expressive as $\mathcal{C}$-automata by exhibiting a functor $\mathcal{L} \to \mathcal{C}$ (with a few additional properties). This would mean that $\lambda\ell^{\oplus\&}$-definable string functions are regular, the converse being easier. According to the discussion in §1.1.5 and §1.2.2, the natural choice would be to take $\mathcal{C}$ to be the category corresponding to copyless streaming string transducers. We can decompose the latter as $\mathcal{SR}_\oplus$ where

- $\mathcal{SR}$ corresponds to *single-state* copyless streaming string transducers: objects are finite sets of register names and morphisms are copyless assignments. (We will also characterize the category $\mathcal{SR}$ through an universal property.)
- $(-)_\oplus$ is the *free coproduct completion*, a standard construction on categories whose automata-theoretic meaning is to add finite states. (Similar completions by certain colimits have been previously exploited [CP17b] within Colcombet and Petrişan's framework.)

At the same time, we would like $\mathcal{C}$ to be a categorical semantics (§1.1.8) of the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus – also known as a *symmetric monoidal closed category* (SMCC), see [Mel09, §4.7]. The functor that we seek would then be the functor $[\![-]\!] : \mathcal{L} \to \mathcal{C}$ that maps each $\lambda\ell^{\oplus\&}$-term (i.e. morphism in $\mathcal{L}$) to its denotation, in the spirit of the semantic evaluation technique described in §1.1.7. Unfortunately, $\mathcal{SR}_\oplus$ is *not* an SMCC.

To fix this, instead of working with $\mathcal{SR}_\oplus$-automata i.e. *copyless* SSTs, we consider a transducer model that is close (although not identical) to *single-use-restricted* SSTs (in the same sense as the so-called single-use-restricted BRTTs mentioned in §1.2.2). Since copylessness and the single use restriction morally differ by the presence of the additive conjunction '&' of linear logic, we "add '&' freely" to achieve a similar effect. Thus, we are led to work with $(\mathcal{SR}_\&)_\oplus$-automata, where $(-)_\&$ denotes the *free product completion* (the dual operation to $(-)_\oplus$). We then get the missing piece of our aforementioned strategy:

Theorem 1.2.4. *$(\mathcal{SR}_\&)_\oplus$ is a symmetric monoidal closed category.*

The double completion $((-)_\&)_\oplus$ is reminiscent of a factorization into free sums and free products of a fibrational Dialectica (§1.1.8) construction [Hof11]. Thanks to this, generic considerations similar to the monoidal closure proof of Dialectica categories can be used to reduce the above theorem to properties of $\mathcal{SR}$ that can then by proved by simple combinatorics on strings.

Theorem 1.2.4 is also meaningful from the point of view of categorical automata theory, independently of any connection with the $\lambda\ell^{\oplus\&}$-calculus. Indeed, an SMCC provides a setting in which constructions relying on *function spaces* (i.e. *internal homsets*) can be carried out in a generic way; this generalizes the kind of proofs where one exploits the finiteness of the set of functions $Q \to Q$ for any finite set of states Q. (A symmetric monoidal category is closed when all internal homsets exist.) To illustrate this, we prove two generic theorems on automata over categories that rely on monoidal closure:

- We show that $(\mathcal{C}_\&)_\oplus$-automata are equivalent to *functional non-deterministic* $\mathcal{C}$-automata with states.[68] We also provide a (somewhat intricate) construction that, assuming that certain internal homsets exist, turns $(\mathcal{C}_\&)_\oplus$-automata into $\mathcal{C}_\oplus$-automata. It can be seen as a

[68] That additive connectives in linear logic have something to do with non-determinism has previously been observed in other settings, for instance in [MT03].

categorical *determinization* procedure, whose main technical ingredient is a data structure called "transformation forest", reminiscent of the Muller–Schupp determinization [MS95] for automata over infinite words (see also [BC18, Chapter 1]). For $\mathcal{C} = \mathcal{SR}$, we get that functional non-deterministic copyless SSTs can be determinized; this was already known, with an indirect proof [AD11] via monadic second-order logic (§1.3.1).

- We give sufficient conditions on $\mathcal{C}$ for the class of functions computed by $\mathcal{C}$-automata to be *closed under precomposition by regular functions*: if f is regular and g is defined by a $\mathcal{C}$-automaton, then there exists a $\mathcal{C}$-automaton that computes $g \circ f$. Those conditions include asking for $\mathcal{C}$ to be an SMCC. For instance, taking $\mathcal{C} = \mathsf{FinSet}$, we recover the fact that regular functions preserve regular languages by preimage (Definition 1.1.3 in §1.1.3). Using Theorem 1.2.4, we can also apply this to $\mathcal{C} = (\mathcal{SR}_{\&})_{\oplus}$.

These two results jointly entail that (the functions computed by) copyless SSTs are closed under precomposition by regular functions. And since we have defined the latter using copyless SSTs, we have actually reproved[69] the fact that the composition of two copyless SSTs can also be computed by a copyless SST. This composition property was previously known, but it is highly non-trivial (in [BC18, Chapter 13], a special case is proved using the aforementioned "transformation forests", and this already requires some efforts); in the light of the second item above, the difficulty can be explained by the fact that $\mathcal{SR}_{\oplus}$ is not monoidal closed. By contrast, composing two $\lambda\ell^{\oplus\&}$-terms of types $\mathtt{Str}_\Gamma[\sigma] \multimap \mathtt{Str}_\Sigma$ and $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{Str}_\Pi$ is completely trivial (the result having type $\mathtt{Str}_\Gamma[\sigma[\tau]] \multimap \mathtt{Str}_\Pi$). Thus, it was to be expected that the same tools used to prove our characterization of regular functions in the $\lambda\ell^{\oplus\&}$-calculus could be used to compose copyless SSTs.

Concerning the rest of Theorem 1.2.3, the result on regular tree functions follows similarly from the monoidal closure of a category $(\mathcal{TR}_{\&})_{\oplus}$ for trees. As for comparison-free polyregular functions, we do not know how to express them directly as the functions computed by automata over some category, but we were able to devise a more ad-hoc semantic evaluation argument that also leverages Theorem 1.2.4 to establish their characterization.

1.2.4. **Benefits for pure automata theory: a new class of transductions.** The class of comparison-free polyregular functions is itself a byproduct of our research on "implicit automata": we discovered it by studying the functions definable by $\lambda\ell^{\oplus\&}$-terms of type $\mathtt{Str}_\Gamma[\tau] \to \mathtt{Str}_\Sigma$ (for purely linear τ). It turned out that they were included in the class of *polyregular functions*, defined in [Boj18] by four equivalent computational models (and a fifth characterization using logic is mentioned in §1.3.1).

One of those models, the *pebble transducers*, is the specialization to strings of a tree transducer model that existed previously in the literature [MSV03; EM03a]. They can also be seen as multi-head automata (with output) that obey a "stack discipline" on their heads. This restriction ensures that polyregular functions preserve regular languages by preimage, suggesting that they belong to the kind of automata theory amenable to algebraic methods, unlike general multi-head automata (see the beginning of §1.1.2, as well as the discussion following Definition 1.1.3). This partially explains the "regular" part of the name, which also hints at the fact that every regular string function is polyregular. As for the prefix "poly", it comes from the polynomial growth property of these functions: $|f(w)| = |w|^{O(1)}$ (where $|w|$ denotes the length of the string w).

Just like multi-head automata, pebble transducers are allowed to *compare* the positions of their reading heads. We discovered that those *without* such comparisons had precisely

[69] We claim that our new proof is "honest" in the following sense: we do not use previous non-trivial results on SSTs as black boxes to establish our categorical theorems, so there are no hidden formal dependencies. But we do largely take inspiration from the preexisting proofs of previous results. Thus, in a way, it is a proof using old techniques, rephrased and factorized through a new abstract apparatus.

the same expressive power as the aforementioned $\lambda\ell^{\oplus\&}$-terms, hence "comparison-free". And since comparison-free polyregular functions did not already exist in the literature, we undertook a purely automata-theoretic study of those functions. (The machine model is quite natural, and some colleagues had thought of it, but they had not yet pushed it further.)

In particular, one insight that the $\lambda\ell^{\oplus\&}$-calculus characterization immediately gave us is that comparison-free polyregular functions are *closed under composition*. This is indeed a trivial observation on $\lambda\ell^{\oplus\&}$-terms (just as in the case of regular functions), but it is far from obvious when looking at comparison-free pebble transducers. To convince an audience unfamiliar with programming language theory, we devised an alternative proof[70] that does not involve any $\lambda\ell^{\oplus\&}$-term or category. But it is nonetheless morally related to our semantic machinery: we use a standard technique for manipulating copyless assignments in SSTs[71] (a decomposition into "shape" plus "labels") which is, in fact, the domain-specific core (as opposed to the Dialectica-like generalities involved) of the proof of Theorem 1.2.4.

Our results on this function class also include:

- alternative characterizations, one of which is the composition closure of regular functions plus some other "basic" functions;
- a relation between the growth rate of a function and the number of pebbles (i.e. reading heads) required to compute it using a comparison-free pebble transducer;
- *separation results*: we give concrete examples of functions demonstrating that
  - some polyregular functions are not comparison-free;
  - comparison-free polyregular functions and polynomially bounded HDT0L transductions are incomparable classes, in other words, neither is included in the other.

To explain the last item above, *HDT0L transductions* are the functions computed by *copyful* streaming string transducers [FR21], and they also admit several other equivalent definitions (e.g. [FMS14; DFG20]). They also preserve regular languages by preimage, and may have exponential growth. Recently, two machine models corresponding to *polynomially growing* HDT0L transductions were introduced by Douéneau-Tabot, Filiot and Gastin [DFG20]:

- *marble transducers*, that can be seen as pebble transducers with a further restriction – this means that every HDT0L transduction with polynomial growth is polyregular;
- *layered streaming string transducers*, that lie in-between copyless and copyful SSTs.

This idea of allowing a limited form of duplication to control expressive power is very similar to the usual approach to implicit complexity using variants of linear logic. In this specific case, we even independently reinvented the "layering" condition by transposing the intuition behind its linear type system to SSTs! The type system in question was that of the *parsimonious λ-calculus* [Maz15; MT15; HMP20], and we will come back to it in §1.4.5.

A few months before the paper [DFG20] appeared on arXiv, we had privately circulated a note on this "new" transducer model. We did not suspect at the time any alternative characterization of the functions computed by layered SSTs, and the main result in this note was that any polyregular function can be realized by a composition of layered SSTs. Since this last property does not appear in [DFG20], we ended up including its elementary proof in our paper [NNP21], and it will also appear in this manuscript. And while we later learned that Douéneau-Tabot et al. had started investigating layered SSTs before us,[72] this

[70] "Alternative" because the detour through the $\lambda\ell^{\oplus\&}$-calculus is a legitimate proof: we do not rely on the fact that comparison-free polyregular functions are closed under composition to show Theorem 1.2.3.

[71] We are able to use SSTs thanks to a reformulation of comparison-free pebble transducers as the closure of 2DFTs under a "composition by substitution" operation, in which 2DFTs can be replaced by any machine model for regular functions (in particular copyless SSTs).

[72] Personal communication from Gaëtan Douéneau-Tabot. Furthermore, we recently found out about a precursor to layered SSTs dating back to the 1960s, see Remark 2.3.15.

anecdote still serves as an additional example of how looking at λ-calculi can lead to new ideas in automata theory. (We will give yet another example in §1.4.3.)

1.2.5. **A methodological commitment: naturality.** Having now exposed our concrete results, let us return briefly to the "programmatic" dimension of this dissertation. We wish to stress that the work presented here has been guided by a desire for *natural* statements and constructions.

Admittedly, it is an inherently subjective notion, devoid of precise technical meaning.[73] But what it means for us is that we explore the expressive power of typed λ-calculi that already existed before, perhaps up to a few minor details – in any case, that are designed according to standard recipes – and whose features have original motivations that are entirely unrelated to complexity or automata. (For example, we mentioned the applications of non-commutative types to linguistics in §1.1.5.) This is unlike a system such as Light Linear Logic [Gir98] (cf. §1.1.7) which is engineered in an ad-hoc fashion in order to get a desired complexity bound[74] – as is often the case, but not always, for ICC results based on linear logic.[75] It is also the same mindset that led us to account for usual features of automata models (finite states and nondeterminism) using simple and classical category-theoretic constructions; our reward for this has been to discover a connection with previous work on denotational semantics. One more example: the Church encodings that we use are completely standard as well.

As a consequence, some of our results have a satisfying feeling of inevitability. For example, in retrospect, Theorem 1.2.1 could have been discovered two decades ago! A more negative take would be to point out that we study old objects using old techniques. But the relationships between the objects are new, and so are the contexts in which the techniques are applied; and those novelties only look inevitable with the benefit of hindsight and the specific background knowledge that we have attempted to impart in §1.1.

1.3. Further related themes

In this section, we will cover some more related work, as well as some background that our research does not directly depend on but which is important to fully grasp its context.

[73] In category theory, *natural transformation* is a technical term (from which "natural isomorphism" – as in Footnote 55 – is derived). In the end notes to [Mac98, Chapter 1], Mac Lane speaks of "the pleasure of purloining words from the philosophers: 'Category' from Aristotle and Kant, 'Functor' from Carnap (*Logische Syntax der Sprache*), and 'natural transformation' from then current informal parlance".

[74] Girard himself has pointed this out in [Gir03, §5.3.2]: "Several systems with 'light exponentials' have been produced; my favourite being **LLL**, *light linear logic*, which has a polytime normalisation algorithm and can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural 'semantic' considerations."

[75] For instance, Elementary Linear Logic (ELL), introduced in the same paper as Light Linear Logic [Gir98] (the latter is ELL with further restrictions), is somewhat better justified: as mentioned in [DJ03], ELL can be motivated both from modal logic considerations (as heretical as this might be[76]) and from the geometry of interaction. It also rules out Russell's paradox (Footnote 43). The parsimonious λ-calculus (cf. §1.2.4) is also not just motivated by complexity, but by the study of affine approximations [Maz17] as well. Moreover, my opinion is that its typing discipline captures quite well – with a better fit than LLL or ELL – the intuition conveyed by Girard's metaphor for restricted non-linearity [Gir11b, §15.1.2]: "[…] the image of this glass of water that one can indefinitely dip from the sea. It is a modest perenniality, which does not allow one to nibble a sea from the sea – at least not indefinitely". In the parsimonious λ-calculus, we indeed have terms of type $!\tau \multimap !\tau \otimes \tau$, but not of type $!\tau \multimap !\tau \otimes !\tau$ – think of τ as the glass and $!\tau$ as the sea.

[76] While Girard harshly criticizes the modal logic **S5** for its lack of cut-elimination [Gir11b, §4.E], it turns out that this can be remediated in modern proof-theoretic frameworks, for instance deep inference [GT07]. For more on the contemporary proof theory of modal logics, see e.g. [Mar18].

1.3.1. **Monadic second-order logic.** As promised in §1.1.3, let us discuss the logical aspects of automata theory. It involves classical logic, usual first-order quantifiers, and truth-value semantics (also known as "Tarskian semantics"): a side of mathematical logic which could be seen as somewhat old-fashioned – yet no less important – compared to the proof theory of constructive logics that is related to programming languages (§1.1.4).

For our exposition of this topic, we first need to recall the usual notion of interpreting a *logical formula* (or sentence) in a structure. Consider for instance the formula $\exists x.\, \forall y.\, x \leq y$. It is meaningless to speak of its truth or falsity in absolute terms, since it is true over $\mathbb{N}$ and false over $\mathbb{Z}$. One must specify:

- a set called a *domain* over which the quantifiers $\exists x$ and $\forall y$ range;
- a binary *relation* over this domain that tells us the meaning of $\leq$ (in our example, we took the canonical ordering relations over $\mathbb{N}$ and $\mathbb{Z}$ respectively).

More generally, a set D equipped with several relations is known as a *relational structure*. By representing strings over an alphabet Σ as relational structures, a formula that can be interpreted in these structures induces a map $\Sigma^* \to \{\texttt{true}, \texttt{false}\}$, which is the same thing as what we called earlier a "decision problem" or a "language"; hence the connection with computation. The classical representation of a string $w \in \Sigma^*$ is as follows:

- the domain is the set $\{1, 2, \ldots, |w|\}$ of positions in w (where $|w|$ is its length);
- there is a binary relation $\leq$ that gives us the left-to-right ordering of positions;
- for each letter $c \in \Sigma$, we have a unary relation (i.e. subset of the domain) which is the set of positions in w that contain the letter c.

For instance, if we interpret the formula $\forall x.\, \forall y.\, (a(x) \text{ and } b(y)) \Rightarrow x \leq y$ in the structure corresponding to $w \in \{a, b\}^*$, the result will be $\texttt{true}$ if and only if $w = a \ldots ab \ldots b$.

The fundamental discovery independently made in the early 1960s by Büchi, Elgot and Trakhtenbrot [Büc60; Elg61; Tra61] is that *a language is regular if and only if it can be defined by a formula in monadic*[77] *second-order logic (MSO)* – the latter is a logic in which one can quantify not just over elements of the domain (positions in a string), but also over *subsets*. This was quickly followed by an extension of this equivalence to infinite words [Büc62] (to represent them as relational structures, use $\mathbb{N}$ as the domain). Nowadays, correspondences with MSO are a central motivation for automata models over various kinds of data, infinite or not, all of which can be encoded as relational structures.

The case of infinite words and infinite trees is especially important for the theoretical underpinnings of (automated) *formal verification* of hardware or software systems: MSO over such structures subsumes many logics used to specify system behaviors (e.g. by considering $\mathbb{N}$ as an ordered set of discrete time steps, linear-time temporal logic can be translated into MSO over $(\mathbb{N}, \leq)$ which corresponds to the aforementioned [Büc62], though it is more efficient to directly translate temporal logics to automata, see [Pin21a, Chapter 38]). This is mostly out of scope for this manuscript, but see §1.3.4 for a brief discussion of something in this spirit. Even ignoring applications to formal verification, connections with logic are pervasive in automata theory, justifying its classification as part of "logic in computer science".

- In *algebraic language theory*, there are many correspondences between algebraically defined subclasses of regular languages and subsystems of MSO, providing further evidence that those classes are natural. The most famous case is, again, star-free languages: they are exactly those definable in *first-order logic* (FO). (Although often attributed to McNaughton and Papert, this result is due to Schützenberger, as discussed in [Str18].)

[77] "Monadic" in MSO just means "unary" – full second order logic allows quantifying over relations of arbitrary arity on the domain, not just subsets. It does not have anything to do with monads from category theory (though monads and MSO can be combined in categorical automata theory [BKS21]).

- Concerning *transducers*, there are various ways to define string-to-string functions (rather than just languages) using MSO. They give rise to characterizations of rational [Fil15, §5.3], regular [EH01] and polyregular [BKL19] functions. And Schützenberger's theorem lifts to transducers as one might hope: in all three cases, replacing MSO by FO leads to a characterization of the corresponding aperiodic transduction class [FGL16; CD15; BKL19] (this explains why aperiodic regular functions are called "FO transductions").

The last thing concerning MSO whose mention is almost mandatory is that it has a celebrated application to computational complexity. To formulate it, we must first remark that a *graph* can be represented as a relational structure using its vertex set as the domain, and an adjacency relation to encode edges. Given any MSO formula over graphs, *Courcelle's theorem* says that for any $k \in \mathbb{N}$, the decision problem that this formula defines can be solved in linear time over graphs of *treewidth* at most k.[78] Fixed-parameter tractability results involving treewidth are a major theme in graph algorithms, a firmly "Volume A" (§1.1.1) topic, and Courcelle's theorem implies many such results at once. There is a lot more to say about MSO over graphs (e.g. about graph-to-graph transductions), see the book [CE12].[79]

1.3.2. **Descriptive vs implicit complexity and their legacies.** Monadic Second-Order Logic over graphs is not the only point of contact between *model theory*[80] – the area of mathematical logic that focuses on the interpretation of formulas in structures – and computational complexity. By considering a logic that is *not* included in MSO, one could hope to characterize some complexity class – rather than an automata-theoretic class of languages – as the set of decision problems (over strings) that are defined by the formulas in this logic. This is indeed the basic conceit of an entire field called *descriptive complexity*[81]. For example, Fagin's theorem [Fag73] says that a problem is in NP if and only if it can be expressed in existential second-order logic. This is considered to be the first main result of the field (coming a decade after the connection between MSO and regular languages), and many other complexity classes have been described in this way since then (see e.g. [Imm99]).

A logic used this way can be seen as a kind of programming language offering a very high level of abstraction, in which one directly[82] expresses *what* one wants to compute, rather than *how* to compute it: a *declarative* language. Thus, strictly speaking, descriptive complexity fits our previous definition of implicit computational complexity (§1.1.4): "machine-free characterizations of complexity classes via high-level programming languages" (and similarly, the MSO-based characterizations from §1.3.1 are arguably a kind of "implicit automata".)

But the common usage of those words does not consider descriptive complexity as a part of ICC. The latter is used as an umbrella term encompassing basically all approaches to "machine-independent complexity" that do not fit in the specific model-theoretic approach of descriptive complexity. The type-theoretic characterizations of complexity classes that we already saw are merely a part of ICC; the field also includes tools such as recursive function

[78] See [CE12, Chapter 6] for a modern exposition. An early statement can be found in [Cou88] with a quadratic rather than linear bound; the result was also independently rediscovered in [BPT92]. The complexity bottleneck was the tree decomposition step, concerning which many further works offered algorithmic improvements, see [CE12, §6.2].

[79] One recent development that it does not cover is the proof of an old conjecture of Courcelle concerning an automata-MSO connection for graphs of bounded treewidth [BP16a].

[80] To explain the terminology, a structure is said to be a *model* of a set of formulas when all of them are true in the structure. Traditional model theory focuses mostly on infinite structures, but the needs of computer science prompted the study of *finite* model theory.

[81] Beware: "descriptive complexity" has a different meaning in the context of descriptive set theory.

[82] Relatively speaking… Intuitively straightforward properties are sometimes expressed as unavoidably convoluted formulas in a given logic. But the point is that even then, the formula reads like a description of some property of the input, rather than an algorithm that computes and then returns a boolean at the end.

algebras or term rewriting. For example, a result published by Gurevich in 1983 [Gur83] captures logarithmic space complexity using a form of primitive recursion over finite relational structures. It can be considered part of the "prehistory" of ICC, as it is often considered that the latter only really took off starting from the early 1990s, with [GSS92; BC92; LM93]; for an overview of the past 30 years of research in the area, we recommend the survey [Péc20].

Oversimplifying a bit, since all our examples of implicit complexity techniques have a "functional programming" flavor, we could summarize what we just said like this:

programming paradigm	*declarative*	*functional*
complexity classes	*Descriptive Complexity*	*Implicit Computational Complexity*
automata theory	*subsystems of MSO*	*our work! + §1.3.3*

However, this actually gives a misleading impression of what contemporary developments in descriptive complexity look like. Since the picture for implicit complexity is, for its part, relatively faithful (though it neglects the more practical applications of ideas derived from ICC to static analysis of programs,[83] just as we did not mention the historical ties of finite model theory to databases[84]), this explains why the descriptive complexity and ICC communities are separate: they are mostly not interested in the same questions.

Descriptive complexity today is mainly concerned with what happens if one considers (finite) relational structures that are not strings as inputs, for instance graphs (cf. the end of §1.3.1). Since one can choose any reasonable encoding of graphs as strings without making any difference for usual complexity theory,[85] it is tempting to think that the choice of input structure does not matter much. This initial intuition is completely wrong.

What must be taken into account is *symmetry*: the fact that some structures may have *automorphisms*, i.e. permutations of the domain that fix all relations. This symmetry is broken when one serializes a structure as a string: for instance, for graphs, the vertices might be numbered in some arbitrary order. And the execution of a program that takes graphs as inputs can depend on this order, e.g. to loop over all vertices successively, even when the end result is invariant under automorphism (example: a depth-first search to decide whether a graph is connected). Logical formulas cannot do that sort of thing:[86] if two vertices can be

[83] For instance, techniques from ICC have been applied to compiler optimizations [Rub17]. Automated analyses of the computational complexity of functional programs are also close to ICC, with some overlap between the people working on those two topics; one manifestation of this was the (now defunct) DICE-FOPARA joint workshops. The more general area of formal methods for time complexity analysis is surveyed in [Gué19, §9.2], as part of a PhD thesis [Gué19] whose main subject is to formally prove complexity bounds using separation logic; and recall from Footnote 32 that separation logic relies on a variant of linear logic.

[84] An important real-world example of declarative programming language is SQL, the usual language for querying *relational databases*. The latter remain the most common kind of database despite some hype in the 2010s around key-value stores (the "NoSQL" movement). Their name comes from the fact that they can be seen as relational structures: think of a k-ary relation as a table with k columns, and of each k-tuple that it contains as a row. The expressivity of SQL can be characterized using an extension of first-order logic [Imm99, Chapter 14] (and the same goes for Datalog, another database query language).

[85] The reader familiar with graph algorithms might object that, for example, finding a path between two vertices in a graph with n vertices and m edges takes $O(n^2)$ time on the adjacency matrix representation but $O(n+m)$ time on the adjacency list representation. But here we mean the kind of complexity that is concerned with "robust" classes such as P, NP, … whose definition should not depend on the choice of "reasonable" model; and "reasonable" is defined by the invariance thesis (§1.1.1) which allows for polynomial overhead in time. In this example, both representations put the problem in P. Nevertheless, there is a recent and active area of research named *fine-grained complexity* that applies the philosophy of complexity theory (e.g. look for conditional lower bounds via reductions) in a way that is sensitive to some (but not all) distinctions within P made by traditional analysis of algorithms; see [Vas19].

[86] To circumvent this limitation of logic, one can add an arbitrary total order to the relational structures considered, and allow formulas that syntactically refer to this order but whose truth value is independent from it. The established keyword for this idea is "order-invariant logics", see e.g. [Gra20].

exchanged by an automorphism, then a formula necessarily treats them the same way. This is why descriptive characterizations of standard complexity classes often apply only to a class of structures that do not admit non-trivial automorphisms: *totally ordered* structures, in which one of the provided relations is a total order on the domain. This is the case for strings, with their order on positions, but not for graphs.

For example, there is no known "logic for polynomial time" over arbitrary finite relational structures. In fact, Gurevich has even conjectured that such a logic *does not exist*, in a precise sense, and this has been called "the perhaps most fundamental and challenging open problem of finite model theory" [GG15]. Since Fagin's theorem actually provides a logic for NP over all structures, the conjecture implies $\mathsf{P} \neq \mathsf{NP}$. By contrast, a logic for P has been known for a long time for totally ordered structures [Var82; Imm86]. Currently, the main candidate for a "logic for P", which would refute the conjecture since it fits Gurevich's definition,[87] is *choiceless polynomial time*, introduced in [BGS99]. It is defined using a non-declarative programming language, so nominally, it should arguably belong to implicit complexity. The fact that it is much better known among descriptive complexity theorists than in the ICC community is more evidence towards their true difference in focus today.

There is an important former candidate logic for P that looks more like the kind of "logic" usually considered in descriptive complexity (it is an extension of first-order logic, with a notion of truth-value semantics in relational structures): *fixed-point logic with counting* (FPC). It was first suggested in the 1980s [Imm86, §5], and it is still relevant today,[88] even though a decision problem in P that it cannot express was found in the same decade [CFI92]. The modern understanding of FPC is that it "can be seen as capturing a natural class of *symmetric algorithms* inside P, with equivalent formulations in circuit complexity and linear programming" – the quote (emphasis included) is taken from the survey [Daw20]. Intuitively, an algorithm is symmetric when its execution – not just its result – is preserved by automorphisms; a logician would say that this is an *intensional* property.

This recently discovered connection with circuits (see [Wil19]) shows that logic is not required to define this notion of *symmetric computation*; it is a convincing argument for considering the latter as part of complexity theory properly speaking. (On top of that, the idea shares some commonalities with monotone complexity,[89] a topic which definitely belongs to "Volume A".) That descriptive complexity, by investigating its internal questions, has been able to come up with such a natural and interesting concept in complexity theory is a remarkable achievement.

It is unmatched by implicit complexity as far as we know, but the thing that comes closest, perhaps, is that programming language theory and denotational semantics provide tools to study the computational complexity of higher-order functions,[90] though this is not

[87] Therefore, it is widely believed that choiceless polynomial time cannot compute all functions in P, but this remains an open problem. For recent progress on this question, see [Pag21].

[88] A notable recent achievement is the very long proof, published as a book [Gro17], that FPC captures polynomial time on every minor-closed class of graphs; it involves deep combinatorics related to the graph isomorphism problem. See also Sandra Kiefer's PhD thesis [Kie20a] (or her introductory paper [Kie20b]) for more recent developments in this vein; its focus is on the Weisfeiler–Leman algorithm for graph isomorphism, which is closely related to FPC over graphs (as already remarked in [CFI92]).

[89] Extensional: a function $\{0,1\}^* \to \{0,1\}$ is monotone when flipping some input bits from 0 to 1 cannot decrease the output from 1 to 0. Intensional: an algorithm is monotone when it "does not use negations". For more information, we refer to the introduction of [DO18] (a paper concerned with ICC for monotone complexity). Let us also cite [Kup21] as an example of work on a very similar theme in automata theory.

[90] In particular, game semantics (previously mentioned in §1.1.8) are the basis of two recent works of this kind: a proposal for a time complexity measure on higher-order functions [Fér17], and an exploratory work on notions of higher-order cryptographic primitives [BCL20] (the theoretical foundations of cryptography depend on complexity theory to bound the computational power of adversaries).

part of ICC *per se*. Non-type-theoretic forms of ICC have also served to characterize the *basic feasible functionals*, a canonical counterpart to P for functions $(\Gamma^* \to \Sigma^*) \to \Pi^*$, called "second-order" or "type-2" functions, cf. [Péc20, §4.2].[91]

And yet, a limitation of both descriptive and implicit complexity (pointed out for instance in [Maz17, §4.2.1]) is that they have provided no major ideas for the main questions concerning the most basic complexity classes such as P or NP, especially when it comes to separations between them. To be fair, two major results (namely $\mathsf{NL} = \mathsf{coNL}$ and PARITY $\notin \mathsf{AC}^0$) of classical complexity theory were discovered through descriptive complexity; but in both cases, they were also independently proved around the same time in other ways, and the essence of the descriptive proofs turned out not to depend on logic on a fundamental level (cf. [Aar17, §3]). Concerning the proofs-as-programs approach to ICC, the somewhat related idea that separation results in complexity could reflect foundational issues concerning "infinity as reuse" best expressed in linear logic (see e.g. [Gir03, §5] and [Gir12][93]) has stayed at the level of speculation.[94] ICC results often teach us about the programming languages involved much more than about the complexity classes that they capture – but the former are sometimes contrived for the sake of the result, as we say in §1.2.5.

On the contrary, as we argued in §1.2, we feel that our work on implicit automata has provided us with a better understanding of transducers,[95] by uncovering some categorical structure. Throughout this introduction, we have indeed attributed our successes to the deeply algebraic nature of modern automata theory. Conversely, one might wonder whether the limitations of ICC until now are due to a lack of compositional methods for fundamental complexity theory.[96] If the toolset of programming language theorists has any role to play

[91] As a cultural remark, let us mention that computability for such functions sometimes goes under the name "type-2 theory of effectivity"; it is part of the computability theory of higher-order functions, for which the standard reference is [LN15]. To get a taste of the field, Martín Escardó's very accessible blog post *Seemingly impossible functional programs*[92] explains a spectacular phenomenon. A major application of type-2 effectivity is to study computability and complexity for functions between separable topological spaces, see the books [Wei00; BH21]. For instance one can define what it means for the integration map $\int : \mathcal{C}([0,1], \mathbb{R}) \to \mathbb{R}$ to be computable, by representing arbitrary real numbers (resp. continuous functions) by convergent sequences (that need not be computable!) in $\mathbb{Q}^{\mathbb{N}}$ (resp. $\mathbb{Q}[X]^{\mathbb{N}}$). See also [Ste17] for complexity-theoretic aspects of computable analysis; finally, we should mention that an implicit characterization of the polynomial time functions $\mathbb{R} \to \mathbb{R}$ has been given using a parsimonious λ-calculus in [HMP20].

[92] Available at `https://archive.md/rUGpG`. The ideas behind this have been unexpectedly applied to prove that in intuitionistic set theory (cf. Footnote 22), the Cantor–Schröder–Bernstein theorem is equivalent to the law of excluded middle [PB19] (see also [Esc21] for an extension to homotopy type theory).

[93] Girard's paper [Gir12] sketches a characterization of non-deterministic logarithmic space using von Neumann algebras. It actually led to quite a few subsequent works, which however abandoned the original foundational and philosophical ambitions that initially motivated [Gir12]: Aubert and Seiller clarified and extended Girard's technical ideas [AS16a; AS16b], and with Bagnol and Pistone, they also adapted their techniques to more down-to-earth settings to characterize various complexity classes [AB18; Aub+14; ABS16]. Finally, Seiller obtained similar results [Sei18] in the setting of his interaction graphings semantics [Sei19]. (Furthermore, those interaction graphings and the aforementioned [Gir12] share the same main inspiration: a version of the geometry of interaction using von Neumann algebras [Gir11a].)

[94] It must be said, however, that the work mentioned above in Footnote 93 has led Seiller to consider graphings as a mathematical abstraction of programs whose structure might help to prove separation results in complexity theory. As a first demonstration of this idea, Pellissier and Seiller [PS21] have rephrased various proofs of complexity lower bounds in a unified way within the framework of graphings, reformulating the key arguments of those proofs using the notion of topological entropy. Thus, [Gir12] ended up indirectly inspiring, through a long and winding path, a promising approach to separation results – even though no trace of logic remains in this approach.

[95] There might be a personal bias here: I embarked on this project not knowing much about automata theory, and my grasp of the field increased alongside the discovery of new results.

[96] There are exceptions in some sub-areas, such as *constraint satisfaction problems* [Bul21]: the recent proofs of the CSP *Dichotomy Conjecture* [Zhu20; Bul20] both use an algebraic approach to leverage symmetries.

in the future of computational complexity, it is perhaps in providing those missing methods (or serving as their inspiration), rather than giving yet another implicit characterization of, say, polynomial time.[97] For an example of very speculative exploration towards this direction, see [Maz17, §4.2.3]. Finally, let us note that the "categorification" of descriptive complexity and finite model theory has been a very active research topic recently; the journal paper [AS21][98] contains references to most of the work on this at the time of writing.

1.3.3. **Previous work on implicit automata and adjacent topics.** After these broad discussions of subfields of TCS, let us focus more narrowly on direct precursors of our work. If we consider non-type-theoretic approaches to ICC (as defined towards the beginning of the previous subsection), then there are several recent formalisms for regular string functions using combinators [AFR14; DGK18; BDK18] that arguably belong to "implicit automata". Recent results of this kind include characterizations of first-order transductions on strings [DGK21] and on trees [BD20].

We also mentioned earlier (§1.2.4) that Bojańczyk gave four different characterizations of polyregular functions in [Boj18]. One of them is a successor to the aforementioned [BDK18]. Interestingly, while the functional language of [BDK18] did not support λ-abstraction, its extension in [Boj18] uses a simply typed λ-calculus enriched with a ground type of lists and several primitive functions on lists. Strings are represented as lists of characters, which differs from our use of functional encodings in a λ-calculus without any primitive data type.

We will now restrict our attention to works that involve typed λ-calculi.

Automata as circular proofs. Aside from Hillebrand and Kanellakis's Theorem 1.1.4 and the above-mentioned characterization of polyregular functions, perhaps the only previous works that completely fit the theme of "implicit automata via proofs-as-programs" are those by DeYoung and Pfenning [DP16] on sequential transducers and by Kuperberg, Pinault and Pous [KPP19] (see also [Pin21b, Chapter 3]) characterizing regular languages and deterministic logarithmic space complexity. Both rely on a proofs-as-programs interpretation of *circular proof systems* for some variants of linear logic with fixed points. Circular proofs allow some amount of controlled self-referentiality to handle recursive definitions; they are sometimes called "cyclic" proofs, but in our context, this would create a confusion with an unrelated non-commutative logic, *cyclic linear logic* [Yet90].

The Church encoding of strings is obtained by a systematic procedure [BB85] from the inductive definition $s ::= \varepsilon \mid c_1 \cdot s \mid \ldots \mid c_{|\Sigma|} \cdot s$ ($\Sigma = \{c_1, \ldots, c_{|\Sigma|}\}$). Using fixed points of formulas, one can instead turn it into the recursive type[99] $\mathtt{Str}^{\mu}_{\Sigma} = 1 \oplus \mathtt{Str}^{\mu}_{\Sigma} \oplus \ldots \oplus \mathtt{Str}^{\mu}_{\Sigma}$; this is the definition of the type of strings in [DP16], and it is also implicit in[100] [KPP19].

---

[97] But it would be unfair to reduce current developments in ICC to this: several research perspectives such as probabilistic and quantum computation are outlined in [Péc20, Chapter 5], and there are other recent works covering new ground such as parallel complexity bounds for process algebras [Ghy21].

[98] The introduction to [AS21] mentions a "structure vs power" dichotomy further discussed in [Abr20] which partially correlates with the "machines vs programs" and "Volume A/B" distinctions from §1.1.1. Interpreted literally, "structure" is what we call algebra/compositionality while "power" refers to expressiveness; but the actual intended meaning is that "structure" covers programming language theory and semantics, while "power" contains basically the rest of theoretical computer science (including complexity, automata and finite model theory). I would slightly object to this terminology: in my opinion, automata theory is a field in which algebra can be fruitfully applied to study expressiveness today (this is one of the points I tried to make in Section 1.1.3). In any case, the three aforementioned dichotomies agree on the gap between computational complexity and programming languages and the disjointness of their research communities.

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

[100] The left rules given in [KPP19, Figure 1] for A and A^* correspond respectively to $A = 1 \oplus \ldots \oplus 1$ and $A^* = 1 \oplus (A \otimes A^*)$.

So both our approach and those using fixed point logics morally work because the consumption of strings represented as inductive data types is similar to their traversal by automata. However, while the use of the "right fold" provided by a Church-encoded string involves an "inversion of control" (in programming jargon) that, in the case of the simply typed λ-calculus, has drastic effects on expressive power (contrast Theorem 1.1.4 with the much larger computational power of the simply typed λ-calculus in other contexts, cf. §1.3.6), circular proofs seem to give the programmer more degrees of freedom: Kuperberg et al. do not need to add polymorphism to go beyond regular languages. In fact it was shown later in [Pin21b, Chapter 4][101] (which is refined in [Das20a]) that much higher complexities (above primitive recursion) can be reached using other monomorphic circular proof systems.[102]

Automata characterizations with a semantic flavor. Using a computational model inspired by[103] denotational semantics of linear logic, Seiller [Sei18] has given a characterization of each level of the k-head two-way non-deterministic automata hierarchy. The lowest level ($k = 1$) corresponds to regular languages, while the union over $k \in \mathbb{N}_{\geq 1}$ gives the complexity class NL (non-deterministic logarithmic space). Something in common with our work is that the representation of strings used by [Sei18] is more or less a semantic version of Church encodings (see [Sei18, §3.2]). There is one main difference with what one usually calls implicit complexity: Seiller's result does not take place inside a syntactically defined programming language (and it is far from obvious how to turn this model into a similarly expressive syntax, because of the previously mentioned inversion of control).[104]

Terui's investigations [Ter11] on a form of game semantics for (polarized[105]) linear logic called ludics [Gir03] also include a characterization of regular languages.

Recognizable languages of λ-terms. We have emphasized in §1.1.7 the use of evaluation in a finite denotational semantics used to prove Theorem 1.1.4. Along these lines, general notions of recognizable languages of closed λ-terms of a given type (specializing to regular languages for the type of Church-encoded strings) have been proposed, based on finite semantics, in the simply-typed λ-calculus by Salvati [Sal09] and in an infinitary λ-calculus by Melliès [Mel17a]. It is plausible that Theorem 1.1.4 can be extended to give an equivalent syntactic definition for Salvati's recognizable languages: for a simple type τ they would be the languages definable by $\tau[\sigma] \to \texttt{Bool}$. An interesting question would be whether one can give an encoding of *higher-dimensional trees* in the simply typed λ-calculus so that this notion of recognizability coincides with Rogers's automata for those trees [Rog03; GK07].

---

[101] Following what we said in §1.3.2, "descriptive" should be replaced by "implicit" in the title of [Pin21b]!

[102] The theory of circular proofs is also connected to automata over infinite structures for reasons that are orthogonal to these "implicit automata" results: circular proofs are subject to "correctness criteria" that correspond to acceptance conditions for such automata. For instance the aforementioned [Das20a] relies on a study of circular arithmetic [Das20b] which in turns applies results on the reverse mathematics of MSO over $(\mathbb{N}, \leq)$ [Koł+19] (that were already mentioned in Footnote 14). We will not attempt to survey the growing field of circular proofs here; let us just mention one application that has garnered significant attention: the solution of an old problem concerning automata and formal verification [Dou17, Part II].

[103] This inspiration has already been discussed at length in Footnote 93.

[104] A personal digression: my first steps in implicit complexity took place during an internship with Thomas Seiller whose initial subject was to design a syntactic counterpart to this result. In retrospect there are major obstructions to doing so, but a discussion with Damiano Mazza during this internship led us to rediscover Theorem 1.1.4, thus planting the first seeds for the "implicit automata in typed λ-calculi" research programme! I first applied Hillebrand and Kanellakis's ideas to explain a puzzling phenomenon in the elementary affine λ-calculus (§1.4.7), and the work on implicit automata properly began one year later after Mikołaj Bojańczyk went around at ETAPS'19 suggesting to a few people (including Pierre Pradic, who relayed this to me) that there could be some connection between polyregular functions and linear logic.

[105] See Footnote 56 and [Lau02]. The polarities of linear logic are also closely related to the *focusing* used in Chapter 5; both were first discovered in the context of proof search for linear logic [And92].

As for Melliès's notion of recognizability for simply typed infinitary λ-terms [Mel17a], it is mainly inspired from his work with Grellois (compiled in the latter's PhD dissertation [Gre16]) on *higher-order model checking*. Let us explain that topic now.

1.3.4. **Higher-order recursion schemes.** Let $\mathbf{\Sigma}$ be a ranked alphabet and $\mathtt{Tree}_{\mathbf{\Sigma}}$ be the type of Church-encoded trees over $\mathbf{\Sigma}$. The simply typed λ-terms of type $\mathtt{Tree}_{\mathbf{\Sigma}}$ correspond to finite trees. Now, consider the $\lambda\mathbf{Y}$*-calculus*, which extends the simply typed λ-calculus with general recursion (concretely, it adds a fixed point operator $\mathbf{Y}$). Because the recursion may be non-terminating, some $\lambda\mathbf{Y}$-terms $t : \mathtt{Tree}_{\mathbf{\Sigma}}$ represent *infinite* trees[106] (though only countably many infinite trees over $\mathbf{\Sigma}$ – so not all of them – can be generated this way).

One can then ask, given t and a formula φ of MSO (§1.3.1) over infinite trees, whether the tree defined by t satisfies φ. This is the *higher-order model checking* problem, whose decidability was first proved in [Ong06] using game semantics, and then reproved several times with various different tools and stronger conclusions (e.g. in [Bro+21; Par21]). The point is that t may represent the possible behaviors of some code written in a functional programming language. It is shown in [Kob13, §3] that several examples of formal verification tasks on functional programs reduce to higher-order model checking, and this has led to actual implementations of verification tools for subsets of the OCaml language, cf. [Kob19].

But the idea is much older than that, going back to the 1970s. Initially, it was formulated in terms of tree-generating grammars called *(higher-order) recursion schemes*;[107] for a survey of the early history of recursion schemes, we refer the reader to [Mir06, §2.6]. This point of view is equivalent to the one we started our exposition with: an infinite tree can be generated by a recursion scheme if and only if it can be defined by a $\lambda\mathbf{Y}$-term (see e.g. [SW16, §3]).

A third equivalent way to define infinite trees is using an automaton model (with no input and a tree output): *collapsible pushdown automata* (see e.g. [Pin21a, Chapter 35]). They were designed by adding a "collapse" operation to *higher-order pushdown automata* (HOPDA) – automata with stacks of stacks … of stacks (with a bounded nesting) – which were already known to correspond to the recursion schemes enjoying a certain *safety* property [KNU02]. Completing this picture, there is also a "safe $\lambda\mathbf{Y}$-calculus" [SW16, §4.4] whose terms of type $\mathtt{Tree}_{\mathbf{\Sigma}}$ correspond to trees over $\mathbf{\Sigma}$ generated by HOPDA. Its type system imposes constraints on the allowed applications and λ-abstractions depending on the *(functionality) order* of simple types – a notion that will also appear in §1.3.6, defined as

$$\mathrm{ord}(\mathbb{o}) = 0 \qquad \mathrm{ord}(\sigma \to \tau) = \max(\mathrm{ord}(\sigma) + 1, \mathrm{ord}(\tau))$$

Going back to the aforementioned work of Grellois and Melliès, it is also related to another major theme of this dissertation, viz. linear logic: one of the results presented in [Gre16] is a decision procedure for higher-order model checking that uses evaluation in a finite semantics of the $\lambda\mathbf{Y}$-calculus derived from denotational models of linear logic (in fact, a variant of the semantics used by [Ter12], cf. §1.1.7). The complexity of model checking for variants of the $\lambda\mathbf{Y}$-calculus with linear types has also been studied recently [CGM17; CM19].

[106] This is similar to how infinite lists can be defined recursively in Haskell thanks to lazy evaluation. For example, different ways to code the list of *all* prime numbers in increasing order are given in [ONe09].

[107] One should be aware of a naming collision here: what Haskell programmers call "recursion schemes" are higher-order functions that abstract common patterns for recursive function definitions. For example, the "fold" function on lists captures such a pattern $(f([x_0, x_1, \dots]) = g(x_0, f([x_1, \dots])))$; its generalizations to other algebraic data types are called "catamorphisms" (and the principle of Church encodings is to represent a datum by a catamorphism over it!). (Another terminological clash: in the title of the founding paper on this kind of recursion scheme [MFP91], the word "lens" refers to a piece of notation, which differs from the current usage of "lens" in the Haskell community – see Footnote 58 for the latter.)

1.3.5. **Simply typed λ-definability over Church encodings.** We did not mention all those notions – recursion schemes, pushdown automata and the safety restriction – merely because they are part of a topic involving both automata and λ-calculi. They have an actual direct relevance to our work in progress on the simply typed λ-calculus which will be discussed in §1.4.1. To give some context for this work, we recall here some classical facts.

It is well-known that there exists a simply typed λ-term $\mathtt{exp} : \mathtt{Nat}[\mathbb{o} \to \mathbb{o}] \to \mathtt{Nat}$ – where $\mathtt{Nat}$ is the type of the Church encoding of natural numbers, or equivalently, of the Church encoding of unary strings – that codes the function $n \mapsto 2^n$. Since the composition of a term of type $\mathtt{Nat}[\sigma] \to \mathtt{Nat}$ and another of type $\mathtt{Nat}[\tau] \to \mathtt{Nat}$ can be given the type $\mathtt{Nat}[\sigma[\tau]] \to \mathtt{Nat}$, towers of exponentials of any fixed height h can be expressed by a term of type $\mathtt{Nat}[\tau_h] \to \mathtt{Nat}$ (where the type τ_h becomes increasingly complicated as $h \to +\infty$).

Remark 1.3.1. However, the heterogeneity of its input and output types prevents $\mathtt{exp}$ from being iterated by a Church numeral (recall that the Church encoding of $n \in \mathbb{N}$ is morally the iterator $f \mapsto f \circ \ldots (n \text{ times}) \cdots \circ f$). Iterating $n \mapsto 2^n$ would indeed give rise to a tower of exponentials of variable height, which is known to be inexpressible by any $\mathtt{Nat}[\tau] \to \mathtt{Nat}$.

By contrast, without type substitutions:

Theorem 1.3.2 (Schwichtenberg [Sch75])**.** *The functions $\mathbb{N}^k \to \mathbb{N}$ defined by closed simply typed λ-terms of type $\mathtt{Nat} \to \cdots \to \mathtt{Nat}$ are exactly the* extended (multivariate) polynomials*, generated from 0, 1, $+$, $\times$ and a conditional $\mathtt{if}\ n = 0\ \mathtt{then}\ p\ \mathtt{else}\ q$.*

The above theorem can be suitably generalized to strings [Zai87] and trees [Zai91; Lei93] (both as inputs and outputs). However, no analogous result is known for $\mathtt{Nat}[\tau] \to \mathtt{Nat}$:

Question 1.3.3. Characterize (without reference to the λ-calculus) the functions $\mathbb{N} \to \mathbb{N}$ definable by closed simply typed λ-terms of type $\mathtt{Nat}[\tau] \to \mathtt{Nat}$ for some type τ.

There are well-known negative results (attributed to Statman in the discussion that follows [FLO83, Theorem 4.4.3]) suggesting at first sight that there might be no satisfying answer (though, of course, it will be our goal in §1.4.1 to argue that this pessimism is unwarranted): while towers of exponentials are expressible, many simple functions of tame growth are not, for example $n \mapsto \lfloor \sqrt{n} \rfloor$. If we look at functions of two variables, there is a striking example: *subtraction* cannot be defined by any simply typed λ-term of type $\mathtt{Nat}[\sigma] \to (\mathtt{Nat}[\tau] \to \mathtt{Nat})$, no matter what types σ, τ are chosen.

When it comes to *predicates*, however, the situation is better understood: a subset of $\mathbb{N}^k$ can be defined by a simply typed λ-term of type $\mathtt{Nat}[\tau_1] \to \cdots \to \mathtt{Nat}[\tau_k] \to \mathtt{Bool}$ if and only if it is *ultimately periodic*. This has been shown by Joly[108] [Jol01, Proposition 5], and the case $k = 1$ is a special case of Theorem 1.1.4 since unary regular languages correspond to ultimately periodic subsets of $\mathbb{N}$ via $\mathbb{N} \cong \{a\}^*$. The λ-undefinability of equality and of subtraction follow immediately from this characterization.

1.3.6. **Remarks on complexity in the simply typed λ-calculus and light logics.** We wrap up this "related themes" section with something that has no relation to automata. As our last clarification concerning the state of the art on the computational power of the simply typed λ-calculus, we wish to drive home the following slogan: it is a "calculus of elementary complexity" in the same way that Light Linear Logic is often said to be a "logic of

[108] The same paper [Jol01] contains a result that explains the parenthetical clause in Question 1.3.3: a characterization of the functions defined by terms of type $\mathtt{Nat}[\tau] \to \mathtt{Nat}$ (which applies more generally to arbitrary free algebras, not just $\mathbb{N}$). It is formulated using untyped λ-terms subject to a kind of complexity constraint in an unrealistic (by Joly's own admission) cost model.

polynomial time".[109] In fact the closest comparison would be with LLL's cousin, Elementary Linear Logic,[110] which is also considered to be a "logic of ELEMENTARY".

Of course, we have already mentioned Hillebrand et al.'s use of simply typed λ-terms to characterize ELEMENTARY in §1.1.4, so what we are saying is just that it realizes the full potential of this type system in terms of expressive power. But some readers might wonder: an old result of Statman [Sta79] says that normalization in the simply typed λ-calculus is non-elementary; is that not a sign that it should be able to express non-elementary functions, and that we are still being bridled by our input representations, just as we were with Church encodings and Theorem 1.1.4? The truth is that Hillebrand et al.'s result *implies* Statman's theorem. To see why, suppose for the sake of contradiction that there were a normalization algorithm with elementary complexity. Then its time complexity would be a tower of exponentials of height $h \in \mathbb{N}$. This would entail algorithms in time "tower of height h" for all functions that are definable according to the specification of [HKM96]: take the λ-term defining the function, apply it to the representation of the input, and normalize. But the time hierarchy theorem implies that there are functions in ELEMENTARY whose complexity is higher than this tower of fixed height!

In fact, one key to the result of [HKM96] is that any simply typed λ-term t has an associated parameter $f(t) \in \mathbb{N}$ such that the complexity of normalizing t is a tower of exponentials whose height depends only on $f(t)$. This parameter $f(t)$ is the maximum *order* of a subterm of t, where the order of a term is the order of its type (§1.3.4). This is also the parameter that is controlled in order to capture each level of the k-EXPTIME / k-EXPSPACE hierarchy in [HK96] (cf. §1.1.7). And the results of [Ter12] (also mentioned in §1.1.7) state that, for any $o \in \mathbb{N}$, the normalization problem for simply typed λ-terms of type `Bool` whose subterms have order at most o is complete for one of the classes in the same k-EXPTIME / k-EXPSPACE hierarchy.

To substantiate our comparison with ELL and LLL, note that they also have a parameter – the box depth – which is used to bound the time complexity of normalization.[111] Moreover, a unification is realized by Linear Logic by Levels (L^3) [BM10], another system which characterizes ELEMENTARY: it is an extension of ELL, admits a translation from the simply typed λ-calculus [GRV09], and both ELL box depth and simply typed order are mapped to L^3 "levels".[112] (This observation on unified phenomena does not seem to have been explicitly written in [BM10; GRV09].)

A final remark: the aforementioned implicit complexity results of [HKM96; HK96] use encodings of relational structures (cf. Sections 1.3.1 and 1.3.2) as inputs. Further connections with database theory are discussed in [HKM96].

1.4. Work in progress

We now outline several ideas for further research. Those are mostly in the form of concrete mathematical statements, of two kinds: "Claims" are assertions for which we already have plausible proof strategies, and believe that only routine details remain to be worked out and properly written up; "Conjectures" are more challenging open problems.

[109] This usual motto for LLL is subject to some subtleties, see [MM02]. The "moral of the story" concerning input encodings could also apply to ELL and to the simply typed λ-calculus.

[110] First mentioned in Footnote 75.

[111] It controls the height of a tower of exponentials for ELL and the degree of a polynomial for LLL.

[112] More precisely, the paper [GRV09] gives a translation of the multiplicative-exponential fragment of linear logic (meLL) into L^3, which sends the !-depth of meLL formulas to the level in L^3. We compose this with the standard call-by-name translation of the simply typed λ-calculus into meLL ($\mathcal{T}(\sigma \to \tau) = !\mathcal{T}(\sigma) \multimap \mathcal{T}(\tau)$) which maps the order to the !-depth.

1.4.1. **Tree transducers in the simply typed (or safe) λ-calculus.** Let us continue discussing Question 1.3.3 on $\mathtt{Nat}[\tau] \to \mathtt{Nat}$ right where we left off in Section 1.3.5. We have seen that the classical limitations of simply typed λ-definable functions $\mathbb{N} \to \mathbb{N}$ on Church encodings can be explained by a special case of Hillebrand and Kanellakis's Theorem 1.1.4. This suggests that to gain more insight into the question, we should generalize it to $\mathtt{Str}_\Gamma \to \mathtt{Str}_\Sigma$. In fact, we will even consider Church-encoded ranked trees. From the obvious extension of Theorem 1.1.4 to trees, a simple necessary condition can be deduced:

Corollary 1.4.1. *Any function* $\mathbf{Tree}(\mathbf{\Gamma}) \to \mathbf{Tree}(\mathbf{\Sigma})$ *definable by a simply typed* λ*-term of type* $\mathtt{Tree}_{\mathbf{\Gamma}}[\tau] \to \mathtt{Tree}_{\mathbf{\Sigma}}$ *preserves regular tree languages by preimage (cf. Definition 1.1.3).*

This suggests looking into classes of regularity-preserving tree transductions, especially ones with hyperexponential growth since towers of exponentials can be defined by terms of type $\mathtt{Nat}[\tau] \to \mathtt{Nat}$. There is a function class $\mathcal{E}$[113] that fits these requirements and seems particularly robust. Its equivalent definitions include:

- two machine models whose equivalence is proved in [EV88]:
  - high level tree transducers;
  - iterated pushdown tree transducers;
- three characterizations of the form "the composition closure of the functions computed by the transducer model $\mathcal{T}$", shown to be equivalent in [EM03a]:
  - $\mathcal{T}$ = macro tree transducers;[114]
  - $\mathcal{T}$ = tree-walking transducers;
  - $\mathcal{T}$ = pebble tree transducers.

(The equivalence between iterated pushdown tree transducers and the composition closure of macro tree transducers is one of the results of [EV86] which is the earliest paper that we know of to consider this class $\mathcal{E}$.) The restriction of $\mathcal{E}$ to unary alphabets (which is relevant to our Question 1.3.3 on functions $\mathbb{N} \to \mathbb{N}$), called the *$k$-computable sequences*, has been studied in detail by Fratani and Sénizergues [FS06], who show that they generalize some integer sequences of interest (e.g. polynomial recurrence equations). Sénizergues has also claimed in an invited paper without proofs[115] [Sén07] that the restriction of $\mathcal{E}$ to strings – which he calls the "k-computable mappings" – can be characterized as the composition closure of HDT0L transductions (cf. §1.2.4).

It is not hard to translate, say, macro tree transducers into simply typed λ-terms. In fact the range of the translation is included in the *safe* λ-calculus[116] [BO09], which is to the simply typed λ-calculus what the safe $\lambda\mathbf{Y}$-calculus of [SW16] is to the $\lambda\mathbf{Y}$-calculus (cf. §1.3.4). Since the "safely λ-definable tree functions" are closed under composition, they contain the class $\mathcal{E}$. We have strong reasons to believe that the converse also holds:

Claim 1.4.2. $\mathcal{E}$ is the class of functions definable by some safe λ-term $t : \mathtt{Tree}_{\mathbf{\Gamma}}[\tau] \to \mathtt{Tree}_{\mathbf{\Sigma}}$.

The conceptual reason for the claim requires some background on higher-order model checking (which was recalled in §1.3.4). Iterated pushdown transducers are very close to HOPDA, while high level transducers are presented in [EV88] as safe recursion schemes

[113] I am not aware of a standard name for this class, so I am calling it $\mathcal{E}$ here in honor of Joost Engelfriet who is the common coauthor of [EV88; EV86; EM03a].

[114] Already mentioned in Footnote 66, and discussed further at the start of Chapter 6.

[115] Some of the claims in [Sén07] were proved in subsequent papers [FMS14; Cad+21], but as far as we know, it is not yet the case for those concerning compositions of HDT0L transductions.

[116] One result of [BO09] characterizes the functions definable by safe λ-terms of type $\mathtt{Str}_\Gamma \to \mathtt{Str}_\Sigma$, that is, without type substitution in the input. In the case of natural numbers, the functions $\mathbb{N}^k \to \mathbb{N}$ definable in this way are exactly the multivariate polynomials in $\mathbb{N}[X_1, \ldots, X_k]$; compare with Theorem 1.3.2.

where the application of rules is controlled by[117] an input tree. We therefore believe that we should be able to translate safe λ-terms defining tree functions to high level tree transducers by adapting the usual translation from safe $\lambda\mathbf{Y}$-terms to safe recursion schemes.[118]

The analogy with higher-order model checking then suggests the following "unsafe" counterpart of the above claim: tree functions definable in the simply typed λ-calculus using Church encodings should correspond to high level tree transducers without the safety restriction. We expect this to be routine, but a more interesting result would be to prove the unsafe version of the equivalence from [EV88]; combining the two would give:

Conjecture 1.4.3. There exists a tree transducer counterpart of collapsible pushdown automata (whose stack is initialized with the input tree) that can compute exactly the functions definable by *simply typed λ-terms* of type $\mathtt{Tree}_{\mathbf{\Gamma}}[\tau] \to \mathtt{Tree}_{\mathbf{\Sigma}}$ for some τ.

Note that this would provide an answer to Question 1.3.3. If this works out, the next step would be to seek alternative characterizations of this function class – typically, as the composition closure of a nice basis of simple enough transducers – and to understand whether the inclusion of $\mathcal{E}$ in it is strict.

More anecdotally, while the functions defined in the simply typed λ-calculus without type substitution (i.e. by some $t : \mathtt{Str}_\Gamma \to \mathtt{Str}_\Sigma$) are already well-understood (again, see [Zai87]), we make the following observation which is new as far as we know:

Claim 1.4.4. A language can be defined by a simply typed λ-term of type $\mathtt{Str}_\Sigma \to \mathtt{Bool}$ if and only if it can be expressed as a boolean combination of languages of the form $\varphi^{-1}(P)$ for $\varphi \in \mathrm{Hom}(\Sigma^*, \mathbb{U}_2)$ and $P \subseteq \mathbb{U}_2$ where[119] $\mathbb{U}_2 = \{1, a, b\}$ with $1x = x$, $ax = a$ and $bx = b$.

For a fixed Σ, there are finitely many such languages. The argument for "only if" is semantic evaluation in Scott domains, with $[\![\mathbb{o}]\!] = \{\bot, \top\}$ where $\bot < \top$ so that $[\![\mathbb{o} \to \mathbb{o}]\!] \cong \mathbb{U}_2$.

Note that all our results on the simply typed λ-calculus also apply to the $\lambda\ell^{\oplus\&}$-calculus where we use the simple type of Church-encoded strings

$$\mathtt{Str}^!_\Sigma = (\mathbb{o} \to \mathbb{o}) \to \cdots \to (\mathbb{o} \to \mathbb{o}) \to \mathbb{o} \to \mathbb{o}$$

rather than its linearized variant $\mathtt{Str}_\Sigma$, and do not impose any constraint on the type substitution in the input. We propose the following speculation on something that sits between those hyperexponential λ-definable functions and the regular functions in Theorem 1.2.3:

Conjecture 1.4.5. A function can be defined by a $\lambda\ell^{\oplus\&}$-term of type $\mathtt{Str}^!_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$ for some purely linear τ if and only if it is an HDT0L transduction.

The mismatch between the input and output types would correspond to the fact that HDT0L transductions are not closed under composition. As for $\mathtt{Str}^!_\Gamma[\tau] \to \mathtt{Str}_\Sigma$ with τ purely linear, we have no idea as to what that would correspond to.

1.4.2. **Affine types without additives, FO transductions & tree-walking automata.** As we said towards the end of §1.2.2, a natural question is whether all regular functions are still expressible if we remove the additive connectives $\oplus/\&$ from the $\lambda\ell^{\oplus\&}$-calculus. We also explained that to compensate, the linearity condition should be relaxed to *affineness*.

[117] In fact, each rule application "consumes" an input node, so the recursion scheme cannot be unfolded indefinitely. This is why such transducers map finite input trees to finite output trees, whereas trees generated by recursion schemes can be infinite.

[118] The recursion schemes used in [EV88] actually obey a slightly stronger restriction than safety coming from [Dam82] and called "Damm-safety" in modern terminology, cf. [Par18]. (The Damm-safe λ-calculus arises naturally from iterating the "clone" operation in universal algebra, see [Sal15, §2.3].) But safe schemes and Damm-safe schemes have the same expressive power (a direct translation is given in [Par18, §4]).

[119] $\mathbb{U}_2$ is called the *flip-flop* monoid. It appears in the algebraic version of Krohn–Rhodes (Theorem 2.2.10).

Claim 1.4.6. We have the following implicit characterizations of transduction classes in λ-calculi without additives (where τ is a *purely affine* type that may be chosen depending on the string-to-string functions that we want to express):

	$\mathtt{Str}_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$	$\mathtt{Str}_\Gamma[\tau] \to \mathtt{Str}_\Sigma$
affine commutative	regular functions	comparison-free polyregular functions
affine non-commutative	FO transductions	first-order cfp functions[120]

In other words, our claim is that we get the same characterizations of string transductions as in the $\lambda\ell^{\oplus\&}$-calculus, and that we also capture the corresponding first-order/aperiodic function classes in by adding non-commutativity. By contrast, Theorem 1.2.1 does not hold if additives are added to the $\lambda\ell\wp$-calculus, as we will see at the start of Chapter 7.

In [NP20, §4], we showed how to use the Krohn–Rhodes decomposition to code all sequential functions (our proof of Theorem 1.2.1 here involves something very similar) as affine λ-terms of type $\mathtt{Str}_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$, and we also showed how to express all *aperiodic* sequential functions in an affine non-commutative λ-calculus. To extend this to regular functions, one possible strategy is to use the following decomposition theorem:

Theorem 1.4.7 (Bojańczyk et al.[121])**.** *The composition closure of the class composed of*

- *sequential functions (resp. aperiodic sequential functions)*
- *and* $\mathtt{mapReverse}_\Sigma, \mathtt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$*, defined as*

$$\forall w_1, \dots, w_n \in \Sigma^*, \quad \begin{aligned} \mathtt{mapReverse}_\Sigma(w_1\# \dots \#w_n) &= \mathtt{rev}(w_1)\# \cdots \#\mathtt{rev}(w_n) \\ \mathtt{mapDuplicate}_\Sigma(w_1\# \dots \#w_n) &= w_1w_1\# \cdots \#w_nw_n \end{aligned}$$

is exactly the class of regular functions (resp. first-order transductions).

As for functions taking trees as input, we claim that the removal of additive connectives hurts expressivity. This could be suspected for tree transductions given the fact that the additive conjunction corresponds to an important feature of BRTTs (as we said in §1.2.2), but we have a stronger argument that also applies to *languages*:

Claim 1.4.8. Any tree language defined by an affine λ-term of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\tau] \multimap \mathtt{Bool}$ without additive connectives can be recognized by some *tree-walking automaton*.

Tree-walking automata are a natural extension to trees of two-way automata, that can recognize a strict subclass of regular tree languages, which does not even contain all the first-order tree languages (the difficult proof of the separation result can be found in [BC08]). We do not have a proof of the converse, nor do we have strong reasons to believe that it holds for now, so the following is an open-ended problem:

Question 1.4.9. What is the class of tree languages definable by such λ-terms?

One thing we can already claim, however, is that the output languages of the affine λ-terms of type $\mathtt{Tree}_{\mathbf{\Gamma}}[\tau] \multimap \mathtt{Str}_\Sigma$ are the same as those of tree-walking tree-to-string transducers. This should follow from the results of [Sal06].

To explain our argument for Claim 1.4.8, we need to talk about *geometry of interaction* and its relationship with tree-walking automata.

[120] While this notion has not been defined properly in this dissertation, its definition is simple: replace regular functions by first-order transductions in the definition of comparison-free polyregular functions that uses "composition by substitution" (cf. Footnote 71). This definition was suggested in [NNP21, §10] as part of a model-theoretic conjecture that has since then been refuted (see the start of Chapter 3).

[121] The non-aperiodic case is a special case of a published result [BS20, Theorem 18] on automata over nominal sets (cf. §1.4.4). The aperiodic case is not entirely explicit in the literature, but it can be recovered from the results of [BDK18] (mentioned in §1.3.3).

1.4.3. **Geometry of interaction, categorical tree automata and planar transducers.** Recall from Section 1.1.9 that the central idea of a paper of Hines [Hin03] can be reformulated as follows: automata over the category $\mathsf{Int}(\mathsf{PFSet})$ are the same thing as two-way automata (2DFA). It turns out that *tree* automata over the same category correspond to none other than tree-walking automata. Furthermore, 2DFTs and tree-walking tree-to-string transducers can be rephrased as (tree) automata over a certain category $\mathsf{Int}(\mathsf{PFSet}_\Sigma)$.[122] Finally, *reversible* 2DFA (resp. 2DFT) [Dar+17] are automata over $\mathsf{Int}(\mathsf{PFInj})$ (resp. $\mathsf{Int}(\mathsf{PFInj}_\Sigma)$) where PFInj is the category of partial *injections* between finite sets.

The Int-construction is also a general way to build denotational models of the purely linear λ-calculus [Abr96]. Those models belong to a family of works grouped under the umbrella of "geometry of interaction" (GoI), sharing a common origin in Girard's attempts to give a sort of "dynamic semantics" for linear logic that would reflect the process of computation while being more mathematically structured[123] than usual operational semantics [Gir89b; Gir89a]. The task of summarizing the wide variety of GoI-related lines of work[124] (such as context semantics (cf. [Mai02]), token machines (see [CVV21] for a recent example), etc.) and their diverging points of view is daunting, and we will not attempt it here.

The point is that this allows us to prove Claim 1.4.8 for *linear* λ-terms using a semantic evaluation argument. However this does not work immediately in the *affine* case. Fortunately, there are known ways to overcome this technical obstacle. One of them is to leverage the close connection between GoI and the history-free game semantics of the purely affine λ-calculus; this is done for instance in [CM19] for an affine version of the $\lambda\mathbf{Y}$-calculus (cf. §1.3.4).

It is also worth noting that the purely linear non-commutative λ-calculus admits a GoI-style semantics in a category of planar[125] diagrams [Abr07], with "planarity" defined as the lack of crossings in a standard graphical representation of PFSet-morphisms (§1.1.9). This representation coincides with the drawings for 2DFA behaviors that often appear in the literature (e.g. [Boj18, p. 45], see also Figure 1.1.1 in this manuscript). Hines [Hin06] has suggested looking at the corresponding "two-way automata with planar behaviors" but he did not have any result on their expressive power. In an upcoming paper, we will show:

Claim 1.4.10 ([NP21])**.** The languages (resp. functions) recognized by planar two-way automata (resp. transducers) are exactly the star-free languages (resp. FO transductions). Moreover, adding a reversibility restriction keeps their expressive power intact.

These are the obvious conjectures given Theorem 1.2.1 and Claim 1.4.6; we establish along the way that monoids of planar behaviors are aperiodic, which gives us an alternative semantic proof for one direction of Theorem 1.2.1.[126] A semantic evaluation argument could perhaps to applied to the bottom row of Claim 1.4.6 as well by devising a corresponding "planar game semantics" for the purely affine non-commutative λ-calculus (also called the "$\lambda\wp$-calculus" in [NP20]); Aurore Alcolei, Pierre Pradic and I had begun to think about this but we have not had the time to delve seriously into it.

[122] $\mathsf{Obj}(\mathsf{PFSet}_\Sigma)$ consists of finite sets, $\mathsf{Hom}_{\mathsf{PFSet}_\Sigma}(X, Y) = (Y \times \Sigma^*)^X$, and composition of morphisms concatenates the "annotations" in Σ^*.

[123] About this (ab)use of the word "geometry" to conjure this connotation, see Footnote 42.

[124] While Girard's first results [Gir89a] took place in operator algebras (from functional analysis), most later works transferred the insights gained from [Gir89a] to much more mathematically elementary settings, involving mainly finite combinatorics. In fact a folklore observation is that the operators on a separable Hilbert space H that appear in [Gir89a] are just partial injections $\mathbb{N} \rightharpoonup \mathbb{N}$ on a fixed basis of H. But Girard later returned to more sophisticated operator-algebraic GoI semantics [Gir11a], cf. Footnote 93.

[125] The relationship between planarity and non-commutativity was first discussed in Footnote 33.

[126] Our proof of the former is also more conceptually satisfying than the brute-force syntactic argument we gave for the latter in [NP20]: we characterize Green's relations $\leq_{\mathcal{L}}$ and $\leq_{\mathcal{R}}$ in the monoid of all planar behaviors over a given finite set of states, from which we quickly deduce its $\mathcal{H}$-triviality i.e. aperiodicity.

We conclude this subsection with two remarks on the equivalence between 2DFTs and copyless SSTs in the light of the geometry of interaction. The first one is a technical claim: there is a functor[127] $\mathsf{PFSet}_\Sigma \to (\mathcal{SR}^{\succcurlyeq})_\oplus$ where $\mathcal{SR}^{\succcurlyeq}$ is the counterpart for strings of the category $\mathcal{TR}^{\succcurlyeq}$ from Chapter 6, which induces a translation from 2DFTs to a machine model which resembles single-use-restricted BRTTs but for strings. (Indeed, $(\mathcal{SR}^{\succcurlyeq})_\oplus$ and $(\mathcal{SR}_{\&})_\oplus$ are quite close.) Thus we can give a categorical account of one direction of the equivalence.

The second remark is more high-level: morally, 2DFTs and copyless SSTs make different space-time tradeoffs (no registers vs single-pass processing). This should be compared with the effectiveness of GoI techniques for space-bounded evaluation of λ-terms,[128] as demonstrated for instance in implicit characterizations of logarithmic space complexity based on linear logic [Sch07; DS16; Maz15].

1.4.4. **Automata and transducers over infinite alphabets (nominal sets).** One extension that we can consider to our characterization of regular string functions is to work over an infinite alphabet. Following [CLP15], let us work with alphabets of the form $\Sigma \times \mathbb{A}$ where Σ is finite and $\mathbb{A}$ is a countable infinite set of "atoms". A string in $(\Sigma \times \mathbb{A})^*$ will then be represented by a term of type

$$\mathtt{Str}_\Sigma^{\mathbb{A}} = \underbrace{(\mathtt{Atom} \to (\mathbb{o} \multimap \mathbb{o})) \to \cdots \to (\mathtt{Atom} \to (\mathbb{o} \multimap \mathbb{o}))}_{|\Sigma| \text{ times}} \to \mathbb{o} \to \mathbb{o}$$

in some λ-calculus with both unrestricted and affine function arrows, as well as a primitive type $\mathtt{Atom}$ and, for each $a \in \mathbb{A}$, a corresponding constant $\underline{a} : \mathtt{Atom}$.

The next question is: what operations will be allowed on the type $\mathtt{Atom}$ within our chosen λ-calculus? A simple possibility is to have as the only primitive operation an equality test[129] $\mathtt{eq}_\tau : \mathtt{Atom} \multimap \mathtt{Atom} \multimap (\tau \,\&\, \tau) \multimap \tau$: the only kind of information on atoms that we can inspect is whether two given atoms are equal. Then definable languages in $\mathcal{P}((\Sigma \times \mathbb{A})^*)$ are invariant under the natural action of the permutation group $\mathfrak{S}(\mathbb{A})$ of $\mathbb{A}$, and definable functions $(\Gamma \times \mathbb{A})^* \to (\Sigma \times \mathbb{A})^*$ "commute" with this action.

There is a whole theory of *nominal sets* concerned with certain sets endowed with an action of $\mathfrak{S}(\mathbb{A})$. Interestingly, it comes from programming language theory. The initial motivation was to handle syntax with variable renaming (see e.g. [Pit16])[130] so the idea was not to add atoms to a λ-calculus as we do above, but to manipulate variables in usual λ-terms as permutable atoms. Later on, there has been a lot of work on computability and automata over nominal sets, see [Boj19].

As we said in Section 1.1.5, the notion of copylessness[131] has recently been demonstrated to be important for automata over nominal sets [BS20]: copyless register automata (with finitely many atom-valued registers) recognize a robust class of languages, and copyless SSTs with atoms have good properties such as a decomposition theorem in the style of Krohn–Rhodes and Theorem 1.4.7. Thus one could expect that working in a linear or affine

[127] In fact this functor arises from the fact that PFSet_Σ can be built from an $(\mathcal{SR}^{\succcurlyeq})_\oplus$-*enriched category* through the change of base induced by $\mathrm{Hom}(\mathbf{I}, -) : (\mathcal{SR}^{\succcurlyeq})_\oplus \to \mathsf{Set}$. This encapsulates in a precise technical way the idea that 2DFT behaviors are composed of a finite state plus finitely many labels; the precise choice of category $(\mathcal{SR}^{\succcurlyeq})_\oplus$ reflects the linearity properties of composition of morphisms with respect to those labels.

[128] It was also conjectured for a long time that a GoI token machine for the untyped λ-calculus could satisfy the invariance thesis (Footnote 3) for *space*, but it now seems that this is probably wrong [ADV21].

[129] We do not take the obvious choice $\mathtt{eq} : \mathtt{Atom} \multimap \mathtt{Atom} \multimap \mathbf{I} \oplus \mathbf{I}$ because we want to restrict to *negative connectives* in order to make syntactic manipulations easier – see the discussion at the start of Chapter 5.

[130] Pitts and Gabbay were given the 2019 Alonzo Church Award for Outstanding Contributions to Logic and Computation for introducing nominal sets, see https://archive.md/DLWbA.

[131] The paper [BS20] uses "single use" to mean "copyless"; this should not be confused with the more permissive "single use restrictions" found in [EM99; AD17].

λ-calculus, we could extend our implicit characterization of regular functions to infinite alphabets. Clovis Eberhart, Pierre Pradic and I have indeed convinced ourselves that:

Claim 1.4.11. The languages (resp. functions) definable by affine λ-terms (with additives) of type $\mathtt{Str}_\Sigma^{\mathbb{A}}[\tau] \multimap \mathtt{Bool}$ (resp. $\mathtt{Str}_\Gamma^{\mathbb{A}}[\tau] \multimap \mathtt{Str}_\Sigma^{\mathbb{A}}$) for some purely affine τ (where $\mathtt{Atom}$ counts as affine) correspond to those studied in [BS20].

When we want to generalize to a broader class of alphabets treated in [BS20]: the "polynomial orbit-finite sets", which have the form $\mathbb{A}^{n_1} + \cdots + \mathbb{A}^{n_k}$ (we recover the case $\Sigma \times \mathbb{A}$ for $n_1 = \cdots = n_k = 1$ and $k = |\Sigma|$), the situation becomes less satisfying. While the Church-like encoding $\mathtt{Str}_\Sigma^{\mathbb{A}}$ admits a straightforward generalization, we have only managed to get a characterization where the input and output types do not quite match; specializing it to the case of functions $(\Gamma \times \mathbb{A})^* \to (\Sigma \times \mathbb{A})^*$, we would characterize those definable by copyless SSTs with atoms by terms of type

$$\mathtt{Str}_\Gamma^{\mathbb{A}} \multimap \Big(\underbrace{(\mathtt{Atom} \multimap (\mathbb{o} \multimap \mathbb{o})) \to \cdots \to (\mathtt{Atom} \multimap (\mathbb{o} \multimap \mathbb{o}))}_{|\Sigma| \text{ times, note the use of } \multimap \text{ instead of } \to} \to \mathbb{o} \to \mathbb{o} \Big)$$

This is a consequence of the details of the copylessness policy in SSTs with atoms, and it might be inevitable.

An open question is whether Claim 1.4.11 can be generalized to trees. For instance, one of the equivalent characterizations of recognizability by copyless register automata involves the "rigid MSO" logic from [CLP15]. Since this logic does not depend much on the relational structure, and trees with atoms also admit a Church-like encoding, we can ask:

Question 1.4.12. For tree languages, is definability in rigid MSO equivalent to expressibility by an affine λ-term of type $\mathtt{Tree}_\Sigma^{\mathbb{A}}[\tau] \multimap \mathtt{Bool}$ for some purely linear τ?

Before tackling this, it might seem prudent to first develop the theory of copyless tree automata and tree transducers with atoms, which does not exist yet at the time of writing. But if we keep in mind that copylessness is too restrictive to capture regular tree functions (§1.2), we can expect the extension of register automata to trees to require more permissive notions of single use restriction. And there is a new difficulty in designing such a notion, compared to usual SSTs over finite (ranked) alphabets: while the contents of registers in SSTs without atoms does not influence the control flow (in other words, they are just potential output pieces that are shuffled around blindly), the whole point of atom-valued registers is to be subjected to equality tests that select conditional branches. This is incompatible with the idea of the $\oplus$-completion (§1.2.3) that adds a layer of finite-state control on top of a base category that cannot influence this control. So maybe it could be better to start from λ-terms and only later try to devise equivalent automata models.

Let us also mention that the above-mentioned issue with the $\oplus$-completion has so far prevented us from finding a suitable denotational model[132] of the purely affine λ-calculus with atoms which would yield Claim 1.4.11 by semantic evaluation. Our current proof idea for this claim uses somewhat ad-hoc syntactic methods.

1.4.5. **Polyregular functions in a parsimonious λ-calculus.** One of the ICC results capturing logarithmic space that were mentioned at the end of §1.4.3 takes place in Mazza's parsimonious λ-calculus [Maz15] and uses Church-encoded strings as input. The type system introduced in [Maz15] is monomorphic, so according to the general principle outlined in Section 1.2.1, one would expect to get the regular languages instead. What allows the

[132] But the category of nominal sets, without any "single use" or "orbit-finiteness" condition (the languages studied in [BS20] are recognized by orbit-finite monoids), is a semantics of the simply typed λ-calculus, and even of higher-order intuitionistic logic, called the *Schanuel topos* [Pit16, §3].

parsimonious λ-calculus to escape this limitation is that unlike the calculi considered in this dissertation, it does not admit a forgetful functor to the simply typed λ-calculus, because of a non-standard feature[133] that it supports. (Another contrasting example is that Light Linear Logic admits a forgetful functor to usual linear logic.)

The most salient property of the parsimonious λ-calculus, a kind of restriction on non-linearity that Mazza calls "parsimony", is orthogonal to the above-mentioned non-standard feature. Therefore, if we remove the latter, the resulting system might still be interesting; let us call it the "fully uniform"[134] parsimonious λ-calculus. Its definable languages are indeed the regular languages, with the consequence that the definable string-to-string functions preserve regular languages by preimage. Thus, those functions probably correspond to some transduction class. At this point, let us remind the reader of two facts from §1.2.4:

- parsimony is very similar to layering in SSTs (which led to our reinvention of the latter);
- the class of polyregular functions is the composition closure of layered SSTs.

Thanks to this, we can easily translate layered SSTs into fully uniform parsimonious λ-terms and compose those terms to express any polyregular function. The converse seems plausible but it appears to be a difficult problem, which Damiano Mazza, Gabriele Vanoni and I tried unsuccessfully to attack with GoI-like token machines (cf. §1.4.3, see also [ALV21]).

Conjecture 1.4.13. A function $f : \Gamma^* \to \Sigma^*$ is polyregular if and only if it can be expressed by a fully uniform parsimonious λ-term of type $\mathtt{Str}_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$ for some arbitrary type τ.

Note that we do not impose any linearity condition on τ in the above statement. In the $\lambda\ell^{\oplus\&}$-calculus, if we did not have such a restriction, then we would have access to the full power of the simply typed λ-calculus since linearity only applies when the linear function arrow $\multimap$ is used. However, in the parsimonious λ-calculus, all non-linear functions must obey the parsimony condition, which is imposed globally by the type system.

1.4.6. **Maximality of (poly)regular functions.** Perhaps the right way to see the unproven part ("if") of the above conjecture is not as a property specific to the parsimonious λ-calculus, but as a consequence of something much stronger:

Conjecture 1.4.14. If a string function is definable in the simply typed λ-calculus (in the sense of Conjecture 1.4.3) and has polynomial growth, then it is polyregular.

(The connection with Conjecture 1.4.13 is that since the "parsimoniously λ-definable" functions are computable in logarithmic space, they have polynomial growth.) Our main reason for believing this conjecture is that the polyregular functions are the largest "seemingly canonical" class of polynomial growth transductions known today. Similarly, among the existing well-studied string transduction classes with linear growth, regular functions are the largest one. In fact, concerning regular functions, the following property is known:

Theorem 1.4.15 ([EIM21])**.** *Every string function in $\mathcal{E}$ (cf. Section 1.4.1) with linear growth is regular.*

[133] A first-class type $!\tau$ of "ultimately constant streams of elements of type τ" thanks to which $n \in \mathbb{N}$ can be represented as the stream $[\mathtt{true}, \dots, \mathtt{true}, \mathtt{false}, \mathtt{false}, \dots] : !\mathtt{Bool}$ with n times $\mathtt{true}$. In usual linear λ-calculi where the *exponential modality* '!' appears, $!\tau$ is the type of "duplicable data of type τ", so an inhabitant of type $!\tau$ is morally not too different from "infinitely many copies of some inhabitant of τ" or in other words a constant stream. (Such λ-calculi would encode $\sigma \to \tau$ as $!\sigma \multimap \tau$, cf. Footnote 112.)

[134] The Taylor expansion (Footnote 46) of parsimonious λ-terms with non-constant streams (Footnote 133) is "non-uniform" in the sense of [ER08]. However "non-uniform parsimonious λ-calculus" refers to another system which is "even less uniform" in this sense: streams do not even need to be *ultimately* constant. It was introduced in [MT15] and the actual reason for its name is that it serves to characterize the circuit complexity class $\mathsf{L/poly}$ which is considered the "non-uniform" version of logarithmic space in complexity theory. There are two notions of uniformity involved here, which are morally related but not identical.

It is similar to the above conjecture, especially if Claim 1.4.2 is taken into account. It also entails, as a special case, that polyregular functions with linear growth are regular – this non-trivial fact can also be derived from the MSO dimension minimization theorem of our upcoming paper[135] [Boj+21], and the comparison-free case will be proved in Chapter 3. This means that our conjecture would imply that string functions definable in the simply typed λ-calculus with linear growth are regular.

1.4.7. **Automata and complexity in the polymorphic elementary affine λ-calculus.** Coming back to typed λ-calculi, my first application[136] of the ideas from Hillebrand and Kanellakis's Theorem 1.1.4 was to solve a minor open problem from "standard" implicit complexity, concerning a characterization of polynomial time by Baillot et al.[137] [BDR18] that makes use of Church encodings. It takes place in a variant of Elementary Linear Logic which they call the *elementary affine λ-calculus*; we will refer to it as $\mu\mathrm{EA}\lambda$ here, following my paper [Ngu19]. The μ stands for a fixed point operator on types, which means that $\mu\mathrm{EA}\lambda$ supports recursive types.[138] The question was whether this feature was necessary for the result of [BDR18]; it turns out that it is, for similar reasons to the necessity of something non-standard in the parsimonious characterization of logarithmic space (see §1.4.5).[139] We call $\mathrm{EA}\lambda$ the system defined by removing recursive types from $\mu\mathrm{EA}\lambda$, again following [Ngu19]. In the statement below, the shape of the first item corresponds to the result on polynomial time in $\mu\mathrm{EA}\lambda$.

Theorem 1.4.16 ([Ngu19])**.** *Let $L \subseteq \Sigma^*$. The following are equivalent:*

- *L is definable by an $\mathrm{EA}\lambda$-term of type $!\mathtt{Str}_\Sigma^{\mathrm{EA}\lambda} \multimap !!\mathtt{Bool}^{\mathrm{EA}\lambda}$.*
- *L is definable by an $\mathrm{EA}\lambda$-term of type $\mathtt{Str}_\Sigma^{\mathrm{EA}\lambda} \multimap !\mathtt{Bool}^{\mathrm{EA}\lambda}$.*
- *L is regular.*

Here '!' denotes an "exponential modality"[140] which plays a key role in controlling complexity in variants of Elementary Linear Logic. The type $\mathtt{Str}_\Sigma^{\mathrm{EA}\lambda}$ differs from our usual linearized Church encoding in two ways. First, it uses '!' in a way that is compatible with a certain "stratification" property of $\mathrm{EA}\lambda$. Second, it incorporates a quantification over types which allows us to state the above without any type substitution in the input; the type system of $\mathrm{EA}\lambda$ indeed supports *polymorphism*. The above theorem is proved by combining the syntactic stratification property with a semantic evaluation argument. The latter relies on the following fact (compare with our proof sketch for Theorem 1.1.4):

Claim 1.4.17. There exists a finite denotational semantics of the purely affine polymorphic λ-calculus which is non-trivial in the sense that $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$.

Such a semantics has also been sketched by Laird [Lai13], as a stepping stone towards another game semantics for an extension of the usual (non-linear) polymorphic λ-calculus.[141] The finiteness property is explicitly noted at the beginning of [Lai13, §7] (without proof, but it is not hard to check).

---

[135] Joint work with Mikołaj Bojańczyk, Gaëtan Douéneau-Tabot, Sandra Kiefer and Pierre Pradic.

[136] Previously alluded to in Footnote 104.

[137] Reformulating the earlier [Bai15] that took place in a slightly less user-friendly system.

[138] We already saw recursive types / fixed points appear in §1.3.3 in relation with circular proofs. But there is a difference: $\mu\mathrm{EA}\lambda$ supports arbitrary recursive types, whereas logics with fixed points that admit circular proof systems generally have positivity restrictions on recursion (one can define $\tau = \tau \oplus \sigma$ but not $\tau = \tau \multimap \sigma$ because the left of a function arrow is a "negative" position).

[139] Once again, this is evidence for our point that connections with automata sometimes arise more "naturally" than with complexity when studying typed λ-calculi.

[140] See also Footnote 133.

[141] Also known as System F and previously mentioned in Footnote 37.

Baillot et al. [BDR18] also give a characterization of *string-to-string functions* computable in polynomial time in $\mu\mathrm{EA}\lambda$, which I simplified in [Ngu19] to the type $!\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda} \multimap !\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda}$. (The nice thing is that it reflects the fact that polynomial time functions are closed under composition.) The obvious question is: what happens without recursive types? The situation, we claim, is analogous to what happens in the $\lambda\ell^{\oplus\&}$-calculus:

Claim 1.4.18. A function $\Gamma^* \to \Sigma^*$ is *regular* (resp. *comparison-free polyregular*) if it can be defined by an EAλ-term of type $\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda} \multimap \mathtt{Str}_\Sigma^{\mathrm{EA}\lambda}$ (resp. $!\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda} \multimap !\mathtt{Str}_\Sigma^{\mathrm{EA}\lambda}$).

(We can even formulate for $k \in \mathbb{N}$ a characterization of cfp functions *of rank* k, i.e. using $k+1$ pebbles.[142]) The proof idea is to extend Laird's game semantics by annotating each move in a strategy with a string in Σ^*. This should give a category admitting a functor to the category $(\mathcal{SR}^{\asymp})_\oplus$ mentioned in §1.4.3.[143]

I also discovered that the properties stated in Claim 1.4.17 are also fulfilled by a semantics based on *coherence spaces* (combining the treatment of affineness in [Mel09, §8.1] and that of polymorphism in [GTL89, Appendix A]). This work has not yet been written up in a satisfactory way due to foundational issues,[144] but it has been applied to sub-polynomial ICC in a prematurely published[146] joint work with Pradic [NP19]. Since EAλ only recognizes regular languages over Church-encoded strings, we took the idea to take relational structures as inputs instead from [HKM96; HK96] (cf. §1.3.6). We gathered some evidence suggesting that this gives a characterization of deterministic logarithmic space, though the upper

[142] They are those definable by EAλ-terms of type $!\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda} \multimap !\mathtt{Str}_\Sigma^{\mathrm{EA}\lambda}$ that "use their argument $k+1$ times", i.e. that are obtained by promotion and contraction from a term of type

$$\underbrace{\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda} \multimap \ldots \multimap \mathtt{Str}_\Gamma^{\mathrm{EA}\lambda}}_{k+1 \text{ times}} \multimap \mathtt{Str}_\Sigma^{\mathrm{EA}\lambda}$$

What makes this work is that the type $\mathtt{Str}_\Gamma^{\mathrm{EA}\lambda}$ is similar to a call-by-value translation of the non-linear Church encoding, this choice being imposed by the stratification property of EAλ. The $\lambda\ell^{\oplus\&}$-calculus cannot do this because the typing rules for its non-linear function arrow '$\to$' correspond to a call-by-name translation (see Footnote 112 and Footnote 133).

[143] Just as for $\mathsf{Int}(\mathsf{PFSet}_\Sigma)$, we expect this functor to be induced by an underlying enriched category structure over $(\mathcal{SR}^{\asymp})_\oplus$ (see Footnote 127).

[144] For instance there is no precise definition of a categorical semantics of the purely affine (or linear) polymorphic λ-calculus in the literature. Such a definition, and the accompanying soundness result for its interpretation of λ-terms, is expected to be routine, but this work should still be done as a matter of principle![145] Then there is the question of whether the above-mentioned semantics really do satisfy the categorical axioms. In the case of the coherence space semantics of the polymorphic λ-calculus, it is known that Girard's original paper [Gir86] contains a major gap: it does not check that a certain uniformity condition required on morphisms is closed under composition, which is non-trivial (I first rediscovered this for myself, and then learned from Thomas Ehrhard that this was folklore). In general I tend to believe that categorical axiomatics is an important hygienic precaution for anything that pretends to be a denotational semantics; for instance, the axioms for the exponentials of linear logic sometimes fail surprisingly, see [Mel09, §8.7] for a discussion of a historical example and [Rib20, §1.3] for a more recent one. Another important example of a "semantics" that isn't an actual semantics is the so-called "Blass problem" [Abr03].

[145] More generally, the "obviousness" of this kind of metatheoretic result relating syntax and semantics has been heavily disputed in the dependent type theory community, particularly by Voevodsky, as discussed in the introduction to [Boe20].

[146] Its reliance on an unfinished preprint is not the only issue with rigor in this paper; I take full responsibility for this (it really was my fault, not Pierre's!). I will not dwell on the details of those mistakes of youth here, but my present thoughts on the general phenomenon of "premature publishing" are well reflected by the following quote from Katrin Wehrheim's *MIT Women in Mathematics* profile (https://archive.md/xwYVr): "'We don't write good papers anymore', she says. Many proofs are published with gaps or unnecessarily complicated logic, as if the author didn't have the time or patience to explain the idea. She likes to use a mountain metaphor: 'It's like if some really well trained climber made it to the mountain top. But they didn't leave any hooks along the way, so someone with less training will have no way of following it without having to find the route for themselves.'"

bound that we manage to "prove" in [NP19] is weaker (but still strictly below P assuming widely believed complexity-theoretic conjectures). Unfortunately, the GoI approach used for previous ICC results [Sch07; DS16; Maz15] on logarithmic space using linear types (mentioned at the end of §1.4.3) does not appear to apply to our setting; instead, we have been exploring an approach using hypercoherences [Ehr93], but its success might require some difficult algorithmics.[147]

1.5. Chapter-by-chapter outline

The last role that this introduction needs to fulfill is to explain the plan of the manuscript. The order of chapters is different from the order of exposition of results in this introduction, since it follows logical dependencies. Each chapter starts with a plan detailing its division into sections. Chapters 3 and 7 are based on the publications [NNP21] and [NP20] respectively, while Chapter 2 draws from both of them; those papers were published in the proceedings of the *International Colloquium on Automata, Languages and Programming (ICALP)*. The rest has not yet been subject to peer review.[148]

The notations used throughout the dissertation are recapitulated in Section 2.1. In the rest of Chapter 2, we recall all the preexisting material on automata and transducers that will be useful in subsequent chapters, and we prove a few results that did not appear before in the literature[149] (such as the one concerning layered SSTs mentioned in §1.2.4).

The next two chapters contain almost no trace of λ-terms. In Chapter 3 we introduce and study the class of comparison-free polyregular functions (§1.2.4) from a purely automata-theoretic perspective. This is followed by Chapter 4, which is dedicated to the constructions and results in categorical automata theory announced in §1.2.3.

The remainder of the dissertation is concerned with "implicit automata". Chapter 5 defines the $\lambda\ell^{\oplus\&}$-calculus and uses the aforementioned category-theoretic material to prove the part of Theorem 1.2.3 that concerns string-to-string functions. In Chapter 6, we extend some results of Chapter 4 to trees in order to finish proving Theorem 1.2.3. Finally, the main subject of Chapter 7 is Theorem 1.2.1 on star-free languages in the $\lambda\ell\wp$-calculus, but it also explains why linear (as opposed to affine) λ-calculi without additives are not suitable to characterize transduction classes.

[147] See the slides https://nguyentito.eu/2021-01-cgcafe.pdf.

[148] Some of it comes from a submission to a journal that was desk-rejected for excessive length.

[149] Excluding our own paper [NNP21] from which most of this material is lifted.

CHAPTER 2

Preliminaries: notations and automata models

This chapter starts with some notations used throughout the entire dissertation, which we put in Section 2.1 for ease of lookup. (Since categories does not appear until Chapter 4 (except for the informal overview in Chapter 1), we have delayed the introduction of their dedicated notation to Section 4.1.1.) After that, it covers many of the function classes and transducer models that we will encounter:

- Sequential transducers (§2.2) are a very simple machine model whose purpose is mostly illustrative… until Chapter 7 where their Krohn–Rhodes decomposition (§2.2.1) will play an important technical role.
- Streaming string transducers (SSTs), introduced in Section 2.3, are more important since their copyless version (§2.3.2) define regular functions, one of the classes that we will later characterize using the $\lambda\ell^{\oplus\&}$-calculus. We introduce some technical tools to work with SSTs in Section 2.3.4, where a special case of the wreath product construction (also used in Krohn–Rhodes) makes an appearance.
- Copyful (§2.3.1) and layered (§2.3.3) SSTs also admit alternative characterizations in terms of HDT0L systems, which we present in Section 2.4.
- Section 2.5 introduces polyregular functions.
- Finally, in Section 2.6, we start with a high-level discussion of how to extend copyless SSTs in order to handle trees, then recall a machine model for regular tree functions.

Almost none of this is an original contribution; the definition of "layered HDT0L system" and the associated Theorem 2.4.5 might be an exception, though it is neither hard nor surprising. We also give some proofs that we could not find in the literature, but which might be folklore.

2.1. Notations & elementary definitions

2.1.1. **Sets.** We follow the convention that natural numbers start at 0, that is, $\mathbb{N} = \{0, 1, \dots\}$. The cardinality of a set X is written $|X|$. We sometimes consider a family $(x_i)_{i\in I}$ as a map $i \mapsto x_i$, which amounts to treating $\prod_{i\in I} X_i$ as a dependent product. Consistently with this, we make use of the dependent sum operation

$$\sum_{i\in I} X_i = \{(i, x) \mid i \in I,\, x \in X_i\}$$

Recall also that the binary coproduct of sets is the tagged union

$$X + Y = \{\mathsf{in}_1(x) \mid x \in X\} \cup \{\mathsf{in}_2(y) \mid y \in Y\}$$

The injection $\mathsf{in}_1/\mathsf{in}_2$ may be omitted by abuse of notation when it is clear from the context, that is, for $x \in X$, we allow ourselves to write x for $\mathsf{in}_1(x)$ when it is understood that this refers to an element of $X + Y$.

2.1.2. **Strings (a.k.a. words).** Alphabets designate finite sets and are written using the variable names Σ, Γ, Π. The set of strings over an alphabet Σ is denoted by Σ^*. We write $|w|$ for the length of a word $w \in \Sigma^*$ and either w_i or $w[i]$ for its i-th letter ($i \in \{1, \dots, |w|\}$); given a letter $c \in \Sigma$, the notation $|w|_c$ refers to the number of occurrences of c in w.

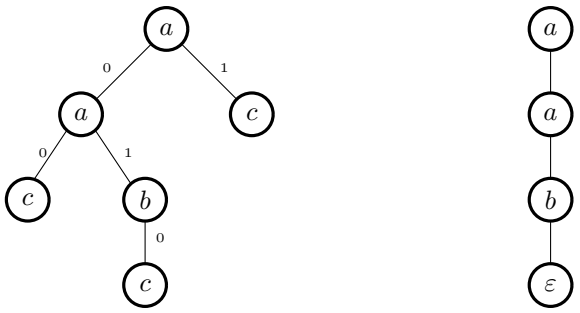


FIGURE 2.1.1. Graphical representations of the tree $a(a(c, b(c)), c)$ over the ranked alphabet $\{a : \{0,1\},\ b : \{0\},\ c : \varnothing\}$ (left) and of the word $aab \in \{a, b\}^*$ seen as an element of $\mathbf{Tree}(\overline{\{a, b\}}) = \mathbf{Tree}(\{a : \{*\},\ b : \{*\},\ \varepsilon : \varnothing\})$ (right).

The concatenation of two strings $u, v \in \Sigma^*$ is written uv (or sometimes $u \cdot v$ for clarity); recall that Σ^* endowed with this operation is the free monoid over the set of generators Σ, and its identity element is the empty string ε. Given monoids M and N, $\mathrm{Hom}(M, N)$ is the set of monoid morphisms.

Finally, we write $\underline{\Sigma} = \{\underline{a} \mid a \in \Sigma\}$ for a disjoint copy of the alphabet Σ made of "underlined" letters.

2.1.3. **Ranked trees.** Ranked alphabets are pairs (Σ, ar) such that Σ is an alphabet and the *arity* ar is a family of finite sets[1] indexed by Σ; they are written using $\mathbf{\Sigma}, \Gamma$. We may write $\{a_1 : A_1,\ \ldots,\ a_n : A_n\}$ for the ranked alphabet $(\{a_1, \ldots, a_n\}, \mathrm{ar})$ with $\mathrm{ar}(a_i) = A_i$.

Given a ranked alphabet $\mathbf{\Sigma}$, the set $\mathbf{Tree}(\mathbf{\Sigma})$ of trees/terms over a ranked alphabet $\mathbf{\Sigma}$ is defined as usual: if a is a letter of arity X in $\mathbf{\Sigma}$ and t a family of $\mathbf{\Sigma}$-trees, we write $a(t)$ for the corresponding tree. Examples of such trees are pictured in Figure 2.1.1.

Remark 2.1.1. Given an alphabet Σ, define $\overline{\Sigma}$ to be the ranked alphabet $(\Sigma + \{\varepsilon\}, \mathrm{ar})$ such that $\mathrm{ar}(\mathsf{in}_1(a)) = \{*\}$ and $\mathrm{ar}(\mathsf{in}_2(\varepsilon)) = \varnothing$. This gives a isomorphism $\mathbf{Tree}(\overline{\Sigma}) \simeq \Sigma^*$, illustrated on the right of Figure 2.1.1.

2.2. SEQUENTIAL TRANSDUCERS

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. A possible reference is [Sak09, Chapter V].

Definition 2.2.1. 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 $w = w[1] \ldots w[n] \in \Sigma^*$, the *run* of the transducer over w is the sequence of states $q_0 = q_I$, $q_1 = \delta_{\mathrm{st}}(q_0, w[1])$, ..., $q_n = \delta_{\mathrm{st}}(q_{n-1}, w[n])$. Its *output* is obtained as the concatenation $\delta_{\mathrm{out}}(q_0, w[1]) \cdot \ldots \cdot \delta_{\mathrm{out}}(q_{n-1}, w[n]) \cdot F(q_n)$.

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

[1] This is slightly non-standard; the more usual notion would be that ar be only a family of numbers $\Sigma \to \mathbb{N}$. Our choice will be more convenient to describe some constructions in Chapter 6.

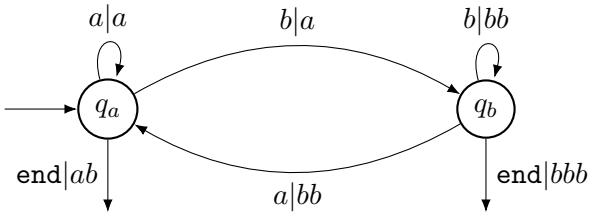


FIGURE 2.2.1. A schematic representation of a sequential transducer whose formal definition is $Q = \{q_a, q_b\}$, $\delta(q, a) = (q_a, a)$ and $\delta(q, b) = (q_b, bb)$ for $q \in Q$, $q_I = q_a$, $F(q_a) = ab$ and $F(q_b) = bbb$.

Definition 2.2.2. The *transition monoid* of a sequential transducer is the submonoid of $Q \to Q$ (endowed with *reverse* function composition: $fg = g \circ f$) generated by the maps $\{\delta_{\mathrm{st}}(-, c) \mid c \in \Sigma\}$ (where $\delta_{\mathrm{st}}(-, c)$ stands for $q \mapsto \delta_{\mathrm{st}}(q, c)$).

A sequential transducer is said to be *aperiodic* when its transition monoid is aperiodic (cf. Definition 1.1.2 in §1.1.2). A function that can be computed by such a transducer is called an *aperiodic sequential function*.

Example 2.2.3. The transducer in Figure 2.2.1 computes $f : w \in \{a, b\}^* \mapsto a \cdot \psi(w) \cdot b$ where ψ is the monoid morphism that doubles every b: $\psi(a) = a$ and $\psi(b) = bb$. Its transition monoid T is generated by $G = \{(\delta_{\mathrm{st}}(-, a) : q \mapsto q_a), (\delta_{\mathrm{st}}(-, b) : q \mapsto q_b)\}$; one can verify that $T = G \cup \{\mathrm{id}\}$ and therefore $\forall h \in T,\ h \circ h = h$. Thus, f is an aperiodic sequential function.

2.2.1. **The Krohn–Rhodes decomposition and wreath products of monoids.** One of the fundamental results of automata theory is a kind of "prime factorization" of sequential functions, which we state below. (Such factorization theorems are also important for larger classes of transductions, see e.g. [BD20], Theorem 1.4.7 and Definition 2.5.1.)

Theorem 2.2.4 (Krohn–Rhodes)**.** *Any* sequential function $f : \Gamma^* \to \Sigma^*$ *can be realized as a composition* $f = f_1 \circ \ldots \circ f_n$ *(with* $f_i : \Pi_i^* \to \Pi_{i-1}^*$, $\Pi_0 = \Sigma$ *and* $\Pi_n = \Gamma$*) where each function* f_i *is computed by:*

- *either a* flip-flop transducer*, i.e. an aperiodic sequential transducer* with 2 states*;*
- *or a sequential transducer whose transition monoid is a* group*.*

Note that the example of sequential transducer in Figure 2.2.1 has two states. It is also aperiodic (Example 2.2.3), so it is a flip-flop.

Theorem 2.2.4 not the standard way to state the Krohn–Rhodes decomposition, though one may find it in the literature, usually without proof (e.g. [Boj18, §1.1]). Usual statements involve either a special case of sequential transducers called Mealy machines (Remark 2.3.10) or, more often, finite monoids (both appear in the original paper[2] [KR65]). Here we show how the algebraic formulation can be shown to imply Theorem 2.2.4; this material is well-known among practitioners of automata theory.

Definition 2.2.5. 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. Next, we recall the algebraic construction that corresponds to composition of sequential functions.

[2] What is called a "sequential machine" in [KR65] is in fact a Mealy machine.

Definition 2.2.6. 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 2.2.7. *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 2.2.8. 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 2.2.9. *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 2.2.10 (Krohn–Rhodes with strong divisors [DKS12, 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 2.2.11. The flip-flop transformation monoid is precisely the transition monoid of the transducer of Example 2.2.3 endowed with its action on the set of states.

Remark 2.2.12. We can also require G above to be a *simple* group. This is the statement given in [DKS12], 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 2.2.13. 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 2.2.4 as a corollary of Theorem 2.2.10.

Proposition 2.2.14. *Let (Q, δ, q_I, F) be a sequential transducter with transition monoid T describing a function $f : \Gamma^* \to \Sigma^*$. Suppose that (Q, T) strongly divides some faithful transformation monoid $(X, M) \wr (Y, N)$. Then there is an alphabet Π and transducers*

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

such that

- *the sequential functions $f_X : \Gamma^* \to \Pi^*$ and $f_Y : \Pi^* \to \Sigma^*$ that they respectively compute verify $f = f_X \circ f_Y$;*
- *there are injective homomorphisms $T_X \hookrightarrow M$ and $T_Y \hookrightarrow N$ from their respective transition monoids.*

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 $a \in \Gamma$, we choose an element $(g_a, n_a) \in M^Y \times N$ which is mapped by φ to $\delta_{\text{st}}(-, a) \in T$. Set $\Pi = (\Gamma \uplus \{*\}) \times Y$, $(x_I, y_I) = s^{-1}(x, y)$ and

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

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

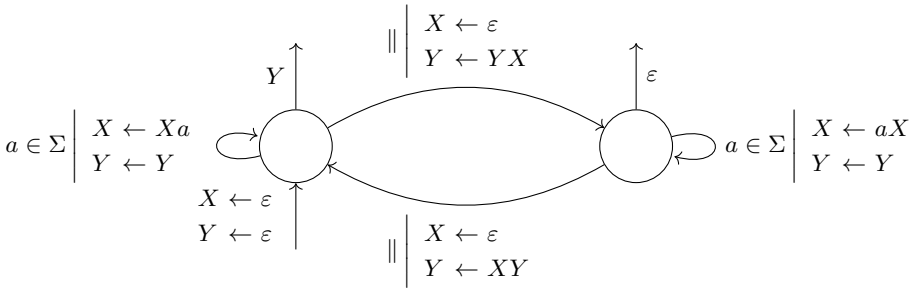
This generalizes to n-fold wreath products in the expected way.

Proposition 2.2.15. *Let T be the transition monoid of a sequential transducer with state space Q computing the function $f : \Gamma^* \to \Sigma^*$. 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 $f_i : \Pi_i^* \to \Pi_{i-1}^*$, $\Pi_0 = \Sigma$ and $\Pi_n = \Gamma$) 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 $a \in \Gamma$, pick an element $m_a \in K$ such that $\varphi(m_a) = \delta_{\text{st}}(-, a)$. Then, letting (Q, δ, q_I, F) being the transducer under scrutiny, a suitable transducer (X, δ', x_I, F') is defined by setting $\delta'(x, a) = (x \cdot m_a, \delta_{\text{out}}(s(x), a))$ and $F'(x) = F(s(x))$.
- For $n > 1$, use Proposition 2.2.14 and the induction hypothesis. □

Proof of Theorem 2.2.4. Let (Q, δ, q_I, F) be a transducter computing a certain sequential function $f : \Gamma^* \to \Sigma^*$ and let T be its transition monoid. By Theorem 2.2.10, 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 2.2.4, thanks to Proposition 2.2.9). By applying Proposition 2.2.15, we may obtain transducers $\mathcal{T}_i$ implementing sequential functions $f_i : \Pi_i^* \to \Pi_{i+1}^*$ such that $\Pi_0 = \Gamma$, $\Pi_k = \Sigma$ 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. □



$$
\begin{array}{rcll}
(\Sigma \sqcup \{\|\})^* & \to & \Sigma^* & (w_i \in \Sigma^*,\ (\Sigma \sqcup \{\|\})^* \cong \Sigma^*(\|\Sigma^*)^*) \\
w_0 & \mapsto & \varepsilon & \\
w_0\| \dots \|w_{2k} & \mapsto & \texttt{reverse}(w_{2k-1})\texttt{reverse}(w_{2k-3}) \dots \texttt{reverse}(w_1) w_0 w_2 \dots w_{2k-2} & \\
w_0\| \dots \|w_{2k+1} & \mapsto & \varepsilon &
\end{array}
$$

FIGURE 2.3.1. An informal depiction of a streaming string transducer and the induced map $(\Sigma \sqcup \{\|\})^* \to \Sigma^*$.

2.3. Streaming string transducers (SSTs)

Following Section 1.1.3, the next classical transduction class after sequential functions would be *rational* functions. However, we do not have any use for them in this manuscript, so we will skip directly to *regular* functions. As we have said many times in Chapter 1, we take copyless streaming string transducers (SSTs) as their reference definition. Let us start by recalling the copyful version; the copylessness restriction will be defined later. (This is a bit anachronistic: Alur and Černý first introduced copyless SSTs [AČ10], and only later were copyful SSTs studied by Filiot and Reynier [FR21].)

2.3.1. **Copyful SSTs.** An SST is an automaton whose internal memory contains, in addition to its control state, a finite number of string-valued registers. It processes its input in a single left-to-right pass. Each time a letter is read, the contents of the registers may be recombined by concatenation to determine the new register values. Formally:

Definition 2.3.1. Fix a finite alphabet Σ. Let R and S be two finite sets *disjoint from* Σ; we shall consider their elements to be "register variables".

For any word $\omega \in (\Sigma \cup R)^*$, we write $\omega^\dagger : (\Sigma^*)^R \to \Sigma^*$ for the map that sends $(u_r)_{r \in R}$ to ω in which every occurrence of a register variable $r \in R$ is replaced by u_r – formally, we apply to ω the morphism $(\Sigma \cup R)^* \to \Sigma^*$ that maps $c \in \Sigma$ to itself and $r \in R$ to u_r.

A *register assignment* α from R to S (over Σ) is a map $\alpha : S \to (\Sigma \cup R)^*$. It induces the action $\alpha^\dagger : \vec{u} \in (\Sigma^*)^R \mapsto (\alpha(s)^\dagger(\vec{u}))_{s \in S} \in (\Sigma^*)^S$ (which indeed goes "from R to S").

For instance, $t : Z \mapsto aXbY$ is a register assignment from $\{X, Y\}$ to $\{Z\}$ over the alphabet $\{a, b\}$ and $t^\dagger(X \mapsto b,\ Y \mapsto aa) = (Z \mapsto abbaa)$.

Remark 2.3.2. Some papers e.g. [AFT12; DJR18] call register assignments *substitutions*. We avoid this name since it differs from its meaning in the context of our "composition by substitution" operation (next chapter) or of the λ-calculus (in the rest of the manuscript).

We will also use the name *register transition* later for a slight variant of the above definition that is more convenient for our category-theoretic developments (Definition 4.2.10).

Definition 2.3.3 ([FR21])**.** A (deterministic copyful) *streaming string transducer* (SST) with input alphabet Γ and output alphabet Σ is a tuple $\mathcal{T} = (Q, q_0, R, \delta, \vec{u}_I, F)$ where

- Q is a finite set of *states* and $q_0 \in Q$ is the *initial state*;
- R is a finite set of *register variables*, that we require to be *disjoint from* Σ;
- $\delta : Q \times \Gamma \to Q \times (R \to (\Sigma \cup R)^*)$ is the *transition function* – we abbreviate $\delta_{\text{st}} = \pi_1 \circ \delta$ and $\delta_{\text{reg}} = \pi_2 \circ \delta$, where π_i is the projection from $X_1 \times X_2$ to its i-th component X_i;
- $\vec{u}_I \in (\Sigma^*)^R$ describes the *initial register values*;
- $F : Q \to (\Sigma \cup R)^*$ describes how to recombine the final values of the registers, depending on the final state, to produce the output.

The function $\Gamma^* \to \Sigma^*$ computed by $\mathcal{T}$ is

$$w_1 \dots w_n \;\mapsto\; F(q_n)^\dagger \circ \delta_{\text{reg}}(q_{n-1}, w_n)^\dagger \circ \dots \circ \delta_{\text{reg}}(q_0, w_1)^\dagger(\vec{u}_I)$$

where the sequence of states $(q_i)_{0 \le i \le n}$ (sometimes called the *run* of the transducer over the input word) is inductively defined, starting from the fixed initial state q_0, by $q_i = \delta_{\text{st}}(q_{i-1}, w_i)$.

An example of SST with 2 states is given in Figure 2.3.1. We also provide below an example of a single-state SST (recall the underlined alphabet notation from Section 2.1).

Example 2.3.4. Let $\Sigma = \Gamma \cup \underline{\Gamma}$. We consider a SST $\mathcal{T}$ with $Q = \{q\}$, $R = \{X, Y\}$ and

$$\vec{u}_I = (\varepsilon)_{r \in R} \qquad F(q) = Y \qquad \forall c \in \Gamma,\ \delta(q, c) = (q,\ (X \mapsto cX,\ Y \mapsto \underline{c}XY))$$

If we write (v, w) for the family $(u_r)_{r \in R}$ with $u_X = v$ and $u_Y = w$, then the action of the register assignments may be described as $(X \mapsto cX,\ Y \mapsto \underline{c}XY)^\dagger(v, w) = (c \cdot v,\ \underline{c} \cdot v \cdot w)$.

Let $\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4} \in \Gamma$. After reading $\mathtt{1234} \in \Gamma^*$, the values stored in the registers of $\mathcal{T}$ are

$$(X \mapsto \mathtt{4}X,\ Y \mapsto \underline{\mathtt{4}}XY)^\dagger \circ \dots \circ (X \mapsto \mathtt{1}X,\ Y \mapsto \underline{\mathtt{1}}XY)^\dagger(\varepsilon, \varepsilon) = (\mathtt{4321}, \underline{\mathtt{4}}\mathtt{321}\underline{\mathtt{3}}\mathtt{21}\underline{\mathtt{2}}\mathtt{11}\underline{\mathtt{1}})$$

Since $F(q) = Y$, the function defined by $\mathcal{T}$ maps $\mathtt{1234}$ to $\underline{\mathtt{4}}\mathtt{321}\underline{\mathtt{3}}\mathtt{21}\underline{\mathtt{2}}\mathtt{11}\underline{\mathtt{1}} \in (\Gamma \cup \underline{\Gamma})^* = \Sigma^*$.

2.3.2. **Copyless SSTs and regular functions.** The streaming string transducer of Figure 2.3.1 is in fact copyless; let us say precisely what this means.

Definition 2.3.5 ([AČ10])**.** A register assignment $\alpha : S \to (\Sigma \cup R)^*$ from R to S is said to be *copyless* when each $r \in R$ *occurs at most once among all the strings* $\alpha(s)$ *for* $s \in S$, i.e. it does not occur at least twice in some $\alpha(s)$, nor at least once in $\alpha(s)$ and at least once in $\alpha(s')$ for some $s \neq s'$. (This restriction does not apply to the letters in Σ.)

A streaming string transducer is *copyless* if all the assignments in the image of its transition function are copyless. In this dissertation, we take computability by copyless SSTs as the definition of *regular functions* (but see Theorem 3.2.3 for another standard definition).

Remark 2.3.6. The SST of Example 2.3.4 is *not* copyless: in a transition $\alpha = \delta_{\text{reg}}(q, c)$, the register X appears twice, once in $\alpha(X) = cX$ and once in $\alpha(Y) = \underline{c}XY$; in other words, its value is *duplicated* by the action $\alpha^\dagger$. In fact, it computes a function whose output size is quadratic in the input size, while regular functions have linearly bounded output.

Example 2.3.7 (Iterated reverse)**.** The following single-state SST is copyless:

$$\Gamma = \Sigma \text{ with } \# \in \Sigma \qquad Q = \{q\} \qquad R = \{X, Y\} \qquad \vec{u}_I = (\varepsilon)_{r \in R} \qquad F(q) = XY$$

$$\delta(q, \#) = (q,\ (X \mapsto XY\#,\ Y \mapsto \varepsilon)) \qquad \forall c \in \Sigma \setminus \{\#\},\ \delta(q, c) = (q,\ (X \mapsto X,\ Y \mapsto cY))$$

For $u_1, \dots, u_n \in (\Sigma \setminus \{\#\})^*$, it maps $u_1\# \dots \#u_n$ to $\mathtt{reverse}(u_1)\# \dots \#\mathtt{reverse}(u_n)$. (This function is taken from [Boj18, p. 1].)

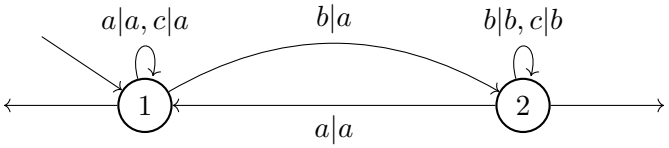
The concrete SSTs (copyless or not) that we have seen for now are all single-state, except for Figure 2.3.1. A source of stateful copyless SSTs is the following:

Proposition 2.3.8. *Every sequential function is regular.*

Proof idea. Any sequential transducer can be translated into a copyless SST with the same set of states and a single register. □

In fact, states are strictly necessary for this, as the example below shows.

Proposition 2.3.9. *The sequential (and therefore regular) function defined by the aperiodic sequential transducer below* cannot *be computed by a* single-state copyless SST.



We did not find Proposition 2.3.9 stated the literature, though it is probably well-known among experts; we will give a proof at the end of Section 2.3.4 once we have recalled the necessary algebraic machinery.

Remark 2.3.10. Compared to Figure 2.2.1, the transducer of Proposition 2.3.9 has the particularity of being *letter-to-letter*: each transition outputs exactly one letter, and the final output word is empty. Letter-to-letter sequential transducers are also known as a *Mealy machines*, and the functions computed by aperiodic Mealy machines form one of the weakest classical transduction classes.

Finally, let us recall some basic properties of regular functions:

Proposition 2.3.11 (folklore, e.g. [AČ10, Proposition 2])**.** *Let $f, g : \Gamma^* \to \Sigma^*$ be regular functions and $L \subseteq \Gamma^*$ be a regular language. The function that coincides with f on L and with g on $\Gamma^* \setminus L$ is regular, and so is $w \mapsto f(w) \cdot g(w)$.*

Proof. Both are easy exercises involving a "product construction" on SSTs: if f and g are computed by copyless SSTs with states Q_f, Q_g and register variables R_f, R_g, and L is recognized by some finite monoid M, then the "regular conditional" is computed by a copyless SST with states $Q_f \times Q_g \times M$ and registers $R_f + R_g$, while the concatenation is computed by an SST with states $Q_f \times Q_g$ and registers $R_f + R_g$. □

Proposition 2.3.12 (special case of Theorem 2.3.14(i))**.** *Every regular function f has a linearly bounded output length: $|f(w)| = O(|w|)$.*

2.3.3. **Layered SSTs.** An intermediate machine model between copyful and copyless SSTs – where duplication is controlled, but not outright forbidden – has been recently introduced by Douéneau-Tabot et al. [DFG20]. Its interest is justified by Theorem 2.3.14 below; for now let us recall its definition.

Definition 2.3.13 ([DFG20])**.** Let $k \in \mathbb{N}$ and $\alpha : R \to (\Sigma \cup R)^*$. The register assignment α is *k-layered* with respect to a partition $R = R_0 \sqcup \ldots \sqcup R_k$ when for $0 \leq i \leq k$,

- for $r \in R_i$, we have $\alpha(r) \in (\Sigma \cup R_0 \cup \ldots \cup R_i)^*$;
- each register variable in R_i appears at most once among all the $\alpha(r)$ for $r \in R_i$ (however, those from $R_0 \sqcup \ldots \sqcup R_{i-1}$ may appear an arbitrary number of times).

A streaming string transducer is *k-layered* if its registers can be partitioned in such a way that all assignments in the transitions of the SST are k-layered.

Beware: with this definition, the registers of a k-layered SST are actually divided into $k+1$ layers, not k. For instance, the transducer of Example 2.3.4 is 1-layered with $R_0 = \{X\}$ and $R_1 = \{Y\}$, and copyless SSTs are the same thing as 0-layered SSTs.

There also exist register assignments that cannot be made k-layered no matter the choice of partition, such as $X \mapsto XX$. Using such assignments, one can indeed build SSTs that compute functions f such that e.g. $|f(w)| = 2^{|w|}$. This kind of superpolynomial growth turns out to be the only obstruction to the existence of an equivalent layered SST, according to the following result.

Theorem 2.3.14 ([DFG20]). *Let $f : \Gamma^* \to \Sigma^*$ and $k \in \mathbb{N}$. The following are equivalent:*

(i) *f is computed by a copyful SST and $f(|w|) = O(|w|^{k+1})$;*
(ii) *f is computed by some k-layered SST;*
(iii) *f is computed by some k-marble transducer – a kind of device introduced in [DFG20] and mentioned in §1.2.4, that we will not use it in the rest of the paper.*

Remark 2.3.15. This use of a syntactic "layering" condition to characterize the polynomially growing functions computable by SSTs arguably has an old precursor: Schützenberger's theorem on polynomially bounded $\mathbb{Z}$-rational series, which dates back to the 1960s (see for instance [BR10, Chapter 9, Section 2] – the preface of the same book describes this theorem as "one of the most difficult results in the area"). Let us give a brief exposition.

For a semiring S, a function $f : \Sigma^* \to S$ is sometimes called a "series" because it can be seen as formal power series in several non-commuting indeterminates (or variables). Indeed, each letter of Σ can be seen as an indeterminate, and a word in Σ^* is then considered as a formal product, i.e. a monomial. Then the function f corresponds to the coefficients of

$$\sum_{w \in \Sigma^*} f(w) \cdot w$$

A *$\mathbb{Z}$-rational series* $f : \Sigma^* \to \mathbb{Z}$ is a function of the form $f : w \in \Sigma^* \mapsto X^T \cdot \Phi(w) \cdot Y$ where $X, Y \in \mathbb{Z}^R$ and Φ is a morphism from Σ^* to the multiplicative monoid of R-indexed square matrices over $\mathbb{Z}$, for some finite set R. (For more on rational series over any semiring, and how they can be seen as generalizations of regular languages, see [Sak09, Chapter III].) This data (X, Φ, Y) has a clear interpretation as a "simple SST" (a single-state SST with an additional condition, cf. Remark 2.4.3) with register set R, whose register values are integers rather than strings. Schützenberger's theorem says that any $\mathbb{Z}$-rational series f with polynomial growth (i.e. $|f(w)| = |w|^{O(1)}$ where $|\cdot|$ on the left is the absolute value) can be written as $f : w \mapsto X^T \cdot \Phi(w) \cdot Y$ where

(i) the image of Φ has a block triangular structure;
(ii) the projection of this image on each diagonal block is a finite monoid.

The first item gives us a partition of the register into layers where each layer "depends" only on the ones below them. The finiteness condition in the second item is equivalent to having bounded coefficients, which means that the register assignments within each layer are *bounded-copy*, while in a layered SST, they would be *copyless* instead – but bounded-copy SSTs are known to be equivalent to copyless SSTs[3] (see e.g. [AFT12; DJR18]). The theorem also states a relationship between the number of blocks and the growth rate.

Via the canonical isomorphism $\{a\}^* \cong \mathbb{N}$, streaming string transducers with unary output alphabet compute *$\mathbb{N}$-rational series*. The counterpart of Schützenberger's theorem over $\mathbb{N}$ is thus a corollary of the results of [DFG20] on layered SSTs.

2.3.4. **Transition monoids of (copyless) SSTs.** We now recall a central tool to study streaming string transducers, that also prefigures their category-theoretic treatment: the monoid structure on register assignments (see e.g. [AFT12, §III.C]).

Definition 2.3.16. Let $\mathcal{M}_{R,\Sigma} = R \to (\Sigma \cup R)^*$ for $R \cap \Sigma = \varnothing$. We endow it with the following composition operation, that makes it into a monoid:

$$\alpha \bullet \beta = \alpha^\star \circ \beta \quad \text{where } \alpha^\star \in \mathrm{Hom}((\Sigma \cup R)^*, (\Sigma \cup R)^*),\ \alpha^\star(x) = \begin{cases} \alpha(x) & \text{for } x \in R \\ x & \text{for } x \in \Sigma \end{cases}$$

[3] This is a consequence of the older and stronger fact that macro tree transducers with linearly bounded growth compute regular tree functions [EM03b].

The monoid $\mathcal{M}_{R,\Sigma}$ thus defined is isomorphic to a submonoid of $\mathrm{Hom}((\Sigma \cup R)^*, (\Sigma \cup R)^*)$ with function composition. It admits a submonoid of *copyless* assignments.

Definition 2.3.17. We write $\mathcal{M}^{\mathrm{cl}}_{R,\Sigma}$ for the set of all $\alpha \in \mathcal{M}_{R,\Sigma}$ such that each letter $r \in R$ occurs at most once among all the $\alpha(r')$ for $r' \in R$.

Proposition 2.3.18. *$\mathcal{M}^{\mathrm{cl}}_{R,\Sigma}$ is a submonoid of $\mathcal{M}_{R,\Sigma}$. In other words, copylessness is preserved by composition (and the identity assignment is copyless).*

The following proposition ensures that this composition does what we expect. Recall from Definition 2.3.1 that $(-)^\dagger$ interprets register assignments as functions on tuples of strings, i.e. it sends $\mathcal{M}_{R,\Sigma}$ to $(\Sigma^*)^R \to (\Sigma^*)^R$.

Proposition 2.3.19. *For all $\alpha, \beta \in \mathcal{M}_{R,\Sigma}$, we have $(\alpha \bullet \beta)^\dagger = \beta^\dagger \circ \alpha^\dagger$. In other words, $(-)^\dagger$ is a monoid morphism if we endow $(\Sigma^*)^R \to (\Sigma^*)^R$ with* reverse *function composition (just as we did for $Q \to Q$ in Definition 2.2.2).*

To incorporate information concerning the states of an SST, we use a construction that can be seen as special case of the wreath product of transformation monoids (Definition 2.2.6): the reader may check that the monoid denoted by $M \wr Q$ below is canonically isomorphic to the wreath product $(M, M) \wr (Q,\ Q \to Q)$ (which justifies our notation for the former).

Definition 2.3.20. Let M be a monoid; let $m \cdot m'$ denote the multiplication of $m, m' \in M$. We define $M \wr Q$ as the monoid whose set of elements is $Q \to Q \times M$ and whose multiplication is, for $\mu, \mu' : Q \to Q \times M$, (writing $\pi_1 : Q \times M \to Q$ and $\pi_2 : Q \times M \to M$ for the projections)

$$(\mu \bullet \mu') : q \mapsto (\pi_1 \circ \mu' \circ \pi_1 \circ \mu(q),\ (\pi_2 \circ \mu(q)) \cdot (\pi_2 \circ \mu' \circ \pi_1 \circ \mu(q)))$$

Proposition 2.3.21. *Let $(Q, q_0, R, \delta, \vec{u}_I, F)$ be an SST that computes $f : \Gamma^* \to \Sigma^*$ (using the notations of Definition 2.3.3). For all $c \in \Gamma$, we have $\delta(-, c) \in \mathcal{M}_{R,\Sigma} \wr Q$, and the SST is copyless if and only if $\{\delta(-, c) \mid c \in \Gamma\} \subseteq \mathcal{M}^{\mathrm{cl}}_{R,\Sigma} \wr Q$. Furthermore, for all $w_1 \dots w_n \in \Gamma^*$,*

$$f(w_1 \dots w_n) \quad = \quad F(g(q_0))^\dagger(\alpha^\dagger(\vec{v})) \quad \text{where} \quad (g, \alpha) = \delta(-, w_1) \bullet \cdots \bullet \delta(-, w_n)$$

Finally, it can be useful to consider assignments over an *empty* output alphabet. This allows us to keep track of how the registers are shuffled around by transitions.

Proposition 2.3.22. *Let R and Σ be disjoint finite sets. There is a monoid morphism $\mathcal{M}_{R,\Sigma} \to \mathcal{M}_{R,\varnothing}$, that sends the submonoid $\mathcal{M}^{\mathrm{cl}}_{R,\Sigma}$ to $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$. For any Q, this extends to a morphism $\mathcal{M}_{R,\Sigma} \wr Q \to \mathcal{M}_{R,\varnothing} \wr Q$ that sends $\mathcal{M}^{\mathrm{cl}}_{R,\Sigma} \wr Q$ to $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q$. We shall use the name $\mathtt{erase}_\Sigma$ for both morphisms (R and Q being inferred from the context).*

Remark 2.3.23. Consider an SST with a transition function δ. Let $\varphi_\delta \in \mathrm{Hom}(\Gamma^*, \mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)$ be defined by $\varphi_\delta(c) = \mathtt{erase}_\Sigma(\delta(-, c))$ for $c \in \Gamma$. The range $\varphi_\delta(\Gamma^*)$ is precisely the *substitution transition monoid (STM)* defined in [DJR18, Section 3].

Proposition 2.3.24. *For any finite R, the monoid $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ is* finite. *As a consequence, the substitution transition monoid of any* copyless *SST is finite.*

Proof idea. For all $\alpha \in \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ and $r \in R$, observe that $|\alpha(r)| \leq |R|$. □

Let us now show how these tools can be applied to prove an earlier claim from §2.3.2.

Proof of Proposition 2.3.9. Consider any single-state copyless SST computing some $g : \{a, b, c\}^* \to \{a, b\}^*$ with a set of registers R. We wish to show g does not coincide with the function computed by the sequential transducer of Proposition 2.3.9.

Let $\omega \in (\{a, b\} \cup R)^*$ be the image of the single state by the output function, and for each letter $x \in \{a, b, c\}$, let $\alpha_x : R \to (\{a, b\} \cup R)^*$ be the copyless assignment performed

by the SST when it reads x (that is, using the notations of Definition 2.3.3, $\omega = F(q)$ and $\alpha_x = \delta_{\text{reg}}(q, x)$ with $Q = \{q\}$). Let $\vec{u}$ be the initial register contents. Then

$$\forall x \in \{a,b,c\}^*,\ \forall n \in \mathbb{N},\ g(x \cdot c^n) = \omega^\dagger \circ (\alpha_c^\dagger)^n \circ \alpha_x^\dagger(\vec{u})$$

Let $\omega_n = (\alpha_c^\star)^n(\omega)$, using the notation $(-)^\star$ from Definition 2.3.16; in particular $\omega_0 = \omega$. One can check that, for all $n \in \mathbb{N}$:

- $\omega_n^\dagger = \omega^\dagger \circ (\alpha_c^\dagger)^n$;
- since α_c is copyless, $|\omega_n|_r \leq |\omega|_r$ for all $r \in R$, writing $|w|_x$ for the number of occurrences of x in $w \in \Sigma^*$ for $x \in \Sigma$.

Let $(v_{x,r})_{r\in R} = \alpha_x^\dagger(\vec{u})$ for $x \in \{a,b,c\}$; that is, $v_{x,r}$ the value stored in the register $r \in R$ after the SST has read the single letter x. We can rewrite the above equation as

$$\forall x \in \{a,b,c\}^*,\ \forall n \in \mathbb{N},\ g(x \cdot c^n) = \omega_n^\dagger((v_{x,r})_{r\in R})$$

and derive a numerical (in)equality

$$\forall x \in \{a,b,c\}^*,\ \forall n \in \mathbb{N},\ |g(x \cdot c^n)|_a = |\omega_n|_a + \sum_{r\in R} |\omega_n|_r |v_{x,r}|_a \underset{n\to+\infty}{=} |\omega_n|_a + O(1)$$

using the fact that $|\omega_n|_r$, as a non-negative quantity lower than the constant $|\omega|_r$, is $O(1)$.

From this, it follows that as n increases, the difference between $|g(a \cdot c^n)|_a$ and $|g(b \cdot c^n)|_a$ stays bounded. This property distinguishes g from the $f : \{a,b,c\}^* \to \{a,b\}^*$ computed by the transducer given in Proposition 2.3.9, since

$$\forall n \in \mathbb{N},\quad |f(a \cdot c^n)|_a = \left|a^{n+1}\right|_a = n+1 \quad \text{and} \quad |f(b \cdot c^n)|_a = \left|b^{n+1}\right|_a = 0 \qquad \square$$

2.4. HDT0L transductions

Filiot and Reynier [FR21] characterized the expressive power of copyful SSTs using some kind of L-systems. The latter were originally introduced by Lindenmayer [Lin68] in the 1960s as a way to generate formal languages, with motivations from biology. While this language-centric view is still predominant, the idea of considering variants of L-systems as specifications for string-to-string functions – whose range are the corresponding languages – seems to be old. For instance, in a paper from 1980 [ERS80], one can find (multi-valued) string functions defined by ET0L systems. For SSTs, the right notion is the following:

Definition 2.4.1 (following [FR21])**.** A *HDT0L system* consists of:

- an input alphabet Γ, an output alphabet Σ, and a working alphabet Δ (all *finite*);
- an initial word $d \in \Delta^*$;
- for each $c \in \Gamma$, a *monoid morphism* $h_c \in \mathrm{Hom}(\Delta^*, \Delta^*)$;
- a final morphism $h' \in \mathrm{Hom}(\Delta^*, \Sigma^*)$.

It defines the transduction taking $w = w_1 \dots w_n \in \Gamma^*$ to $h' \circ h_{w_1} \circ \dots \circ h_{w_n}(d) \in \Sigma^*$.

Theorem 2.4.2 ([FR21])**.** *A function $\Gamma^* \to \Sigma^*$ can be computed by a copyful streaming string transducer (§2.3.1) if and only if it can be specified by an HDT0L system.*

Other equivalent ways to define this class of functions, which we call *HDT0L transductions*, include catenative recurrent equations, higher-order pushdown transducers of level 2 (both by Ferté, Marin and Sénizergues [FMS14][4]) and unbounded marble transducers [DFG20].

[4] Those characterizations had previously been announced in an invited paper by Sénizergues [Sén07]. Some other results announced in [Sén07] are proved in [Cad+21]. Moreover, the definition of HDT0L systems given in [Sén07; FMS14] makes slightly different choices of presentation: the family $(h_c)_{c\in\Gamma}$ is presented as a morphism $H : \Gamma^* \to \mathrm{Hom}(\Delta^*, \Delta^*)$, and an initial *letter* is used instead of an initial word; this is of no consequence regarding the functions that can be expressed.

Remark 2.4.3. As observed in [FR21, Lemma 3.3], there is a natural translation from HDT0L systems to SSTs whose range is composed precisely of the *single-state* SSTs whose transitions and final output function *do not access the letters of their output alphabet* – those are called *simple SSTs* in [DFG20, §5.1]. This involves a kind of reversal: the initial register values correspond to the final morphisms, while the final output function corresponds to the initial word. Thus, Theorem 2.4.2 is essentially a state elimination result; a direct translation from SSTs to single-state SSTs[5] has also been given in Benedikt et al.'s paper on polynomial automata[6] [Ben+17, Proposition 8]. However, it does *not* preserve the subclass of *copyless* SSTs (this would contradict Proposition 2.3.9).

In more detail, an HDT0L system $(\Gamma, \Sigma, \Delta, d, (h_c)_{c\in\Gamma}, h')$ is equivalent to the streaming string transducer specified by the following data:

- a singleton set of states: $Q = \{q\}$;
- the working alphabet as the set of registers: $R = \Delta$ (minor technicality: if $\Delta \cap \Sigma \neq \varnothing$, one should take R to be a copy of Δ that is disjoint from Σ);
- $h_c \in \mathrm{Hom}(\Delta^*, \Delta^*) \cong (\Delta \to \Delta^*) \subseteq (\Delta \to (\Sigma \cup \Delta)^*)$ as the register assignment associated to an input letter $c \in \Gamma$ – in other words, the transition function is $\delta : (q, c) \mapsto (q, (h_c)_{\restriction \Delta})$;
- $(h'(r))_{r\in\Delta} \in (\Sigma^*)^R$ as the initial register values;
- $F : q \mapsto d$ as the final output function ($d \in \Delta^* \subseteq (\Sigma \cup \Delta)^*$).

The cases of the transition and output functions involve a codomain extension from Δ^* to $(\Sigma \cup \Delta)^*$. This reflects the intuition that a HDT0L system "cannot access the output alphabet" (except in the initial register contents).

To prove the equivalence, the key observation is that h_c is turned into $\delta(-, c)$ by a *morphism* from $\mathrm{Hom}(\Delta^*, \Delta^*)$ to $\mathcal{M}_{\Delta,\varnothing} \wr \{q\} \subset \mathcal{M}_{\Delta,\Sigma} \wr \{q\}$, using the notations from Section 2.3.4. We leave the details to the reader.

2.4.1. **Layered HDT0L systems.** Let us transpose the layering condition (§2.3.3) from streaming string transducers to HDT0L systems. The hierarchy of models that we get corresponds *with an offset* to layered SSTs.

Definition 2.4.4. A HDT0L system $(\Gamma, \Sigma, \Delta, d, (h_c)_{c\in\Gamma}, h')$ is *k-layered* if its working alphabet can be partitioned as $\Delta = \Delta_0 \sqcup \cdots \sqcup \Delta_k$ such that, for all $c \in \Gamma$ and $i \in \{0, \dots, k\}$:

- for $r \in \Delta_i$, we have $h_c(r) \in (\Delta_0 \sqcup \cdots \sqcup \Delta_i)^*$;
- each letter in Δ_i appears at most once among all the $\alpha(r)$ for $r \in \Delta_i$ (but those in $\Delta_0 \sqcup \cdots \sqcup \Delta_{i-1}$ may appear an arbitrary number of times).

Theorem 2.4.5. *For $k \in \mathbb{N}$, a function can be computed by a k-layered SST if and only if it can be specified by a $(k+1)$-layered HDT0L system.*

In particular, regular functions correspond to 1-layered HDT0L systems.

Proof of ($\Rightarrow$). The translation from SSTs to HDT0L systems in [FR21, Lemma 3.5] turns out to work. It is also formulated in terms of "simple SSTs" (isomorphic to HDT0L systems, cf. Remark 2.4.3) in [DFG20, §5.1], where the authors remark that it "does not preserve copylessness nor k-layeredness": indeed, what we show is that it increments the number of layers by one! For the sake of clarity, we give an alternative presentation that decomposes it into two steps.

Let Γ be the input alphabet and Σ be the output alphabet. Let $\mathcal{T}$ be a streaming string transducer with a *k-layered* set of register variables $R = R_0 \sqcup \cdots \sqcup R_k$. First, we build a

[5] The lookahead elimination theorem for macro tree transducers [EV85, Theorem 4.21] arguably generalizes this to trees. Indeed, lookahead in MTTs corresponds to bottom-up states in register transducers according to the duality sketched at the beginning of Chapter 6.

[6] See Example 4.2.6 for a definition of polynomial automata.

$(k+1)$-*layered* SST $\mathcal{T}'$ that computes the same function, with the set of registers

$$R' = \underline{\Sigma} \cup R = R'_0 \sqcup \cdots \sqcup R'_{k+1} \qquad R'_0 = \underline{\Sigma} \qquad \forall i \in \{1, \dots, k+1\},\ R'_i = R_{i-1}$$

assuming $\underline{\Sigma} \cap R = \varnothing$, and whose register assignments are *without fresh letters*: the range of every $\alpha' : R' \to (\Sigma \cup R')^*$ is included in R'^*, which allows us to write $\alpha' : R' \to R'^*$. This already brings us closer to the definition of HDT0L systems, since $(R \to R^*) \cong \mathrm{Hom}(R^*, R^*)$. Similarly, we will ensure that the range of the output function of $\mathcal{T}'$ is included in R'.

Let $\mathtt{underline}_\Sigma \in \mathrm{Hom}((\Sigma \cup R)^*, (\underline{\Sigma} \cup R)^*)$ be defined in the expected way, and note that its codomain is equal to R'^*. We specify $\mathcal{T}'$ as follows (and leave it to the reader to check that this works):

- the state space Q, initial state and state transitions are the same as those of $\mathcal{T}$;
- the initial value of $r' \in R'$ is the same as for $\mathcal{T}$ if $r' \in R$, or the single letter c if $r' = \underline{c} \in \underline{\Sigma}$;
- every assignment $\alpha : R \to (\Sigma \cup R)$ that appears in some transition of $\mathcal{T}$ becomes, in $\mathcal{T}'$,

$$\alpha' : R'^* \to R'^* \qquad \alpha' : \underline{c} \in \underline{\Sigma} \mapsto \underline{c} \qquad \alpha' : r \in R \mapsto \mathtt{underline}_\Sigma(\alpha(r))$$

- its output function is $F' = \mathtt{underline}_\Sigma \circ F$ where F is the output function of $\mathcal{T}$.

Thus, the idea is to store a copy of $c \in \Sigma$ in the register $\underline{c}$. Since this register may feed in a copyful way all other registers (in a SST, there are no restrictions on the use of output alphabet letters), it must sit at the lowest layer, hence $R'_0 = \underline{\Sigma}$ and the resulting offset.

Next, we turn $\mathcal{T}'$ into an equivalent HDT0L system with $(k+1)$-layered working alphabet

$$\Delta = R' \times Q = \Delta_0 \sqcup \ldots \sqcup \Delta_{k+1} \qquad \forall i \in \{0, \dots, k+1\},\ \Delta_i = R'_i \times Q$$

For $q \in Q$, let $\mathtt{pair}_q \in \mathrm{Hom}(R'^*, \Delta^*)$ be such that $\mathtt{pair}_q(r') = (r', q)$ for $r' \in R'$.

Let $Q = \{q^{(1)}, \dots, q^{(n)}\}$ be the states of $\mathcal{T}'$ (which are also those of $\mathcal{T}$), with $q^{(1)}$ being its initial state[7]. Using the fact that $\mathcal{T}'$ is without fresh letters, let $F' : Q \to R'^*$ be its final output function. The initial word of our HDT0L system is then

$$d = \mathtt{pair}_{q^{(1)}}\left(F'\left(q^{(1)}\right)\right) \cdot \ldots \cdot \mathtt{pair}_{q^{(n)}}\left(F'\left(q^{(n)}\right)\right) \in \Delta^*$$

From the initial register values $(u_{I,r'})_{r' \in R'} \in (\Sigma^*)^{R'}$ of $\mathcal{T}'$, we define the final morphism:

$$h' \in \mathrm{Hom}(\Delta^*, \Sigma^*) \qquad \forall r' \in R', \quad \left[h'\left(r', q^{(1)}\right) = u_{I,r'} \quad \text{and} \quad \forall q \neq q^{(1)}, h'(r', q) = \varepsilon\right]$$

Finally, let $\delta'_{\mathrm{st}} : Q \to Q$ and $\delta'_{\mathrm{reg}} : Q \to (R' \to R'^*)$ be the components of the transition function of $\mathcal{T}'$. The morphisms $h_c \in \mathrm{Hom}(\Delta^*, \Delta^*)$ for $c \in \Gamma$ send $(r', q) \in \Delta$ to

$$h_c(r', q) = \mathtt{pair}_{q^{(i_1)}}(\delta'_{\mathrm{reg}}(q^{(i_1)}, c)(r')) \cdot \ldots \cdot \mathtt{pair}_{q^{(i_m)}}(\delta'_{\mathrm{reg}}(q^{(i_m)}, c)(r'))$$

where $i_1 < \ldots < i_m$ and $\{q^{(i_1)}, \dots, q^{(i_m)}\} = \{q^{(?)} \in Q \mid \delta'_{\mathrm{st}}(q^{(?)}, c) = q\}$.

Checking that this HDT0L system computes the right function is a matter of mechanical verification, that has already been carried out in [FR21]. To wrap up the proof, we must justify that it is $(k+1)$-layered. To do so, let us fix a letter $c \in \Gamma$ and two layer indices $i, j \in \{0, \dots, k+1\}$, and count the number $N_{r',q}$ of occurrences of $(r', q) \in \Delta_i$ among all the $h_c(\tilde{r}', \tilde{q})$ for $(\tilde{r}', \tilde{q}) \in \Delta_j$. The letter (r', q) can only appear in $h_c(\tilde{r}', \tilde{q})$ when $\tilde{q} = \delta(q, c)$, and in that case, its occurrences (if any) are in the substring $\mathtt{pair}_q(\delta'_{\mathrm{reg}}(q, c)(\tilde{r}'))$. So $N_{r',q}$ counts the occurrences of $r \in R'_i$ among the $\delta'_{\mathrm{reg}}(q, c)(\tilde{r}')$ for $\tilde{r}' \in R'_j$. Since $\mathcal{T}'$ is a $(k+1)$-layered SST, we are done. □

[7] Except for that, this enumeration of Q is arbitrary. We write $q^{(i)}$ instead of q_i to avoid confusion with the run of an automaton.

PROOF OF ($\Leftarrow$). The translation from HDT0L systems to single-state SSTs mentioned in Remark 2.4.3 is not enough: starting from a $(k+1)$-layered HDT0L system, it gives us a $(k+1)$-layered SST. But we can bring this down to k layers by adding states.

Let $(\Gamma, \Sigma, \Delta, d, (h_c)_{c\in\Gamma}, h')$ be a HDT0L system ($d \in \Delta^*$, $h_c \in \mathrm{Hom}(\Delta^*, \Delta^*)$ for $c \in \Gamma$, and $h \in \mathrm{Hom}(\Delta^*, \Sigma^*)$). Suppose that it is $(k+1)$-layered with $\Delta = \Delta_0 \sqcup \cdots \sqcup \Delta_{k+1}$. This entails that $h_c(\Delta_0) \subseteq \Delta_0^*$, and furthermore that $(h_c)_{\restriction \Delta_0} : \Delta_0 \to \Delta_0^*$ satisfies a copylessness condition, that may succinctly be written as $(h_c)_{\restriction \Delta_0} \in \mathcal{M}^{\mathrm{cl}}_{\Delta_0,\varnothing}$ (cf. Definition 2.3.17).

We define a k-layered SST with:

- $\mathcal{M}^{\mathrm{cl}}_{\Delta_0,\varnothing}$ as the set of states (finite by Proposition 2.3.24), with the identity element of the monoid as its initial state;
- the set of registers $R = \Delta \setminus \Delta_0 = \Delta_1 \sqcup \cdots \sqcup \Delta_{k+1}$, whose i-th layer is the $(i+1)$-th layer of the original HDT0L system ($0 \le i \le k$);
- the initial register contents $(h'(r))_{r\in R}$ – recall that h' is the final morphism;
- the transition function $(\alpha, c) \mapsto (\alpha \bullet (h_c)_{\restriction \Delta_0}, (h'_{\restriction \Delta_0^*} \circ \alpha)^\star \circ (h_c)_{\restriction R})$ where $(-)^\star$ extends functions $\Delta_0 \to \Sigma^*$ into morphisms in $\mathrm{Hom}((\Delta \cup \Sigma)^*, (R \cup \Sigma)^*)$ that map each letter in $R \cup \Sigma$ to itself (since $\Delta = \Delta_0 \sqcup R$, the domain of these morphisms is $(\Delta_0 \sqcup R \sqcup \Sigma)^*$);
- the final output function $\alpha \mapsto (h'_{\restriction \Delta_0^*} \circ \alpha)^\star(d)$.

The layering condition for this SST is inherited from the layering of the original HDT0L system, and one can check the functions computed by the two are the same. □

Corollary 2.4.6. *Every regular function can be computed by a single-state 1-layered SST.*

PROOF. Any regular function is definable by some copyless SST, i.e. 0-layered SST. By Theorem 2.4.5, it can be turned into a 1-layered HDT0L system. The latter can be translated to a single-state SST by the naive construction of Remark 2.4.3. As can readily be seen from the definitions, this construction preserves the 1-layered property. □

The converse does not hold: the single-state 1-layered SST of Example 2.3.4 computes a function which is not regular (cf. Remark 2.3.6). However, 1-layered HDT0L systems – or equivalently 1-layered "simple SSTs", cf. Remark 2.4.3 – *do* characterize regular functions according to Theorem 2.4.5. Thus, there is a one-way transducer model for regular functions that does not use an explicit control state. This is in contrast with copyless SSTs, whose expressivity critically depends on the states (unlike copyful SSTs), cf. Proposition 2.3.9.

2.5. POLYREGULAR FUNCTIONS

Let us now recall the penultimate class of string transductions that we will consider – the last one will be, of course, the new one that we will introduce in Chapter 3.

Definition 2.5.1 (Bojańczyk [Boj18])**.** The class of *polyregular functions* is the smallest class of string-to-string functions closed under composition containing:

- the sequential functions (§2.2);
- the "iterated reverse" function of Example 2.3.7, over any finite alphabet containing #;
- the "squaring with underlining" functions $\mathtt{squaring}_\Gamma : \Gamma^* \to (\Gamma \cup \underline{\Gamma})^*$, for any finite alphabet Γ, illustrated by $\mathtt{squaring}_\Gamma(1234) = \underline{1}234\underline{12}34\underline{123}4\underline{1234}$.

We can state a variant of Definition 2.5.1 which is a bit more convenient for us.

Proposition 2.5.2. *Polyregular functions are the smallest class closed under composition that contains the regular functions and the squaring with underlining functions* $\mathtt{squaring}_\Gamma$.

PROOF. One direction of the equivalence follows from the fact that all regular functions are polyregular. This is stated in the introduction to [Boj18] and can also be recovered from Proposition 2.5.4 below (since copyless SST are 0-layered). For the converse, observe that:

- sequential functions are regular (Proposition 2.3.8);
- since the SST of Example 2.3.7 is copyless, the iterated reverse function is regular. □

The name "polyregular" is justified by:

Proposition 2.5.3. *Every polyregular function f has a polynomially bounded output:*

$$|f(w)| = |w|^{O(1)}$$

Proof. This property is preserved by composition and satisfied by both regular functions (that are linearly bounded) and squaring (which has quadratic growth). □

2.5.1. **Layered SSTs vs polyregular functions.** Let us compare this function class with the transducer model that we have already seen in Section 2.3.3, which also computes functions with polynomial growth.

Proposition 2.5.4. *Every function computed by a layered SST is polyregular.*

Proof. We have already alluded to the simplest way to see this in §1.2.4, and it goes through machine models that we have not properly introduced: it is asserted in [DFG20, §6] – and the reader familiar with the definitions can easily check this – that *k-marble transducers* (mentioned in Theorem 2.3.14), which are equivalent to k-layered SSTs, can be seen as a special case of *pebble transducers* [Boj18, §6] (see also Section 3.2), a device that computes polyregular functions. □

Combining this with Theorem 2.3.14(i) and Proposition 2.5.3, we have:

Corollary 2.5.5. *A string-to-string function can be computed by a layered SST if and only if it is both a polyregular function and an HDT0L transduction.*

The two conditions are not redundant: we have already seen that there exist exponentially growing HDT0L transductions, that cannot be polyregular; conversely, we shall see in the next chapter (to be precise, in Theorem 3.5.1) that some polyregular functions are not HDT0L. Thus, the functions computed by layered SST do not coincide with either of those classes. However, if we take their composition closure, then we get:

Theorem 2.5.6. *Let $f : \Gamma^* \to \Sigma^*$. The following are equivalent:*

(i) *f is polyregular;*
(ii) *f can be obtained as a composition of layered SSTs;*
(iii) *f can be obtained as a composition of single-state 1-layered SSTs.*

As an immediate consequence:

Corollary 2.5.7 (claimed in [DFG20, §6])**.** *Layered SSTs are* not *closed under composition.*

Proof of Theorem 2.5.6 (i) ⇒ (iii). Thanks to Proposition 2.5.2, we know that any polyregular functions can be written as a composition of a sequence of functions, each of which is either regular or equal to $\texttt{squaring}_\Gamma$ for some finite alphabet Γ. It suffices to show that each function in the sequence can in turn be expressed as a composition of single-state 1-layered SSTs. We decompose $\texttt{squaring}_\Gamma$ as

$$1234 \mapsto \underline{4}321\underline{3}21\underline{2}1\underline{1} \mapsto \underline{1}234\underline{1}23\underline{4}12\underline{3}41\underline{2}34\underline{1}$$

The first step is performed by the SST of Example 2.3.4, which has a single state and, as mentioned in Section 2.5, is 1-layered. The second step can be implemented using a SST with a single state q (that we omit below for readability), two registers X (at layer 0) and Y (at layer 1) with empty initial values, an output function $F(q) = Y$, and

$$\forall c \in \Gamma, \quad \delta(c) = (X \mapsto X, Y \mapsto cY) \quad \text{and} \quad \delta(\underline{c}) = (X \mapsto cX, Y \mapsto \underline{c}XY)$$

As for regular functions, Corollary 2.4.6 takes care of them. □

Proof of Theorem 2.5.6 (iii) ⇒ (ii). Immediate by definition. □

Proof of Theorem 2.5.6 (ii) ⇒ (i). Proposition 2.5.4 applies in particular to single-state 1-layered SSTs. Therefore, their composition is also polyregular (indeed, according to Definition 2.5.1, polyregular functions are closed under composition). □

2.5.2. **Polynomial list functions.** Let us also report briefly on the functional formalism for polyregular functions [Boj18, Section 4] that we mentioned in Section 1.3.3. The explanation is geared towards a reader familiar with typed λ-calculi.

The λ-terms defining the *polynomial list functions* are generated by the grammar of simply typed λ-terms enriched with constants, whose meaning can be specified by extending the naive set-theoretic semantics from Section 1.1.7 (one can also extend the β-reduction rules correspondingly). For instance, given a *finite* set S and $a \in S$, every element of S can be used as a constant, another allowed constant is $\mathtt{is}_a^S$ and we have

$$[\![\mathtt{is}_a^S\ t]\!] = \mathtt{true} \quad \text{if } a = [\![t]\!] \qquad [\![\mathtt{is}_a\ t]\!] = \mathtt{false} \quad \text{if } [\![t]\!] \in S \setminus \{a\}$$

The grammar of simple types and the typing rules are also extended accordingly. For instance, any finite set S induces a type also written S, such that every element $a \in S$ corresponds to a term $a : S$ of this type with $[\![a]\!] = a$. There are also operations expressing the cartesian product ($\times$) and disjoint union ($+$) of two types; and, for any type τ, there is a type τ^* of lists whose elements are in τ. So the constant $\mathtt{is}_a^S$ receives the type

$$\mathtt{is}_a^S : S \to \{\mathtt{true}\} + \{\mathtt{false}\} \qquad \text{for any finite set } S$$

See [Boj18, §4] for the other primitive operations that are added to the simply typed λ-calculus in order to define polynomial list functions; we make use of `is`, `case`, `map` and `concat` here. The `map` combinator, whose name follows the usual functional programming conventions,[8] expresses the functoriality of the type constructor $(-)^*$ as follows:

$$[\![\mathtt{map}\ f\ t]\!] = [[\![f]\!](x_1), \dots, [\![f]\!](x_n)] \quad \text{when} \quad [\![t]\!] = [x_1, \dots, x_n]$$

Bojańczyk's result is that if Γ and Σ are finite sets, then the polynomial list functions of type $\Gamma^* \to \Sigma^*$ correspond exactly the polyregular functions.

2.5.3. **More on the "map" combinator.** The higher-order function `map` above also admits an obvious counterpart as an operator on string-to-string functions. First, note that given a word $w \in (\Gamma \cup \{\#\})^*$ and assuming that $\# \notin \Gamma$, there exists a unique decomposition

$$w = w_1 \# \dots \# w_n \qquad \text{where} \quad w_1, \dots, w_n \in \Gamma^*$$

Definition 2.5.8. Let $f : \Gamma^* \to \Sigma^*$ and suppose that $\# \notin \Gamma \cup \Sigma$. We define the function

$$\mathbf{map}(f) : w_1 \# \dots \# w_n \in (\Gamma \cup \{\#\})^* \mapsto f(w_1) \# \dots \# f(w_n) \in (\Sigma \cup \{\#\})^*$$

Proposition 2.5.9. *If f is an HDT0L transduction, then so is* $\mathbf{map}(f)$. *For each $k \geq 1$, the functions that can be computed by k-layered HDT0L systems are also closed under* **map**.

Proof. Let $(\Gamma, \Sigma, \Delta, d, (h_c)_{c \in \Gamma}, h')$ be a HDT0L system computing $f : \Gamma^* \to \Sigma^*$. We define below a HDT0L system that computes $\mathbf{map}(f) : (\Gamma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$.

- The intermediate alphabet is $\widehat{\Delta} = \Delta \cup \Sigma \cup \{\#, X\}$, assuming w.l.o.g. that $\# \notin \Delta \cup \Sigma$, where $X \notin \Delta \cup \Sigma \cup \{\#\}$ is an arbitrarily chosen fresh letter.
- The starting word is $Xd \in \widehat{\Delta}^*$.
- For $c \in \Gamma$, we extend h_c into $\widehat{h}_c \in \mathrm{Hom}(\Delta^*, \Delta^*)$ by setting $\widehat{h}_c(x) = x$ for $x \in \Sigma \cup \{\#, X\}$. Since the input alphabet is now $\Gamma \cup \{\#\}$, we also define the morphism $\widehat{h}_\#$ as the extension of h' (using $\Sigma \subset \widehat{\Delta}$) such that $\widehat{h}_\#(X) = Xd\#$ and $\widehat{h}_\#(x) = x$ for $x \in \Sigma \cup \{\#\}$.

[8] In the case of "map", the Lisp and ML traditions agree, which is not the case for "reduce" vs "fold".

- The final morphism $\widehat{h'}$ extends h' with $\widehat{h'}(X) = \varepsilon$ and $\widehat{h'}(x) = x$ for $x \in \Sigma \cup \{\#\}$.

This shows that HDT0L transductions are closed under **map**.

We now prove that for any $k \in \mathbb{N}_{\geq 1}$, this closure property holds for k-layered HDT0L transductions (so, in particular, for regular functions by taking $k = 1$). Suppose that f is computed by a k-layered HDT0L system with intermediate alphabet Δ and initial word $d \in \Delta^*$. One can build a k-layered HDT0L system which computes the same function f and such that *the initial word contains at most one occurrence of each letter*; the idea is to replace Δ and $d = d_1 \dots d_n$ by $\Delta \times \{1, \dots, n\}$ and $(d_1, 1), \dots, (d_n, n)$ where $n = |d|$, and to adapt the morphisms accordingly. Applying the above construction then results in a k-layered HDT0L system that computes $\mathbf{map}(f)$; note that if we did not have this property for the inital word, we would get a $(k+1)$-layering instead. □

Corollary 2.5.10. *Regular and polyregular functions are both closed under* **map**.

PROOF. For regular functions, apply Proposition 2.5.9 with $k = 0$. For polyregular functions, use Theorem 2.5.6 and the elementary property $\mathbf{map}(f \circ g) = \mathbf{map}(f) \circ \mathbf{map}(g)$. □

Of course, this is also a consequence of the fact that the `map` combinator appears in polynomial list functions (§2.5.2) as well as in the regular list functions [BDK18]. Conversely, if a class of string-to-string functions is *not* closed under **map**, then this means that one probably cannot hope to capture it using this kind of list functions formalism. We will see in the next chapter that this will be the case for comparison-free polyregular functions.

2.6. TREE TRANSDUCERS

The notion of *regular tree-to-tree function* is defined by generalizing the characterization of regular string functions by Monadic Second-Order Logic [EM99; BE00; EH01], in a way that is compatible with the isomorphism of Remark 2.1.1. There are two orthogonal difficulties that have to be overcome to extend copyless streaming *string* transducers (§2.3.2) to a machine model for regular tree functions: one comes from producing trees as output, while the other comes from taking trees as input.

BRTTs [AD17] (and the similar model of register tree transducers in [BD20, §4]) provide solutions for both. To explain the acronym: the name "streaming tree transducer" is used in [AD17] for a transducer model operating over unranked trees (in which the number of children of a node is not determined by its label, and might in fact be arbitrarily large); *bottom-up ranked tree transducers* are proposed in the same paper as a simpler, equally expressive variant for the special case of ranked trees. In this dissertation it might be more fitting to think of "BRTT" as standing for "bottom-up *register* tree transducer", as the title of Section 2.6.3 suggests.

In the next two subsections, we give a high-level overview of the challenges posed by trees, while Section 2.6.3 describes BRTTs more formally.

2.6.1. **Trees as output.** String-to-tree regular functions require a modification of the kind of data stored in the registers of an SST. Tree-valued registers are *not enough*, for the following reasons: to recover the flexibility of string concatenation, one should be able to perform operations such as grafting the root of some tree to a leaf of another tree; but then the latter should be a tree with a distinguished leaf, serving as a "hole" waiting to be substituted by a tree. (This is fundamental in the theory of forest algebras, which proposes various counterparts for trees to the monoid of strings with concatenation, see [Pin21a, Chapter 22].) By allowing both trees and "one-hole trees" as register values, with the appropriate notion of copyless register assignment, one gets the *copyless streaming string-to-tree transducers*, whose expressive power corresponds exactly to the regular tree functions [AD17, Theorem 3.16].

2.6.2. **Trees as input.** To compute tree-to-tree regular functions, the first idea would be to blend the notion of copyless SST with the classical bottom-up *tree automata*. One would then get *copyless bottom-up ranked tree transducers*. However, this model is believed to be too weak to express all regular tree functions (even in the case of tree-to-string functions). An explicit counterexample is conjectured in [AD17, §2.3], in the case of regular functions on unranked trees; we adapt it here into a function from ranked trees to strings.

In the example below, for a ranked letter a of arity $2 = \{\triangleleft, \triangleright\}$, we use the abbreviation $a(t,u)$ for $a(\triangleleft \mapsto t, \triangleright \mapsto u)$.

Example 2.6.1 ("Conditional swap")**.** Define $f : \mathbf{Tree}(\{a:2, b:2, c:\varnothing\}) \to \{a,b,c\}^*$ by

$$f(a(t,u)) = f(u) \cdot a \cdot f(t) \qquad f(t) = \texttt{inorder}(t) \text{ if } t \text{ doesn't match the previous pattern}$$

where `inorder` prints the nodes of t following a depth-first in-order traversal. In other words, $f = \texttt{inorder} \circ g$ where $g(a(t,u)) = a(g(u), g(t))$ and $g(t) = t$ otherwise (i.e. when the root of t is either b or c).

Conjecture 2.6.2 (adapted from [AD17, §2.3])**.** The above f cannot be computed by a copyless BRTT.

One must then allow more register assignments than the copyless ones. This cannot be done haphazardly, for arbitrary assignments would lead to a much larger class of functions than regular tree functions (as the case of copyful SST already witnesses, cf. §2.3.1). Alur and D'Antoni call their relaxed condition [AD17] the *single use restriction*; the following single-state BRTT for f provides a typical example of the new possibilities allowed.

Example 2.6.3 (Non-copyless BRTT for conditional swap)**.** Take $R = \{x, y\}$, initialized at the c-labeled leaves with $(x \mapsto c,\ y \mapsto c)$. At a subtree $a(u,v)$, we need to combine the registers $x_\triangleleft, y_\triangleleft$ (resp. $x_\triangleright, y_\triangleright$) coming from the left (resp. right) child u (resp. v) to produce the values of the registers x, y at this node: this is performed by a register assignment

$$t_a \in [\{x_\triangleleft, y_\triangleleft, x_\triangleright, y_\triangleright\} \to_{\mathcal{SR}} \{x,y\}] \qquad t_a(x) = x_\triangleright a x_\triangleleft \qquad t_a(y) = y_\triangleleft a y_\triangleright$$

The idea is that the register values produced by processing a subtree u are $f(u)$ for x and $\texttt{inorder}(u)$ for y. The register transition for a b-labeled node is then $t_b(x) = t_b(y) = y_\triangleleft b y_\triangleright$, reflecting the fact that $f(b(u,v)) = \texttt{inorder}(b(u,v))$.

This t_b is not copyless since $y_\triangleleft$ occurs twice: once in $t_b(x)$ and once in $t_b(y)$. The observation at the heart of the single use restriction is that the values of x and y for a given subtree can never be combined in the same expression in the remainder of the BRTT's run, so that allowing this duplication of $y_\triangleleft$ will never lead to having two copies of $y_\triangleleft$ inside the value of a single register.

2.6.3. **Bottom-up (ranked|register) tree transducers.** We give here a self-contained definition of the notion of BRTT corresponding to regular tree functions, as they were designed in [AD17]. Since the paper [AD17] is mainly concerned with transducers over unranked trees, the information concerning the definition of BRTTs is spread over its sections 2.1, 3.7 and 3.8. There is also a difference with our general formalism for ranked trees: BRTTs are only defined over *binary* trees in [AD17] – but the extension to ranked trees is obvious, and we will perform it in Chapter 6 when we give a categorical account of BRTTs. For now, we stay faithful to our source by working with binary trees.

Definition 2.6.4 ([AD17, p. 31:36])**.** The set $\mathrm{BinTree}(\Sigma)$ of *binary trees* over the alphabet Σ, and the set $\partial\mathrm{BinTree}(\Sigma)$ of *one-hole binary trees* are generated by the respective grammars

$$T, U ::= \lceil\cdot\rceil \mid a\lceil T, U\rceil \quad (a \in \Sigma) \qquad\qquad T', U' ::= \square \mid a\lceil T', U\rceil \mid a\lceil T, U'\rceil \quad (a \in \Sigma)$$

That is, $\mathrm{BinTree}(\Sigma)$ consists of binary trees whose leaves are all equal to $\langle\rangle$ and whose nodes are labeled with letters in Σ. As for $\partial\mathrm{BinTree}(\Sigma)$, it contains trees with exactly one leaf labeled $\Box$ instead of $\lceil\cdot\rceil$. This "hole" $\Box$ is intended to be substituted by a tree: for $T' \in \partial\mathrm{BinTree}(\Sigma)$ and $U \in \mathrm{BinTree}(\Sigma)$, $T'[U]$ denotes T' where $\Box$ has been replaced by U.

Definition 2.6.5 ([AD17, p. 31:40])**.** The *binary tree expressions* (E, F below, forming the set $\mathrm{ExprBT}(\Sigma, V, V')$) and *one-hole binary tree expressions* (E', F' below, forming the set $\mathrm{Expr}\partial\mathrm{BT}(\Sigma, V, V')$) over the variable sets V and V' are generated by the grammar (with $x \in V$, $x' \in V'$ and $a \in \Sigma$)

$$E, F ::= \lceil\cdot\rceil \mid x \mid a\lceil E, F\rceil \mid E'[F] \qquad E', F' := \Box \mid x' \mid a\lceil E', F\rceil \mid a\lceil E, F'\rceil \mid E'[F']$$

Given $\rho : V \to \mathrm{BinTree}(\Sigma)$ and $\rho' : V' \to \partial\mathrm{BinTree}(\Sigma)$, one defines $E(\rho, \rho') \in \mathrm{BinTree}(\Sigma)$ for $E \in \mathrm{ExprBT}(\Sigma)$ and $E'(\rho, \rho') \in \partial\mathrm{BinTree}(\Sigma)$ for $E \in \mathrm{Expr}\partial\mathrm{BT}(\Sigma)$ in the obvious way.

Definition 2.6.6 ([AD17, §3.7])**.** Let us fix an input alphabet Γ and output alphabet Σ. The set of *register assignments* over two disjoint sets R, R', whose elements are called *registers*, is

$$\mathcal{A}(\Sigma, R, R') = \mathrm{ExprBT}(\Sigma, R_{\triangleleft\triangleright}, R'_{\triangleleft\triangleright})^R \times \mathrm{Expr}\partial\mathrm{BT}(\Sigma, R_{\triangleleft\triangleright}, R'_{\triangleleft\triangleright})^{R'} \quad \text{where } R_{\triangleleft\triangleright} = R \times \{\triangleleft, \triangleright\}$$

A *register tree transducer* consists of a *finite* set Q of *states* with an *initial state* $q_I \in Q$, two disjoint *finite* sets R, R' of *registers*, a *transition function* $\delta : Q \times Q \times \Gamma \to Q \times \mathcal{A}(\Sigma, R, R')$ and an *output function* $F : Q \to \mathrm{ExprBT}(\Sigma, R, R')$. To each tree $T \in \mathrm{BinTree}(\Gamma)$, it associates inductively a *configuration* $\mathrm{Conf}(T) \in Q \times \mathrm{BinTree}(\Sigma)^R \times \partial\mathrm{BinTree}(\Sigma)^{R'}$:

- The base case is $\mathrm{Conf}(\lceil\cdot\rceil) = (q_I, (r \mapsto \lceil\cdot\rceil), (r' \mapsto \Box))$.
- When $\mathrm{Conf}(T) = (q_\triangleleft, \rho_\triangleleft, \rho'_\triangleleft)$, $\mathrm{Conf}(U) = (q_\triangleright, \rho_\triangleright, \rho'_\triangleright)$ and $\delta(c, q_\triangleleft, q_\triangleright) = (q, (\varepsilon, \varepsilon'))$, we set $\mathrm{Conf}(c\lceil T, U\rceil) = (q, (r \mapsto \varepsilon(r)(\rho, \rho')), (r' \mapsto \varepsilon'(r')(\rho, \rho')))$ where $\rho(r, d) = \rho_d(r)$ for $(r, d) \in R \times \{\triangleleft, \triangleright\}$ and similarly for ρ'.

The function defined by the transducer is $T \in \mathrm{BinTree}(\Gamma) \mapsto F(q_{\mathrm{fin}}(T))(\rho_{\mathrm{fin}}(T), \rho'_{\mathrm{fin}}(T))$ where $(q_{\mathrm{fin}}(T), \rho_{\mathrm{fin}}(T), \rho'_{\mathrm{fin}}(T)) = \mathrm{Conf}(T)$ (recall that F is the output function).

Example 2.6.7 (illustrated by Figure 2.6.1)**.** Let us consider a transducer over the alphabets $\Gamma = \Sigma = \{a, b\}$, with a single state ($|Q| = 1$) and two registers, both tree-valued (so $R = \{r_1, r_2\}$ and $R' = \varnothing$). This simplifies the transition function δ into a function $\Gamma \to \mathrm{ExprBT}(\Sigma, R_{\triangleleft\triangleright}, \varnothing)^R$ – equivalently, we will consider $\delta : \Gamma \times R \to \mathrm{ExprBT}(\Sigma, R_{\triangleleft\triangleright}, \varnothing)$.

We take $\delta(c, r_1) = c\lceil(r_1)_\triangleleft, (r_1)_\triangleright\rceil$ and $\delta(c, r_2) = c\lceil(r_2)_\triangleright, (r_2)_\triangleleft\rceil$ for $c \in \{a, b\}$, where $r_\triangleleft$ is a notation for $(r, \triangleleft) \in R_{\triangleleft\triangleright} = R \times \{\triangleleft, \triangleright\}$. If we write $\hat{r}_i(T)$ ($i \in \{1, 2\}$) for the contents of the register r_i at the end of a run of the transducer on $T \in \mathtt{BT}(\Gamma)$, then this δ translates into:

$$\hat{r}_1(c\lceil T, U\rceil) = c\lceil\hat{r}_1(T), \hat{r}_1(U)\rceil \qquad \hat{r}_2(c\lceil T, U\rceil) = c\lceil\hat{r}_2(U), \hat{r}_2(T)\rceil \qquad (c \in \{a, b\})$$

And the initial condition is $\hat{r}_1(\lceil\cdot\rceil) = \hat{r}_2(\lceil\cdot\rceil) = \lceil\cdot\rceil$. Therefore $\hat{r}_1(T) = T$ and $\hat{r}_2(T)$ is T "mirrored" by exchanging left and right; let us write $\hat{r}_2(T) = \mathtt{reverse}(T)$.

The output function F is also simplified into an expression in $\mathrm{ExprBT}(\Sigma, R, \varnothing)$. By taking $F = a\lceil r_1, r_2\rceil$, we define a transducer computing the regular function $T \mapsto a\lceil T, \mathtt{reverse}(T)\rceil$.

The "single use restriction" that must be imposed on the register assignments is not intrinsic: it depends on an additional piece of data, namely a binary relation between the registers called *conflict*.

Definition 2.6.8 ([AD17, §2.1])**.** A *conflict relation* is a reflexive and symmetric relation.

An expression $E \in \mathrm{ExprBT}(\Sigma, V, V') \cup \mathrm{Expr}\partial\mathrm{BT}(\Sigma, V, V')$ is said to be *consistent with* a conflict relation $\asymp$ over $V \cup V'$ when each variable in $V \cup V'$ appears at most once in E, and for all $x, y \in V \cup V'$, if $x \neq y$ and $x \asymp y$, then E does not contain both x and y.

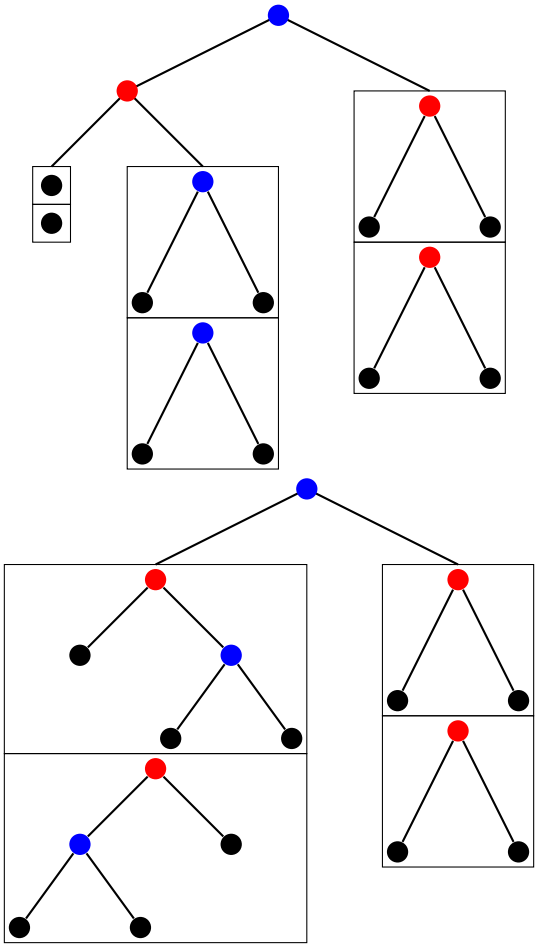


FIGURE 2.6.1. Two consecutive steps of the run of the bottom-up ranked tree transducer of Example 2.6.7 over a tree whose alphabet of node labels is $\Sigma = \{\bullet, \bullet\} \cong \{a, b\}$.
The configuration at each subtree is represented by two boxes; the top (resp. bottom) box displays the contents of r_1 (resp. r_2). (The single state is omitted from the visual representation of the configuration.)

A register assignment $(\varepsilon, \varepsilon') \in \mathcal{A}(\Sigma, R, R')$ is *single use restricted* with respect to a conflict relation $\asymp$ over $R \cup R'$ when:

- all $\varepsilon(r)$ for $r \in R$ and all $\varepsilon'(r')$ for $r' \in R'$ are consistent with $\asymp$;
- if $x_1, x_2, y_1, y_2 \in R \cup R'$, $x_1 \asymp x_2$ and, for some $d \in \{\triangleleft, \triangleright\}$, (x_1, d) appears in[9] $(\varepsilon \cup \varepsilon')(y_1)$ and (x_2, d) appears in $(\varepsilon \cup \varepsilon')(y_2)$, then $y_1 \asymp y_2$ (note that this includes the case $x_1 = x_2$).

Definition 2.6.9. A *bottom-up ranked tree transducer (BRTT)* is a register tree transducer $(Q, q_I, R, R', \delta, F)$ endowed with a conflict relation $\asymp$ on $R \cup R'$, such that $F(q)$ is consistent with $\asymp$ and all register assignments in the image of δ are single use restricted w.r.t. $\asymp$.

A *regular tree function* is a function computed by a BRTT.

When the conflict relation is trivial (i.e. coincides with equality), we say that the BRTT is *copyless*. We also say that a register tree transducer is copyless if it becomes a BRTT when endowed with a trivial conflict relation.

[9] By $\varepsilon \cup \varepsilon'$ we mean the map on the *disjoint* union $R \cup R'$ induced in the obvious way by ε and ε'.

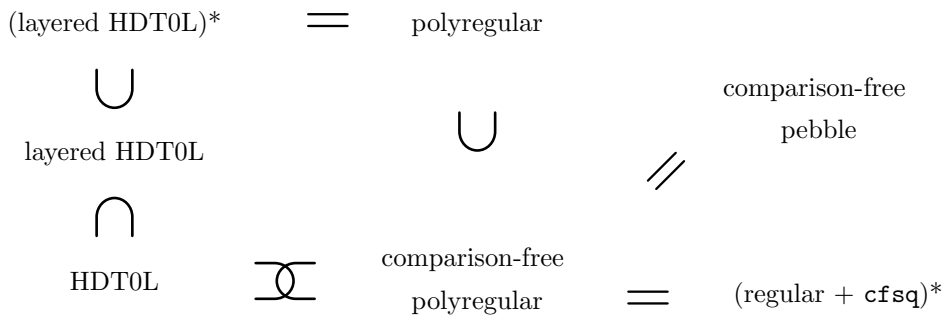
CHAPTER 3

Comparison-free polyregular functions

At last, we shall now introduce the new class of *comparison-free polyregular functions* (or "cfp functions" for short). The simplest way to define them is to start from the regular functions and combine them using a "composition by substitution" (CbS) operation (§3.1). In order to relate this to the preexisting literature, we give a machine model for cfp functions (§3.2), obtained by putting a restriction on *pebble transducers* which compute polyregular functions (cf. §2.5).

After this, we give two more difficult characterizations of cfp functions in Section 3.3. The first one is analogous to Proposition 2.5.2, and part of the statement is that cfp functions are closed under composition – a property that was a major focus of the discussion in §1.2.4.) The second one characterizes comparison-free polyregular functions *of growth* $O(n^k)$ for each $k \in \mathbb{N}$ with $k \geq 1$; we will say more about this below. We also prove in Section 3.4 a "closed form" description of cfp functions *with unary inputs*.

Finally, we apply many of those previous results to obtain separation results between cfp functions and some other transduction classes described in Chapter 2. With those results, the known relationships between the function classes with superlinear growth studied in this manuscript are as follows:[1]



One of the properties that we prove on cfp functions of growth $O(n^k)$ is that they can always be computed by comparison-free pebble transducers with k pebbles. This is a *pebble minimization* result, in the vein of those proved by Douéneau-Tabot et al. [DFG20] for marble transducers (cf. Theorem 2.3.14) and by Lhote [Lho20] for general pebble transducers. But we recently realized that the latter is actually *wrong*; the counterexample and its proof will be shown in the upcoming paper[2] [Boj+21]. That said, Lhote's approach is far from worthless: we were able to reuse it for our proof of the comparison-free pebble minimization theorem. In fact, the familiarity that we acquired with [Lho20] in the process of adapting its arguments later helped us discover this mistake. Incidentally, the counterexample to pebble minimization also refutes the logical characterization of cfp functions conjectured in [NNP21, §10]. (Since it turned out to be false, we do not reproduce its statement here.)

[1] We write C^* for the composition closure of the class C. Inclusions denoted by $\subset$ are strict, and $\not\supset\not\subset$ means that there is no inclusion either way.

[2] Joint work with Mikołaj Bojańczyk, Gaëtan Douéneau-Tabot, Sandra Kiefer and Pierre Pradic.

3.1. Composition by substitution

Definition 3.1.1. Let $f : \Gamma^* \to I^*$, and for each $i \in I$, let $g_i : \Gamma^* \to \Sigma^*$. The *composition by substitution* of f with the family $(g_i)_{i\in I}$ is the function

$$\mathrm{CbS}(f, (g_i)_{i\in I}) \ : \ w \ \mapsto \ g_{i_1}(w) \dots g_{i_k}(w) \text{ where } i_1 \dots i_k = f(w)$$

That is, we first apply f to the input, then every letter i in the result of f is substituted by the image of the original input by g_i. Thus, $\mathrm{CbS}(f, (g_i)_{i\in I})$ is a function $\Gamma^* \to \Sigma^*$.

Definition 3.1.2. The smallest class of string-to-string functions closed under CbS and containing all regular functions is called the class of *comparison-free polyregular functions*.

Example 3.1.3. The following variant of "squaring with underlining" (cf. Definition 2.5.1) is comparison-free polyregular: $\mathtt{cfsquaring}_\Gamma : \mathtt{123} \in \Gamma^* \mapsto \underline{1}123\underline{2}123\underline{3}123 \in (\Gamma \cup \underline{\Gamma})^*$.

Indeed, it can be expressed as $\mathtt{cfsquaring}_\Gamma = \mathrm{CbS}(f, (g_i)_{i\in I})$ where $I = \Gamma \cup \{\#\}$, the function $f : w_1 \dots w_n \mapsto w_1\# \dots w_n\#$ is regular (more than that, a morphism between free monoids) and $g_\# = \mathrm{id}$, $g_c : w \mapsto \underline{c}$ for $c \in \Gamma$ are also regular. Its growth rate is quadratic, while regular functions have at most linear growth. Other examples that also require a single composition by substitution are given in Theorem 3.5.1.

We can already justify the latter half of the name of our new class.

Proposition 3.1.4. *Polyregular functions are closed under composition by substitution.*

Corollary 3.1.5. *Every comparison-free polyregular function is, indeed, polyregular.*

Proof of Proposition 3.1.4. We use polynomial list functions (Section 2.5.2), though *for-transducers* [Boj18, §3] could also be used for an easy proof.

First, we claim that for any $I = \{i_1, \dots, i_{|I|}\}$, there exists a polynomial list function $\mathtt{match}^{I,\tau} : I \to \tau \to \dots \to \tau \to \tau$ that returns its $(k+1)$-th argument when its first argument is i_k. This can be shown by induction on $|I|$, using $\mathtt{is}_i^I$ $(i \in I)$ and $\mathtt{case}^{\{\mathtt{true}\},\{\mathtt{false}\},\tau}$.

Next, let $f : \Gamma^* \to I^*$, and for $i \in I$, $g_i : \Gamma^* \to \Sigma^*$ be polyregular functions. Assuming that f and g_i $(i \in I)$ are defined by polynomial list functions of the same name, the λ-term

$$\lambda w.\ \mathtt{concat}^{\Sigma}\ (\mathtt{map}^{I,\Sigma^*}\ (\lambda i.\ \mathtt{match}^{I,\Sigma^*}\ i\ (g_{i_1}\ w)\ \dots\ (g_{i_{|I|}}\ w))\ (f\ w))$$

computes $\mathrm{CbS}(f, (g_i)_{i\in I})$. □

Fundamentally, Definition 3.1.2 is inductive: it considers the functions generated from the base case of regular functions by applying compositions by substitution. The variant below with more restricted generators is sometimes convenient.[3]

Definition 3.1.6. A string-to-string function is said to be:

- of *rank at most* 0 if it is regular;
- of *rank at most* $k+1$ *(for* $k \in \mathbb{N}$*)* if it can be written as $\mathrm{CbS}(f, (g_i)_{i\in I})$ where $f : \Gamma^* \to I^*$ is *regular* and each $g_i : \Gamma^* \to \Sigma^*$ is of rank at most k.

Proposition 3.1.7. *A function f is comparison-free polyregular if and only if there exists some $k \in \mathbb{N}$ such that f has rank at most k. In that case, we write* $\mathrm{rk}(f)$ *for the least such k and call it the* rank *of f. If $(g_i)_{i\in I}$ is a family of comparison-free polyregular functions,*

$$\mathrm{rk}(\mathrm{CbS}(f, (g_i)_{i\in I})) \leq 1 + \mathrm{rk}(f) + \max_{i\in I} \mathrm{rk}(g_i)$$

Proof. This is equivalent to claiming that the smallest class $\mathcal{C}$ of functions such that

[3] This discussion does not use any of the property of regular functions; the same would apply to any class of function defined as a closure under CbS.

- every regular function is in $\mathcal{C}$,
- and $\mathrm{CbS}(f,(g_i)_{i\in I})\in\mathcal{C}$ for any regular $f:\Gamma^*\to I^*$ and any $(g_i:\Gamma^*\to\Sigma^*)_{i\in I}\in\mathcal{C}^I$,

contains all comparison-free polyregular functions. It suffices to show that $\mathcal{C}$ is closed under composition by substitution, which can be done by induction using the equation

$$\mathrm{CbS}(\mathrm{CbS}(f,(g_i)_i),(h_j)_j)=\mathrm{CbS}(f,(\mathrm{CbS}(g_i,(h_j)_j))_i)$$

The same equation explains the inequality on the rank that we claim in the proposition. □

A straightforward consequence of this definition is that, just like regular functions (see Proposition 2.3.11), cfp functions are closed under *regular conditionals* and concatenation.

Proposition 3.1.8. *Let $f,g:\Gamma^*\to\Sigma^*$ be comparison-free polyregular functions and $L\subseteq\Gamma^*$ be a regular language. The function that coincides with f on L and with g on $\Gamma^*\setminus L$ is cfp, and so is $w\in\Gamma^*\mapsto f(w)\cdot g(w)$; both have rank at most* $\max(\mathrm{rk}(f),\mathrm{rk}(g))$.

Proof. Let $f,g:\Gamma^*\to\Sigma^*$ be cfp with $\max(\mathrm{rk}(f),\mathrm{rk}(g))=k\geq 1$ (otherwise, we just apply Proposition 2.3.11). It means that we have regular functions f' and g', as well as families of functions $(f''_i)_{i\in I}$ and $(g''_j)_{j\in J}$ of rank at most $k-1$ such that

$$f\;=\;\mathrm{CbS}(f',(f''_i)_{i\in I})\qquad\text{and}\qquad g\;=\;\mathrm{CbS}(g',(g''_j)_{j\in J})$$

Assume without loss of generality that $I\cap J=\varnothing$.

Let us first treat regular conditionals. Using Proposition 2.3.11 applied to the functions $f',g':\Gamma^*\to(I\cup J)^*$, the function $h':\Gamma^*\to(I\cup J)^*$ that coincides with f' over L and g' over $L\setminus\Gamma^*$ is regular. Setting $(h''_k)_{k\in I\cup J}$ to be the family of functions such that $h''_i=f''_i$ for $i\in I$ and $h''_j=g''_j$ for $j\in J$, we obtain a cfp function $h=\mathrm{CbS}(h',(h''_k)_{k\in I\cup J})$ corresponding to the desired conditional. Since all the h''_k are of rank at most $k-1$, we also get the result on the rank.

Concerning concatenation, one check that the pointwise concatenation $f\cdot g$ is equal to $\mathrm{CbS}(f'\cdot g',(h''_k)_{k\in I\cup J})$, which is cfp since $f'\cdot g'$ is regular (by Proposition 2.3.11 again). □

3.2. Comparison-free pebble transducers

We now characterize our function class by a machine model that will explain our choice of the adjective "comparison-free", as well as the operational meaning of the notion of rank we just defined. However, the definition using composition by substitution will remain our tool of choice to prove further properties, so the next sections do not depend on this one.

Definition 3.2.1. Let $k\in\mathbb{N}$ with $k\geq 1$. Let Γ,Σ be finite alphabets and $\triangleright,\triangleleft\notin\Gamma$.

A *k-pebble stack* on an input string $w\in\Gamma^*$ consists of an ordered list of p positions in the string $\triangleright w\triangleleft$ (i.e. of p integers between 1 and $|w|+2$) for some $p\in\{1,\dots,k\}$. We therefore write $\mathrm{Stack}_k=\mathbb{N}^0\cup\mathbb{N}^1\cup\cdots\cup\mathbb{N}^k$, keeping in mind that given an input w, we will be interested in "legal" values bounded by $|w|+2$.

A *comparison-free k-pebble transducer (k-CFPT)* consists of a finite set of states Q, an initial state $q_I\in Q$ and a family of transition functions

$$Q\times(\Gamma\cup\{\triangleright,\triangleleft\})^p\to Q\times(\mathbb{N}^p\to\mathrm{Stack}_k)\times\Sigma^*\quad\text{for } 1\leq p\leq k$$

where the $\mathbb{N}^p$ on the left is considered as a subset of Stack_k. For a given state and given letters $(c_1,\dots,c_p)\in(\Gamma\cup\{\triangleright,\triangleleft\})^p$, the allowed values for the stack update function $\mathbb{N}^p\to\mathrm{Stack}_k$ returned by the transition function are:

$$\begin{array}{lrcl}
\text{(identity)} & (i_1,\dots,i_p) & \mapsto & (i_1,\dots,i_p) \in \mathbb{N}^p \\
\text{(move left, only allowed when } c_p \neq \triangleright\text{)} & (i_1,\dots,i_p) & \mapsto & (i_1,\dots,i_p-1) \in \mathbb{N}^p \\
\text{(move right, only allowed when } c_p \neq \triangleleft\text{)} & (i_1,\dots,i_p) & \mapsto & (i_1,\dots,i_p+1) \in \mathbb{N}^p \\
\text{(push, only allowed when } p \leq k-1\text{)} & (i_1,\dots,i_p) & \mapsto & (i_1,\dots,i_p,1) \in \mathbb{N}^{p+1} \\
\text{(pop, only allowed when } p \geq 1\text{)} & (i_1,\dots,i_p) & \mapsto & (i_1,\dots,i_{p-1}) \in \mathbb{N}^{p-1}
\end{array}$$

(Note that the codomains of all these functions are indeed subsets of Stack_k.)

The run of a CFPT over an input string $w \in \Gamma^*$ starts in the initial configuration comprising the initial state q_I, the initial k-pebble stack $(1) \in \mathbb{N}^1$, and the empty string as an initial output log. As long as the current stack is non-empty, a new configuration is computed by applying the transition function to q and to $((\triangleright w \triangleleft)[i_1],\dots,(\triangleright w \triangleleft)[i_p])$ where $(i_1,\dots,i_p)$ is the current stack; the resulting stack update function is applied to $(i_1,\dots,i_p)$ to get the new stack, and the resulting output string in Σ^* is appended to the right of the current output log. If the CFPT ever terminates by producing an empty stack, the *output associated to w* is the final value of the output log.

This amounts to restricting in two ways[4] the *pebble transducers* from [Boj18, §2]:

- in a general pebble transducer, one can *compare positions*, i.e. given a stack $(i_1,\dots,i_p)$, the choice of transition can take into account whether[5] $i_j \leq i_{j'}$ (for any $1 \leq j, j' \leq p$);
- in a "push", new pebbles are initialized to the leftmost position ($\triangleright$) for a CFPT, instead of starting at the same position as the previous top of the stack (the latter would ensure the equality of two positions at some point; it is therefore an implicit comparison that we must relinquish to be truly "comparison-free").

This limitation is similar to (but goes a bit further than) the "invisibility" of pebbles in a transducer model introduced by Engelfriet et al. [EHS21] (another difference, unrelated to comparisons, is that their transducers use an unbounded number of invisible pebbles).

Remark 3.2.2. Our definition guarantees that "out-of-bounds errors" cannot happen during the run of a comparison-free pebble transducer. The sequence of successive configurations is therefore always well-defined. But it may be infinite, that is, it may happen that the final state is never reached. Thus, a CFPT defines a *partial* function.

That said, the set of inputs for which a given pebble tree transducer does not terminate is always a regular language [MSV03, Theorem 4.7]. This applies *a fortiori* to CFPTs. Using this, it is possible[6] to extend any partial function $f : \Gamma^* \rightharpoonup \Sigma^*$ computed by a k-CFPT into a total function $f' : \Gamma^* \to \Sigma^*$ computed by another k-CFPT for the same $k \in \mathbb{N}$, such that $f'(x) = f(x)$ for x in the domain of f and $f'(x) = \varepsilon$ otherwise. This allows us to *only consider CFPTs computing total functions* in the remainder of the paper.

A special case of particular interest is $k = 1$: the transducer has a single reading head, push and pop are always disallowed.

Theorem 3.2.3 ([AČ10])**.** *Copyless SSTs and 1-CFPTs – which are more commonly called* two-way (deterministic) finite transducers (2DFTs) *– are equally expressive.*

[4] There is also an inessential difference: the definition given in [Boj18] does not involve end markers and handles the edge case of an empty input string separately. This has no influence on the expressiveness of the transducer model. Our use of end markers follows [EH01; Lho20].

[5] One would get the same computational power, with the same stack size, by only testing whether $i_j = i_p$ for $j \leq p-1$ as in [MSV03] (this is also essentially what happens in the nested transducers of [Lho20]).

[6] Proof idea: do a first left-to-right pass to determine whether the input leads to non-termination of the original CFPT; if so, terminate immediately with an empty output; otherwise, move the first pebble back to the leftmost position and execute the original CFPT's behavior. This can be implemented by adding finitely many states, including those for a DFA recognizing non-terminating inputs.

Since we took copyless SSTs as our reference definition of regular functions, this means that 2DFTs characterize regular functions. But putting it this way is historically backwards: the equivalence between 2DFTs and MSO transductions came first [EH01] and made this class deserving of the name "regular functions" before the introduction of copyless SSTs.

Let us now show the equivalence with Definition 3.1.2. The reason for this is similar to the reason why k-pebble transducers are equivalent to the *k-nested transducers*[7] of [Lho20], which is deemed "trivial" and left to the reader in [Lho20, Remark 6].

Proposition 3.2.4. *If f is computed by a k-CFPT, and the g_i are computed by l-CFPTs, then $\mathrm{CbS}(f,(g_i)_{i\in I})$ is computed by a $(k+l)$-CFPT.*

Proof. First note that any k-CFPT can be transformed into an equivalent k-CFPT whose transition functions $\delta : Q \times (\Gamma \cup \{\triangleright,\triangleleft\})^p \to Q \times (\mathbb{N}^p \to \mathrm{Stack}_k) \times \Sigma^*$ are such that, for every input $(q,\vec{b})$, we have either $\pi_3(\delta(q,\vec{b})) = \varepsilon$ (in which case we call $\delta(q,\vec{b})$ a *silent transition*) or $\pi_3(\delta(q,\vec{b})) \in \Sigma$ and $\pi_2(\delta(q,\vec{b}))$ is the identity. So, without loss of generality, suppose that we have a k-CFPT $\mathcal{T}_f$ implementing f is of this shape, with state space Q_f and transition function δ_f Similarly, we may assume without loss of generality that the current height of the stack is tracked by the state of CFPTs if we allow multiple final states; assume that we have such height-tracking l-CFPT and that we have l-CFPTs $\mathcal{T}_i$ implementing g_i with state spaces Q_i and transition functions δ_i.

We combine these CFPTs into a single $k+l$ CFPT $\mathcal{T}'$ with state space

$$Q' \;=\; Q_f \;\sqcup\; Q_f \times \bigsqcup_{i\in I} Q_i$$

The initial and final states are those of $\mathcal{T}_f$. The high-level idea is that $\mathcal{T}'$ behaves as $\mathcal{T}_f$ until it produces an output $i \in I$; in such a case it "performs a call" to $\mathcal{T}_i$ that might spawn additional heads to perform its computations. At the end of the execution of $\mathcal{T}_i$, we return the control to $\mathcal{T}_f$. Formally speaking, the transition function δ' of $\mathcal{T}'$ behaves as follows:

- $\delta'(q,\vec{b}) = \delta_f(q,\vec{b})$ if $q \in Q_f$ and $\delta_f(q,\vec{b})$ is *silent*.
- otherwise we take, we have $\pi_3(\delta_f(q,\vec{b})) = i$ for some $i \in I$. Calling r_i the initial state of $\mathcal{T}_i$, we set $\pi_1(\delta'(q,\vec{b})) = (q,r_i)$ and $\pi_2(\delta'(q,\vec{b}))$ corresponds to push a new pebble onto the stack. We make $\delta'(q,\vec{b})$ silent in such a case.
- $\delta'((q,r),\vec{b}\vec{b'})$ then corresponds to $\delta_i(r,\vec{b'})$ if we are not in the situation where the stack height is 1 and the stack update function is pop.
- otherwise we take $\pi_1(\delta'((q,r),\vec{b}\vec{b'})) = \pi_1(\delta_f(q,\vec{b}))$, $\pi_2(\delta'(q,n+1,\vec{b}\vec{b'}))$ to be a pop action and $\pi_3(\delta'((q,r),\vec{b}\vec{b'})) = \pi_3(\delta_i((q,r),\vec{b}\vec{b'}))$. □

Theorem 3.2.5. *If $f : \Gamma^* \to \Sigma^*$ is computed by a k-CFPT, for $k \geq 2$, then there exist a finite alphabet I, a regular function $h : \Gamma^* \to I^*$ and a family $(g_i)_{i\in I}$ computed by $(k-1)$-CFPTs such that $f = \mathrm{CbS}(h,(g_i)_{i\in I})$.*

Proof. Assume we have $f : \Gamma^* \to \Sigma^*$ computed by a k-CFPT $\mathcal{T}$ with state space Q and transition function δ that we assume to be disjoint from Σ. For each $q \in Q$, we describe a $k-1$ CFPT $\mathcal{T}_q$ with the same state space, initial state q and transition function δ_q such that, for every $b' \in \mathbb{N}$, $\vec{b} \in \mathrm{Stack}_l$ for $l \leq k-1$ and $q' \in Q$, $\delta(q',b'\vec{b})$ and $\delta_q(q',\vec{b})$ coincide on the first and last component; on the second component, we require they also coincide up to the difference in stack size. If we fix $r \in Q$, by [MSV03, Theorem 4.7], the language consisting of those $w \in \Gamma^*$ such that $\mathcal{T}_q$ halts on r is regular. Since regular languages are

[7] Remark: nested transducers should yield a machine-independent definition of polyregular functions as the closure of regular functions under a CbS-like operation that relies on *origin semantics* [MP19, §5].

closed under intersection, for any map $\gamma \in Q^Q$, the language $L_\gamma \subseteq \Gamma^*$ of those words w such that $\mathcal{T}_q$ halts on $\gamma(q)$ is regular.

Now fix $\gamma \in Q^Q$ and let us describe a 1-CFPT transducer $\mathcal{T}_\gamma$ intended to implement the restriction of a function $h : \Gamma^* \to (\Sigma \cup Q)^*$ to L_γ. $\mathcal{T}_\gamma$ has the same state space and initial state as $\mathcal{T}$, but has a transition function δ_γ defined by

$$\delta_\gamma(q,b) = \begin{cases} \delta(q,b) & \text{if } \pi_2(\delta(q,b)) \text{ is not a push} \\ (\gamma(r),(p \mapsto p),r) & \text{otherwise, for } r = \pi_1(\delta(q,b)) \end{cases}$$

Since $\Gamma^* = \bigcup_{\gamma \in Q^Q} L_\gamma$, by applying repeatedly Proposition 2.3.11, this determines the regular function $h : \Gamma^* \to (\Sigma \cup Q)^*$. We can then check that $f = \mathrm{CbS}(h, (g_i)_{i \in \Sigma \cup Q})$ where g_a is the constant function outputting the one-letter word a for $a \in \Sigma$ (certainly, some 1-CFPT implements it) and g_q is the function $\Gamma^* \to \Sigma^*$ implemented by the $(k-1)$-CFPT $\mathcal{T}_q$. □

Corollary 3.2.6. *For all $k \in \mathbb{N}$, the functions computed by $(k+1)$-CFPTs are exactly the comparison-free polyregular functions of rank at most k.*

Proof. The proof goes by induction over $k \in \mathbb{N}$. The result holds for $k = 0$ by Theorem 3.2.3; let us detail each direction of the inductive case $k > 0$:

- for the left-to-right inclusion, assume we are given a $(k+1)$-CFPT computing f and apply Theorem 3.2.5 to obtain h and g_is such that $f = \mathrm{CbS}(h, (g_i)_{i \in I})$ with h regular and the g_is computable by k-CFPTs. The induction hypothesis implies that the g_is have rank $< k$, and thus f has rank $\leq k$.
- conversely, if f has rank k, it can be written as $\mathrm{CbS}(h, (g_i)_{i \in I})$ with h regular and the g_is with rank $< k$; the induction hypothesis implies that the g_is can be computed by k-CFPTs. By Theorem 3.2.3, h is computable by a 1-CFPT, so by Proposition 3.2.4, f is computed by a $(k+1)$-CFPT. □

3.3. Key properties of comparison-free polyregular functions

Now that we have both a technically workable definition of cfp functions and a natural model for them, let us study some of their properties. Those that we cover in this section arguably have the most complicated proofs in this chapter.

The first important property is that this function class is *closed under composition* (Theorem 3.3.3). This is the hard part in establishing the third equivalent definition of cfp functions, which consists in swapping out $\mathtt{squaring}_\Gamma$ for some other function in the characterization of polyregular functions from Proposition 2.5.2:

Theorem 3.3.1. *The class of comparison-free polyregular functions is the smallest class closed under usual function composition and containing both all regular functions and the functions $\mathtt{cfsquaring}_\Gamma$ (cf. Example 3.1.3) for all finite alphabets Γ.*

Our next result is the comparison-free incorporates the comparison-free pebble minimization theorem announced at the beginning of this chapter. Our proof techniques mainly draw inspiration from [Lho20]; in particular, one classical tool that we use is Ramsey's theorem specialized to monoid-valued colorings.

Theorem 3.3.2. *Let $f : \Gamma^* \to \Sigma^*$ and $k \in \mathbb{N}$. The following are equivalent:*

(i) *f is comparison-free polyregular with rank at most k;*
(ii) *f is comparison-free polyregular and $|f(w)| = O(|w|^{k+1})$;*
(iii) *there exists a regular function $g : (\{0,\dots,k\} \times \Gamma)^* \to \Sigma^*$ such that $f = g \circ \mathtt{cfpow}_\Gamma^{(k+1)}$, with the following inductive definition: $\mathtt{cfpow}_\Gamma^{(0)} : w \in \Gamma^* \mapsto \varepsilon \in (\varnothing \times \Gamma)^*$ and*

$$\mathtt{cfpow}_\Gamma^{(n+1)} : w \mapsto (n, w_1) \cdot \mathtt{cfpow}_\Gamma^{(n)}(w) \cdot \ldots \cdot (n, w_{|w|}) \cdot \mathtt{cfpow}_\Gamma^{(n)}(w)$$

To make (ii) $\implies$ (i) more precise, if f is cfp with $\mathrm{rk}(f) \geq 1$, then it admits a sequence of inputs $w_0, w_1, \ldots \in \Gamma^*$ such that $|w_n| \to +\infty$ and $|f(w_n)| = \Omega(|w_n|^{\mathrm{rk}(f)+1})$.

Note that $\mathtt{cfpow}^{(2)}_\Gamma$ and $\mathtt{cfsquaring}_\Gamma$ are the same up to a bijection $\{0,1\} \times \Gamma \cong \Gamma \cup \underline{\Gamma}$. We have combined those two theorems into the same section because we have found a way to conveniently share some work between their respective proofs – though this sharing concerns only the easier parts. In Section 3.3.1 we show most of Theorem 3.3.1, and in Section 3.3.2 we show most of Theorem 3.3.2. Finally in Section 3.3.3 we finish proving both.

3.3.1. **Closure under composition.** Let us prove the following.

Theorem 3.3.3. *Comparison-free polyregular functions are closed under composition.*

Remark 3.3.4. As mentioned in the introduction, this will also be an immediate consequence (cf. Proposition 5.1.16) of our characterization of these functions in the $\lambda\ell^{\oplus\&}$-calculus. But we give here a direct proof without this detour outside automata theory.

Consider the composition of two cfp functions. First, by induction on the rank of the left-hand side of the composition, we can reduce to the case where that side is a mere regular function, using the straightforward identity

$$\mathrm{CbS}(f, (g_i)_{i\in I}) \circ h = \mathrm{CbS}(f \circ h, (g_i \circ h)_{i\in I})$$

We then treat this case by another induction, this time on the rank of the right-hand side. The base case is handled by invoking the closure under composition of regular functions (for which we shall give a self-contained proof using our category-theoretic technology in Section 4.7). Therefore, what remains is the following inductive case.

Lemma 3.3.5. *Let $f : \Gamma^* \to I^*$ be a regular function and let $(g_i)_{i\in I}$ be a family of comparison-free polyregular functions $\Gamma^* \to \Sigma^*$. Suppose that for all regular $h : \Sigma^* \to \Delta^*$ and all $i \in I$, the composite $h \circ g_i$ is comparison-free polyregular.*

Then, for all regular $h : \Sigma^* \to \Delta^*$, $h \circ \mathrm{CbS}(f, (g_i)_{i\in I})$ is comparison-free polyregular.

Our proof of this lemma relies on some properties of the transition monoids introduced in §2.3.4. At a high level, the idea is to exploit the following combinatorial phenomenon: in a copyless SST, a transition acting on the state and registers can be specified by

- a "shape" described the *finite* monoid $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ (Proposition 2.3.24);
- plus finitely many "labels" in Σ^* (where Σ is the output alphabet) describing the constant factors that will be concatenated with the old register contents to give the new ones.

This technique is often applied to the study of copyless SSTs (see [BC18, Chapter 13] for instance). It will also be central to our later categorical developments (cf. §4.3.3). Not coincidentally, those categorical tools also appear in our proof of the aforementioned $\lambda\ell^{\oplus\&}$-calculus characterization of cfp functions.

Proposition 3.3.6. *Let $\beta \in \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$. For each $r \in R$, one can write $\beta(r) = w_0 r'_1 w_1 \ldots r'_n w_n$ with $w_0, \ldots, w_n \in \Delta^*$ and $r'_1 \ldots r'_n = \mathtt{erase}_\Delta(\beta)(r) \in R^*$ (cf. Proposition 2.3.22). Hence*

$$\mathcal{M}^{\mathrm{cl}}_{R,\Delta} \;\cong\; \left\{ \left(\alpha, \vec{\ell}\right) \;\middle|\; \alpha \in \mathcal{M}^{\mathrm{cl}}_{R,\varnothing},\ \vec{\ell} \in \prod_{r\in R} (\Delta^*)^{|\alpha(r)|+1} \right\}$$

In other words, register assignments in $\mathcal{M}^{\mathrm{cl}}_{R,\Delta}$ can be decomposed into a "shape" in $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ plus finitely many string labels. Through this bijection, $\mathtt{erase}_\Delta \in \mathrm{Hom}(\mathcal{M}^{\mathrm{cl}}_{R,\Delta} \wr Q, \mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)$ can be seen as simply removing the labels, i.e. the $\vec{\ell}$ component.

Lemma 3.3.7. *Let δ be the transition function of some copyless SST $\Sigma^* \to \Delta^*$ whose sets of states and registers are Q and R respectively, so that $\delta(-,c) \in \mathcal{M}^{\mathrm{cl}}_{R,\Delta} \wr Q$ for $c \in \Sigma$. Let*

$$\psi_\delta \in \mathrm{Hom}(\Sigma^*, \mathcal{M}^{\mathrm{cl}}_{R,\Delta} \wr Q) \quad \text{such that} \quad \forall c \in \Sigma,\ \psi_\delta(c) = \delta(-,c)$$

and $\varphi_\delta = \mathtt{erase}_\Delta \circ \psi_\delta$ as in Remark 2.3.23, $q \in Q$, $r \in R$, $\alpha \in \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ and $j \in \{0, \dots, |\alpha(r)|\}$. Then the following function $\Sigma^* \to \Delta^*$, defined thanks to Proposition 3.3.6, is regular:

$$s \quad \mapsto \quad \begin{cases} w_j & \text{where} \quad \pi_2(\psi_\delta(s)(q))(r) = w_0 r'_1 w_1 \dots r'_n w'_n \quad \text{if } \pi_2(\varphi_\delta(s)(q)) = \alpha \\ \varepsilon & \text{otherwise} \end{cases}$$

(recall that $\pi_2 : Q \times M \to M$ is the second projection and $M \wr Q = Q \to Q \times M$).

Proof. We consider during this proof that the names q, r, α and j introduced in the above statement are *not in scope*, so that we can use those variable names for generic elements of Q, R, $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ and $\mathbb{N}$ instead. Those data will be given other names when we need them.

We build a copyless SST whose set of states is $Q \times \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$. This is made possible by the finiteness of $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ (Proposition 2.3.24). As for the set of registers, we would like it to *vary depending on the current state* for the sake of conceptual clarity, i.e. to have a family of finite sets indexed by $Q \times \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$; when the SST moves from state (q,α) to (q',α'), it would perform a register assignment from $R_{q,\alpha}$ to $R_{q',\alpha'}$ (described by a map $R_{q',\alpha'} \to (\Delta \cup R_{q,\alpha})^*$). Such devices will be called *state-dependent memory copyless SSTs* later (§4.3.2) and they are clearly equivalent in expressive power to usual copyless SSTs.

The idea is that we want the configuration (current state plus register contents) of our new SST, after reading $s = s_1 \dots s_n$, to faithfully represent

$$\psi_\delta(s)(q_0) = (\delta(-,s_1) \bullet \dots \bullet \delta(-,s_n))(q_0) \in Q \times \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$$

where δ and ψ_δ are given in the lemma statement, and q_0 is the given state that was called q in that statement. Following Proposition 3.3.6, since we already have the "shape" stored in the second component $\mathcal{M}^{\mathrm{cl}}_{\Delta,\varnothing}$ of the set $Q \times \mathcal{M}^{\mathrm{cl}}_{\Delta,\varnothing}$ of new states, it makes sense to use the register to store the "labels", hence $R_{q,\alpha} = R_\alpha$ with

$$R_\alpha = \{(r,j) \mid r \in R,\ j \in \{0, \dots, |\alpha(r)|\}\} \quad \text{so that} \quad (\Delta^*)^{R_\alpha} \cong \prod_{r \in R} (\Delta^*)^{|\alpha(r)|+1}$$

The configurations of our SST are thus in bijection with $Q \times \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$ via Proposition 3.3.6, and we would like the transition performed when reading $c \in \Sigma$ to correspond through this bijection to (using the notations of Definition 2.3.3)

$$(q,\beta) \in Q \times \mathcal{M}^{\mathrm{cl}}_{R,\Delta} \quad \mapsto \quad (\delta_{\mathrm{st}}(q),\ \beta \bullet \delta_{\mathrm{reg}}(q))$$

For a fixed $\beta' \in \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$, let us consider the right multiplication $\beta \mapsto \beta \bullet \beta'$ in $\mathcal{M}^{\mathrm{cl}}_{R,\Delta}$. Since $\mathtt{erase}_\Delta : \mathcal{M}^{\mathrm{cl}}_{R,\Delta} \to \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ is a morphism, the "shape" of $\beta \bullet \beta'$ can be obtained from the "shape" of β by multiplying by $\alpha' = \mathtt{erase}_\Delta(\beta')$. The important point is to show that we can obtain the new labels from the old ones by a *copyless assignment* – formally speaking, that for any $\alpha \in \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$ there exists a copyless

$$\gamma_{\alpha,\beta'} : R_{\alpha \bullet \alpha'} \to (\Delta \cup R_\alpha)^*$$

such that for any $\beta \in \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$ such that $\mathtt{erase}_\Delta(\beta) = \alpha$, which therefore corresponds to

$$\left(\alpha, \vec{\ell}\right) \quad \text{for some} \quad \vec{\ell} \in (\Delta^*)^{R_\alpha} \cong \prod_{r \in R} (\Delta^*)^{|\alpha(r)|+1}$$

the shape-label pair that corresponds to $\beta \bullet \beta'$ is $(\alpha \bullet \alpha', \gamma^\dagger_{\alpha,\beta'}(\vec{\ell}))$ (cf. Definition 2.3.1).

Our next task is to analyze the composite assignment $\beta \bullet \beta'$ in order to derive a $\gamma_{\alpha,\beta'}$ that works. Let $r'' \in R$. First, if $\alpha'(r'') = r'_1 \dots r'_n \in R^*$, then

$$\beta'(r'') = w'_0 r'_1 w'_1 \dots r'_n w'_n \quad \text{for some} \quad w'_0, \dots, w'_n \in \Delta^*$$

and by applying the unique morphism $\beta^\star \in \mathrm{Hom}((\Delta \cup R)^*, (\Delta \cup R)^*)$ that extends β and sends letters of Δ to themselves, we have

$$(\beta \bullet \beta')(r'') = \beta^\star(\beta'(r)) = w'_0 \cdot \beta(r'_1) \cdot w'_1 \cdot \ldots \cdot \beta(r'_n) \cdot w'_n$$

Let us decompose further, for $i \in \{1, \dots, n\}$:

$$\beta(r'_i) = w_{i,0} r_{i,1} w_{i,1} \dots w_{i,n_i} r_{n_i} \quad \text{for some} \quad w_{i,0}, \dots, w_{i,n_i} \in \Delta^*$$

By plugging this into the previous equation, we have $(\beta \bullet \beta')(r'') = w_0 r_1 w_1 \dots r_m w_m$ where

$$\{r_1, \dots, r_m\} = \bigcup_{i=1}^{n} \{r_{i,1}, \dots, r_{i,n_i}\}$$

Furthermore, *each w_k for $k \in \{0, \dots, m\}$ is a concatenation of some w'_i and some $w_{i,j}$*, and from the formal expression of w_k depending on these w'_i and $w_{i,j}$ – which only depends on the shape α and α' – we can derive a definition of $\gamma_{\alpha,\beta'}(r'', k)$. For instance,

$$w_{42} = w_{3,2} w'_3 w_{4,0} \qquad \rightsquigarrow \qquad \gamma(r'', 42) = (r'_3, 2) \cdot w'_3 \cdot (r'_4, 0) \in (\Delta \cup R_{\alpha \bullet \alpha'})^*$$

Observe that this does not refer to the $w_{i,j}$; therefore, $\gamma_{\alpha,\beta'}$ does not depend on β, as required. One can check that defined this way, $\gamma_{\alpha,\beta'}$ is indeed a copyless assignment and that the desired property of $\gamma^\dagger_{\alpha,\beta'}$ holds.

What we have just seen is the heart of the proof. We leave it to the reader to finish the construction of the copyless SST. □

With this done, we can move on to proving Lemma 3.3.5, which suffices to finish the proof of Theorem 3.3.3.

Proof of Lemma 3.3.5. Let $w \in \Gamma^*$ be an input string. In the composition, we feed to a copyless SST $\mathcal{T}_h$ that computes h the word $\mathrm{CbS}(f, (g_i)_{i \in I})(w) = g_{i_1}(w) \dots g_{i_k}(w)$ where $f(w) = i_1 \dots i_k$. A first idea is therefore to tweak $\mathcal{T}_h$ into a new copyless SST that takes I^* as input and which executes, when it reads $i \in I$, the transition of $\mathcal{T}_h$ induced by $g_i(w)$. If we call h'_w the regular function computed by this new SST, we would then have $h'_w(f(w)) = h \circ \mathrm{CbS}(f, (g_i)_{i \in I})(w)$. The issue is of course that h'_w depends on the input w.

More precisely, the data that h'_w depends on is the family of transitions

$$(\psi_\delta \circ g_i(w))_{i \in I} \in (\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)^I \qquad \text{(see Lemma 3.3.7 for } \psi_\delta)$$

where Q, R and δ are respectively the set of states, the set of registers and the transition function of $\mathcal{T}_h$. We will be able to disentangle this dependency by working with

$$(\varphi_\delta \circ g_i(w))_{i \in I} = (\mathtt{erase}_\Delta \circ \psi_\delta \circ g_i(w))_{i \in I} \in (\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)^I$$

Concretely:

Claim 3.3.8. For each $\vec{\mu} \in (\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)^I$, there exist:

- a finite alphabet $\Lambda_{\vec{\mu}}$ equipped with a function $\iota_{\vec{\mu}} : \Lambda_{\vec{\mu}} \to I$;
- a regular function $h''_{\vec{\mu}} : I^* \to (\Delta \cup \Lambda_{\vec{\mu}})^*$;
- and regular functions $l_\lambda : \Sigma^* \to \Delta^*$ for $\lambda \in \Lambda$;

such that for $i_1 \dots i_n \in I^*$ and $w \in \Gamma^*$, *if* $(\varphi_\delta \circ g_i(w))_{i \in I} = \vec{\mu}$, *then*

$$h(g_{i_1}(w) \cdot \ldots \cdot g_{i_n}(w)) \;=\; \text{replace each } \lambda \in \Lambda_{\vec{\mu}} \text{ in } h''_{\vec{\mu}}(i_1 \dots i_n) \text{ by } l_\lambda \circ g_{\iota(\lambda)}(w)$$

Subproof of the claim. Proposition 3.3.6 says that every $\beta = \psi_\delta(g_i(w)) \in \mathcal{M}^{\mathrm{cl}}_{R,\Delta}$ can be decomposed into a shape $\alpha = \mathtt{erase}_\Delta(\beta) \in \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ and a finite family $\vec{\ell}$ of strings in Δ^*. Each $\beta(r)$ for $r \in R$ can then be reconstituted as an interleaving of letters in $\alpha(r)$ with labels in $\vec{\ell}$, a process that can be decomposed into two steps:

- first, interleave the letters of $\alpha(r)$ with placeholder letters, taken from an alphabet disjoint from both Δ and R;
- then *substitute* the labels for those letters.

Roughly speaking, this will allow us to manipulate an assignment with placeholders without knowing the labels, and then add the labels afterwards.

Let $\vec{\mu} \in (\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)^I$. We define a copyless SST $\mathcal{T}_{\vec{\mu}}$ with the same sets of states and registers as $\mathcal{T}_h$, namely Q and R. Its initial register values and final output function are also the same. It computes a function $I^* \to (\Delta \cup \Lambda_{\vec{\mu}})^*$, and its transition function is

$$\delta_{\vec{\mu}} : (q,i) \mapsto \Big(\pi_1 \circ \mu_i(q),\ \Big(r \mapsto \mathtt{interleave}\Big(\lambda_0^{q,i,r} \dots \lambda_{|\pi_2(\mu_i(q))(r)|}^{q,i,r},\ \pi_2(\mu_i(q))(r)\Big)\Big)\Big)$$

where $\mathtt{interleave}(u_0 \dots u_n, v_1 \dots v_n) = u_0 v_1 u_1 \dots v_n u_n$ for letters $u_0, \dots, u_n, v_1, \dots, v_n$ over some alphabet (recall also that $\mu_i : Q \to Q \times \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ for $i \in I$). Thus, we take

$$\Lambda_{\vec{\mu}} = \Big\{ \lambda_j^{q,i,r} \;\Big|\; q \in Q,\ i \in I,\ r \in R,\ j \in \{0, \dots, |\pi_2(\mu_i(q))(r)|\} \Big\} \qquad \iota(\lambda_j^{q,i,r}) = i$$

and $h''_{\vec{\mu}}$ to be the function computed by $\mathcal{T}_{\vec{\mu}}$. (Note that although $\delta_{\vec{\mu}}$ does not involve letters from Δ, the final output function and the initial register contents do.) Finally, given $\lambda = \lambda_j^{q,i,r} \in \Lambda_{\vec{\mu}}$, we define l_λ to be the regular function provided by Lemma 3.3.7 for the transition function δ of $\mathcal{T}_h$, the state q_0 (which is the initial state of both $\mathcal{T}_h$ and $\mathcal{T}_{\vec{\mu}}$), the register r, the assignment shape $\alpha = \pi_2(\mu_i(q))$ and the position $j \in \{0, \dots, |\alpha(r)|\}$.

Let $w \in \Gamma^*$ be such that $(\varphi_\delta \circ g_i(w))_{i \in I} = \vec{\mu}$. Consider $\chi_w \in \mathrm{Hom}((\Delta \cup \Lambda_{\vec{\mu}})^*, \Delta^*)$ which maps each letter of Δ to itself and each $\lambda \in \Lambda_{\vec{\mu}}$ to $l_\lambda \circ g_{\iota(\lambda)}(w)$. It lifts to a morphism $\widehat{\chi_w} \in \mathrm{Hom}(\mathcal{M}^{\mathrm{cl}}_{R,\Delta \cup \Lambda_{\vec{\mu}}}, \mathcal{M}^{\mathrm{cl}}_{R,\Delta})$, and we have $\widehat{\chi_w}(\delta_{\vec{\mu}}(-,i)) = \psi_\delta \circ g_i(w)$. This leads to the following invariant: the configuration of $\mathcal{T}_h$ after reading $g_{i_1}(w) \cdot \ldots \cdot g_{i_n}(w)$ is, in a suitable sense, the "image by χ_w" of the configuration of $\mathcal{T}_{\vec{\mu}}$ after reading $i_1 \dots i_n$. (In other words, the "image of the SST $\mathcal{T}_{\vec{\mu}}$ by χ_w" is the copyless SST computing h'_w that we sketched at the very beginning of this proof of Lemma 3.3.5.) This directly implies the property relating h, $h''_{\vec{\mu}}$ and $(l_\lambda)_{\lambda \in \Lambda_{\vec{\mu}}}$ that we wanted. (End of subproof.) □

Let us finish proving Lemma 3.3.5 using the fact we just proved. First of all, since the letters of $\Lambda_{\vec{\mu}}$ only serve as placeholders to be eventually substituted, they can be renamed at our convenience. That means that we can take the $\Lambda_{\vec{\mu}}$ to be *disjoint* for $\vec{\mu} \in (\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)^I$, and define Λ to be their disjoint union. We also take $\iota : \Lambda \to I$ to be the unique common extension of the $\iota_{\vec{\mu}}$. In the same spirit, we glue together the functions $h''_{\vec{\mu}} \circ f$ into

$$H : w \in \Gamma^* \mapsto h''_{(\varphi_\delta \circ g_i(w) | i \in I)}(f(w)) \in (\Delta \cup \Lambda)^*$$

From the above equation on $h''_{\vec{\mu}}$, one can then deduce for *all* $w \in \Gamma^*$ *without condition* that

$$h(\mathrm{CbS}(f, (g_i)_{i \in I})(w)) = \mathrm{CbS}(H, (l_\lambda \circ g_{\iota(\lambda)})_{\lambda \in \Lambda})(w)$$

(strictly speaking, one should have a family indexed by $\Delta \cup \Lambda$ on the right-hand side – to comply with that, just extend the family with constant functions equal to x for each $x \in \Delta$).

Using the above equation, we can rephrase our goal: we want to prove that the function $\mathrm{CbS}(H, (l_\lambda \circ g_{\iota(\lambda)})_{\lambda \in \Lambda})$ is comparison-free polyregular. This class of functions is – by definition – closed under composition by substitution, so we can reduce this to the following subgoals:

- H is comparison-free polyregular: in fact, it is regular, because regular functions are closed under composition and regular conditionals (Proposition 2.3.11). This argument relies on the finiteness of the indexing set $(\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q)^I$ – a consequence of Proposition 2.3.24 – and on the regularity of the language $\{w \in \Gamma^* \mid (\varphi_\delta \circ g_i(w))_{i\in I} = \vec{\mu}\}$ for any $\vec{\mu}$. The reasons for the latter are as follows:
  - φ_δ is a morphism whose codomain $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q$ is finite, so $\varphi_\delta^{-1}(\{\mu_i\})$ is regular for $i \in I$;
  - the functions g_i for $i \in I$ are assumed to be cfp, so they preserve regular languages by inverse image, as all polyregular functions do [Boj18];
  - regular languages are closed under finite intersections, and I is finite.
- $l_\lambda \circ g_{\iota(\lambda)}$ is comparison-free polyregular for all $\lambda \in \Lambda$: because our main existence claim states that l_λ is regular for all $\lambda \in \Lambda$, and one of our assumptions is that any g_i (for $i \in I$) postcomposed with any regular function gives us a cfp function. □

3.3.2. **A lower bound on growth from the rank.** In this section, we prove a statement that directly implies the last claim of Theorem 3.3.2:

Theorem 3.3.9. *Let $f : \Gamma^* \to \Sigma^*$ be comparison-free polyregular of rank at least 1. Then there exists a sequence of inputs $(s_n)_{n\in\mathbb{N}} \in (\Gamma^*)^{\mathbb{N}}$ such that $|s_n| = O(n)$ and $|f(s_n)| \geq n^{\mathrm{rk}(f)+1}$.*

Let us recall a few tools inspired from [Lho20]. First, a notation: for $s \in \Sigma^*$ and $\Pi \subseteq \Sigma$, we shall write $|s|_\Pi$ for the number of occurrences of letters from Π in s.

Lhote's paper [Lho20] also makes a heavy use of *factorizations* of strings that depend on a morphism to a finite monoid. This is the case for our proof as well, but we use a slightly different kind of factorization in order to fix what seems to be a bug in [Lho20].

Definition 3.3.10. An *r-split* of a string $s \in \Gamma^*$ according to a morphism $\varphi : \Gamma^* \to M$ is a tuple $(u, v_1, \dots, v_r, w) \in (\Gamma^*)^{r+2}$ such that:

- $s = uv_1 \dots v_n w$ with v_i non-empty for all $i \in \{1, \dots, r\}$;
- $\varphi(u) = \varphi(uv_1) = \dots = \varphi(uv_1v_r)$;
- $\varphi(w) = \varphi(v_r w) = \dots = \varphi(v_1 \dots v_r w)$.

Proposition 3.3.11 (immediate from the definition)**.** *$(u, v_1, \dots, v_r, w)$ is an r-split if and only if, for all $i \in \{1, \dots, r\}$, $(uv_1 \dots v_{i-1}, v_i, v_{i+1} \dots v_r w)$ is a 1-split.*

Remark 3.3.12. The difference with the *$(1, r)$-factorizations* of [Lho20, Definition 19] is that we have replaced the equality and idempotency requirements on $\varphi(v_1), \dots, \varphi(v_n)$ by the "boundary conditions" involving $\varphi(u)$ and $\varphi(w)$ (actually, $(1, r+2)$-factorizations induce r-splits). We will see in Remark 3.3.17 how this change is helpful.

The point of r-splits is that given a split of an input string according to the morphism that sends it to the corresponding transition in a SST, we have some control over what happens to the output of the SST if we pump a middle factor in the split. Furthermore, it suffices to consider a quotient of the transition monoid which is finite when the SST is copyless (this is similar to Proposition 2.3.24). More precisely, we have the key lemma below, which is used pervasively throughout our proof of Theorem 3.3.9:

Lemma 3.3.13. *Let $f : \Gamma^* \to \Sigma^*$ be a regular function. There exist a morphism to a* finite *monoid $\nu_f : \Gamma^* \to \mathcal{N}(f)$ and, for each $c \in \Sigma$, a set of* producing triples *$P(f, c) \subseteq \mathcal{N}(f)^3$ such that, for any* 1-split *according to ν_f composed of $u, v, w \in \Gamma^*$ – i.e. $\nu_f(uv) = \nu_f(u)$ and $\nu_f(vw) = \nu_f(w)$ – we have:*

- *if $(\nu_f(u), \nu_f(v), \nu_f(w)) \in P(f, c)$, then $|f(uvw)|_c > |f(uw)|_c$;*
- *otherwise (when the triple is not producing), $|f(uvw)|_c = |f(uw)|_c$.*

Furthermore, in the producing case, we get as a consequence that $\forall n \in \mathbb{N}$, $|f(uv^n w)|_c \geq n$.

Something like this (but not exactly) appears in the proof of [Lho20, Lemma 18].

PROOF IDEA. We reuse an idea from [Lho20], but instead of using transition monoids of two-way transducers, we rely on monoids of copyless register assignments. We shall use the notations introduced in Section 2.3.4 for these monoids and the operations they support.

Let R and Σ be finite alphabets. First, we factor $\mathtt{erase}_\Sigma : \mathcal{M}^{\mathrm{cl}}_{R,\Sigma} \to \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$ into two surjective morphisms $\mathcal{M}^{\mathrm{cl}}_{R,\Sigma} \to \mathcal{M}^{\mathrm{cl01}}_{R,\Sigma} \to \mathcal{M}^{\mathrm{cl}}_{R,\varnothing}$, going through a new monoid which keeps some information about the letters of Σ but is still *finite*. To do so, we define an equivalence relation on register assignments as follows: for $\alpha, \beta \in \mathcal{M}^{\mathrm{cl}}_{R,\Sigma}$, we say that $\alpha \sim \beta$ when

- $\mathtt{erase}_\Sigma(\alpha) = \mathtt{erase}_\Sigma(\beta)$;
- for each $r \in R$, the sets of letters *from* Σ that appear in $\alpha(r)$ and $\beta(r)$ are equal.

One can show that $\sim$ is a congruence, so we may form the quotient monoid $\mathcal{M}^{\mathrm{cl01}}_{R,\Sigma} = \mathcal{M}^{\mathrm{cl}}_{R,\Sigma}/\!\sim$. Thanks to the first clause in the definition of $\sim$, the morphism $\mathtt{erase}_\Sigma$ factors through the canonical projection. The quotient is finite since each equivalence class has a representative α such that $|\alpha(r)| \leq |R| + |\Sigma|$ for all $r \in R$: essentially, $\sim$ only takes into account the presence or absence of each letter in Σ, not their multiplicity (hence the notation "01").

Next, let $f : \Gamma^* \to \Sigma^*$ be computed by some copyless SST $(Q, q_0, R, \delta, \vec{u}_I, F)$. We take $\mathcal{N}(f) = \mathcal{M}^{\mathrm{cl01}}_{R,\Sigma} \wr Q$ and define ν_f as a composition $\Gamma^* \to \mathcal{M}^{\mathrm{cl}}_{R,\Sigma} \wr Q \to \mathcal{N}(f)$ where the first morphism – which we may call ψ_δ, as in Lemma 3.3.7 – maps $c \in \Gamma$ to $\delta(-, c)$ and the second morphism is the canonical projection.

What we need to show now is that, given a *1-split* $(u, v, w) \in (\Gamma^*)^3$ with respect to ν_f, the comparison between $|f(uvw)|$ and $|f(uw)|$ depends only on $\nu_f(x)$ for $x \in \{u, v, w\}$.

Let q' and $\vec{u}'_I$ be the state and register values of the SST after reading u; to be more formal, $\psi_\delta(u)(q_0) = (q', \alpha)$ and $\alpha^\dagger(\vec{u}_I) = \vec{u}'_I$. Note that q' is also the first component of the pair $\nu_f(u)(q_0)$; since $\nu_f(uv) = \nu_f(u)$ (by definition of 1-split), the SST reaches the state q' after reading uv as well: $\psi_\delta(q', v) = (q', \beta)$ for some $\beta \in \mathcal{M}^{\mathrm{cl}}_{R,\Sigma}$.

Let $\psi_\delta(w)(q') = (q'', \gamma)$. Then

$$f(uvw) = F(q'')^\dagger \circ (\beta \bullet \gamma)^\dagger(\vec{u}'_I) \qquad f(uw) = F(q'')^\dagger \circ \gamma^\dagger(\vec{u}'_I)$$

Since $\nu_f(vw) = \nu_f(w)$, we have $\mathtt{erase}_\Sigma(\beta \bullet \gamma) = \mathtt{erase}_\Sigma(\gamma)$. Therefore, $\omega = (\beta \bullet \gamma)^\star(F(q''))$ and $\omega' = \gamma^\star(F(q''))$ have the same letters from R with the same multiplicities (and appearing in the same order, although this does not matter for us here): $|\omega|_r = |\omega'|_r$ for all $r \in R$. This is why the two sums over R cancel out in the following computation (writing $\vec{u}'_I = (u'_r)_{r \in R}$):

$$\begin{aligned}
\forall c \in \Sigma, \quad |f(uvw)|_c - |f(uw)|_c &= |\omega^\dagger(\vec{u}'_I)|_c - |(\omega')^\dagger(\vec{u}'_I)|_c \\
&= |\omega|_c + \sum_{r \in R} |\omega|_r \cdot |u'_r|_c - |\omega'|_c - \sum_{r \in R} |\omega'|_r \cdot |u'_r|_c \\
&= |\omega|_c - |\omega'|_c
\end{aligned}$$

From now on, let $c \in \Sigma$. From the definition of $\beta^\star$, we have

$$|\omega|_c = |\beta^\star(\omega')|_c = |\omega'|_c + \sum_{r \in R} |\omega'|_r \cdot |\beta(r)|_c$$

So we get the dichotomy of the lemma statement:

- if there exists some $r \in R$ such that $|\omega'|_r > 0$ and $|\beta(r)|_c > 0$, then $|f(uvw)| > |f(uw)|$;
- otherwise, $|f(uvw)| = |f(uw)|$.

For each $r \in R$, the condition $|\omega'|_r > 0$ can be checked from q' and $\nu_f(w)$; in turn, q' depends only on $\nu_f(u)$. As for $|\beta(r)|_c > 0$, since it is a condition on the presence or not of a certain letter from Σ in $\beta(r)$, without considering its precise multiplicity, it depends only on $\nu_f(v)$:

this is the information that ν_f was designed to encode. This gives us the definition of the set of producing triples $P(f,c)$.

There remains a final claim to prove in the lemma statement, concerning $|f(uv^nw)|_c$ when $(u,v,w) \in P(f,c)$ and $n \in \mathbb{N}$. Using $\mu(uv) = \mu(u)$, one can show that for all $m \in \mathbb{N}$, the triple (uv^m, v, w) is also a producing 1-split. So we have

$$|f(uv^nw)|_c > |f(uv^{n-1}w)|_c > \cdots > |f(uw)|_c$$

and since all elements of this sequence are natural numbers, $|f(uv^nw)|_c \geq n$. □

Definition 3.3.14. We fix once and for all a choice of $\mathcal{N}(f)$, ν_f and $P(f,c)$ for each regular function $f : \Gamma^* \to \Sigma^*$ and letter $c \in \Sigma$. We say that a 1-split (u,v,w) is *producing with respect to* (f,c) when $(\nu_f(u), \nu_f(v), \nu_f(w)) \in P(f,c)$. For $\Pi \subseteq \Sigma$, we also set

$$P(f,\Pi) = \bigcup_{c \in \Pi} P(f,c)$$

Lemma 3.3.15 (Rank 0 dichotomy)**.** *Let $f : \Gamma^* \to \Sigma^*$ be a regular function, M be a finite monoid and $\varphi : \Gamma^* \to M$ be a morphism. Suppose that there exists another morphism $\pi : M \to \mathcal{N}(f)$ such that $\pi \circ \varphi = \nu_f$. Let $r \geq 1$ and $\Pi \subseteq \Sigma$.*

We define $L(f,\Pi,\varphi,r)$ to be the set of strings that admit an r-split $s = uv_1 \dots v_r w$ according to φ such that all the triples $(uv_1 \dots v_{i-1}, v_i, v_{i+1} \dots v_r w)$ are producing with respect to (f,Π) – let us call this a producing r-split with respect to (f,Π,φ).

Then $L(f,\Pi,\varphi,r)$ is a regular language, and $\sup\{|f(s)|_\Pi \mid s \in \Gamma^* \setminus L(f,\Pi,\varphi,r)\} < \infty$.

To prove this, we first recall a standard Ramsey argument. The statement below is a bit stronger than necessary for the purpose of showing the above lemma, but it will be reused in the proof of Theorem 3.5.3.

Proposition 3.3.16 (analogous to [Lho20, Claim 20])**.** *Let Γ be an alphabet, M be a finite monoid and $\varphi : \Gamma^* \to M$ be a morphism. There exists $N \in \mathbb{N}$ such that any string $s = uvw \in \Gamma^*$ such that $|v| \geq N$ admits an r-split $s = u'v'_1 \dots v'_r w'$ according to φ in which u is a prefix of u' and w is a suffix of w'.*

Proof. By the finite Ramsey theorem for pairs, there exists $R \in \mathbb{N}$ such that every complete undirected graph with at least R vertices whose edges are colored using $|M|$ colors contains a monochromatic clique with $r+3$ vertices. We take $N = R - 1$.

Let $s = uvw \in \Gamma^*$ with $|v| \geq N$. Let us write $s[i \dots j]$ for the substring of s between two positions $i, j \in \{0, \dots, |s|\}$. Those indices are considered as positions *in-between* letters, so, for instance, $s = s[0 \dots |s|]$, while $s[(i-1) \dots i]$ is the i-th letter of s; note also that $s[i \dots j] \cdot s[j \dots k] = s[i \dots k]$. In particular, we have $v = s[|u| \dots |uv|]$.

Consider the following coloring of the complete graph over $V = \{|u|, \dots, |uv|\}$: the edge $(i,j) \in V^2$ with $i < j$ is given the color $\varphi(s[i \dots j])$. Since $|V| \geq N + 1 = R$, there exists a monochromatic clique $\{i_0, \dots, i_{r+2}\} \subseteq V$ with $i_0 < \cdots < i_{r+2}$.

We now define $u' = s[0 \dots i_1]$ and $w' = s[i_{r+1} \dots |s|]$, which ensures that u is a prefix of u' and w is a suffix of w' since i_1 and i_{r+1} are positions in v. For $m \in \{1, \dots, r\}$, we also take $v'_m = s[i_m \dots i_{m+1}] \neq \varepsilon$ (because $|v'_m| = i_{m+1} - i_m \geq 1$). Then $s = u'v'_1 \dots v'_r w'$, and

$$\begin{aligned}
\varphi(u'v'_1 \dots v'_m) &= \varphi(s[0 \dots i_{m+1}]) = \varphi(s[0 \dots i_0])\varphi(s[i_0 \dots i_{m+1}]) \\
&= \varphi(s[0 \dots i_0])\varphi(s[i_0 \dots i_1]) \quad \text{by monochromaticity} \\
&= \varphi(s[0 \dots i_1]) = \varphi(u')
\end{aligned}$$

and similarly, $\varphi(v'_m \dots v'_1 w') = \varphi(w')$. Thus, by definition, we have an r-split of s. □

PROOF OF LEMMA 3.3.15. $L(f, \Pi, \varphi, r)$ can be recognized by a non-deterministic automaton that guesses an adequate r-split and computes $\varphi(u), \varphi(v_1), \dots, \varphi(v_r), \varphi(w)$. The hard part is showing that $|f(-)|_\Pi$ is bounded on the complement of this language.

By the previous proposition, there exists some $N \in \mathbb{N}$ such that any string $s \in \Gamma^*$ of length at least N admits an r-split according to φ. Thanks to the existence of π, it is also an r-split according to ν_f. So if this long string is in $\Gamma^* \setminus L(f, \Pi, \varphi, r)$, then it is of the form $s = uv_1 \dots v_r w$ where, for some $i \in \{1, \dots, r\}$, $(uv_1 \dots v_{i-1}, v_i, v_{i+1} \dots v_r w)$ is *not* producing. Therefore, $|f(uv_1 \dots v_{i-1}v_{i+1} \dots v_r w)|_\Pi = |f(uv_1 \dots v_r w)|_\Pi$. The important part is that the argument in the left-hand side is strictly shorter (the definition of r-split contains $v_i \neq \varepsilon$). Furthermore, we claim that $s' = uv_1 \dots v_{i-1}v_{i+1} \dots v_r w \in \Gamma^* \setminus L(f, \Pi, \varphi, r)$. Once this is established, a strong induction on the length suffices to show that $|f(-)|_\Pi$ restricted to $\Gamma^* \setminus L(f, \Pi, \varphi, r)$ reaches its maximum at some string of length smaller than N, and thus to conclude the proof.

It remains to show that $s' \notin L(f, \Pi, \varphi, r)$. If this were false, then by definition we would have a producing r-split $s' = u'v'_1 \dots v'_r w'$. Assuming this, we will lift this split to a producing r-split of s in order to contradict $s \notin L(f, \Pi, \varphi, r)$. We give notations to the components of our non-producing triple: $\hat{u} = uv_1 \dots v_{i-1}$, $\hat{v} = v_i$, $\hat{w} = v_{i+1} \dots v_r w$.

Suppose that for some $j \in \{1, \dots, r\}$ and $x \in \Gamma^*$, we have $\hat{u} = u'v'_1 \dots v'_{j-1}x$ and $|x| \leq |v'_j|$. Then there must exist a unique $y \in \Gamma^*$ such that $v'_j = xy$ and $\hat{w} = yv'_{j+1} \dots v'_r w'$. What we want to show now is that $(u', v'_1, \dots, v'_{j-1}, x\hat{v}y, v'_{j+1}, \dots, v'_r, w')$ is a producing r-split of s.

- First, the concatenation of this sequence of length $r + 2$ is indeed equal to $\hat{u}\hat{v}\hat{w} = s$.
- Next, we have $\varphi(u'v'_1 \dots v_{j-1}(x\hat{v}y)) = \varphi(\hat{u}\hat{v}y) = \varphi(\hat{u}\hat{v})\varphi(y) = \varphi(\hat{u})\varphi(y)$ since $(\hat{u}, \hat{v}, \hat{w})$ is a 1-split of s, and $\varphi(\hat{u})\varphi(y) = \varphi(\hat{u}y) = \varphi(u'v'_1 \dots v'_j)$. For $k \geq j$, by multiplying by $\varphi(v'_{j+1} \dots v'_k)$ on the right, we get $\varphi(u'v'_1 \dots v'_{j-1}(x\hat{v}y)v'_{j+1} \dots v'_k) = \varphi(u'v'_1 \dots v'_k)$. Similarly, for $k \leq j$, we have $\varphi(v'_k \dots v'_{j-1}(x\hat{v}y)v'_{j+1} \dots v'_r w') = \varphi(v'_k \dots v'_r w')$.
- Combining the above with the fact that $(u', v'_1, \dots, v'_r, w')$ is an r-split of s' gives us directly from the definitions that our new $(r + 2)$-tuple with $x\hat{v}y$ is an r-split of s.
- Finally, we must check that it is producing.
  - Let $k \leq j - 1$. We must show that $(u'v'_1 \dots v'_{k-1}, v_k, v'_{k+1} \dots v'_{j-1}(x\hat{v}y)v_{j+1} \dots v'_r w')$ is producing with respect to (f, Π). We have seen previously that the componentwise image by φ of this triple is the same as the one for $(u'v'_1 \dots v'_{k-1}, v'_k, v'_{k+1} \dots v'_r w')$. The latter is producing (since it comes from an r-split chosen to be producing), and therefore so is the former, because thanks to $\nu_f = \pi \circ \varphi$, the image by φ suffices to determine whether a triple is producing.
  - The case $k \geq j + 1$ is symmetrical.
  - The remaining case is $(u'v'_1 \dots v'_{j-1}, x\hat{v}y, v'_{j+1} \dots v'_r w')$. It would be convenient if $x\hat{v}y$ and v'_j had the same image by φ, but this is not guaranteed. Instead, we come back to the dichotomy concerning what happens when we remove the substring $x\hat{v}y$ in s. This can be done in two steps: first remove $\hat{v}$ in s, which gives us s', then remove $xy = v'_j$ from s', resulting in $s'' = u'v'_1 \dots v'_{j-1}v'_{j+1} \dots v'_r w'$. Using the fact that the r-split of s' is producing while $(\hat{u}, \hat{v}, \hat{w})$ is not, we have $|f(s)|_\Pi = |f(s')|_\Pi > |f(s'')|_\Pi$. This means that the 1-split containing $x\hat{v}y$ must be producing.

If the (j, x) chosen previously does not exist, then either u' is a prefix of $\hat{u}$ or w' is a suffix of $\hat{w}$. In those cases, there is an analogous lifting procedure, and its proof of correctness is simpler; we leave this to the reader. □

Remark 3.3.17. It is not clear whether the above reasoning can be made to work if we require idempotency in the definition of r-split, as is the case for the (k, r)-factorizations

in [Lho20] (cf. Remark 3.3.12). An analogous argument is made in the first paragraph of the proof of the original Dichotomy Lemma in [Lho20], but we were unable to check that $s' \notin L(f, \Pi, \varphi, r)$ when forcing the central elements of producing triples to be idempotent. (This is an arguably minor bug and does not constitute the main reason for the failure of the proof strategy of [Lho20] in the case of general pebble transducers.) Thankfully, idempotency does not seem to be required to carry out further arguments leading to a proof of Theorem 3.3.9.

Lemma 3.3.18. *Let $f : \Gamma^* \to \Sigma^*$ be comparison-free polyregular. There exist a morphism to a* finite *monoid $\nu'_f : \Gamma^* \to \mathcal{N}'(f)$ such that, for any* 1-split *according to ν'_f composed of $u, v, w \in \Gamma^*$ and any $c \in \Sigma^*$, the sequence $(|f(uv^n w)|_c)_{n\in\mathbb{N}}$ is non-decreasing.*

Proof. By straightforward induction on $\mathrm{rk}(f)$, using Lemma 3.3.13: for g regular and h_i comparison-free, we take $\mathcal{N}'(\mathrm{CbS}(g, (h_i)_{i\in I})) = \mathcal{N}(g) \times \prod_{i\in I} \mathcal{N}'(h_i)$. □

Lemma 3.3.19. *Let $g : \Gamma^* \to I^*$ be a regular function and, for each $i \in I$, let $h_i : \Gamma^* \to \Sigma^*$ be comparison-free polyregular of rank at most k. Suppose that $\sup_{s\in\Gamma^*} |g(s)|_J < \infty$ where*

$$J = \begin{cases} \{i \in I \mid \mathrm{rk}(h_i) = k\} & \text{when } k \geq 1 \\ \{i \in I \mid |h_i(\Gamma^*)| = \infty\} & \text{when } k = 0 \end{cases}$$

(Morally, regular functions with finite range play the role of "comparison-free polyregular functions of rank -1".) Then $\mathrm{rk}(\mathrm{CbS}(g, (h_i)_{i\in I})) \leq k$.

Remark 3.3.20. This is analogous to [Lho20, Claim 22], but it also seems to be related to the way the "nested transducer" R_{k+1} is defined in the proof of the dichotomy lemma in [Lho20]: indeed, R_{k+1} can call either a k-nested subroutine or a $(k-1)$-nested one.

Proof. We write $f = \mathrm{CbS}(g, (h_i)_{i\in I})$.

First, let us consider the case $k = 0$. For convenience, we assume w.l.o.g. that $I \cap \Sigma = \varnothing$. Let $N = \sup\{|g(s)|_J \mid s \in \Gamma^*\}$ – in the degenerate case $J = \varnothing$, this leads to $N = 0$ – and $\iota_n(s)$ be the n-th letter of J in $g(s)$ if it exists, or else ε. Then we use the equation $g(s) = \rho_0(s)\iota_1(s)\rho_1(s)\dots\iota_N(s)\rho_N(s)$ to define uniquely $\rho_0, \dots, \rho_N : \Gamma^* \to (I \setminus J)^*$. One can build for each $n \in \{0, \dots, N\}$ a sequential transducer whose composition with g yields ρ_n; therefore, since g is regular, so is ρ_n. We define $\psi_n(s)$ next as $h_{\iota_n(s)}(s)$ when $\iota_n(s) \in J$, and ε otherwise. For any $n \in \{1, \dots, N\}$, since the languages $g^{-1}(((I \setminus J)^* J)^{n-1}(I \setminus J)^* i I^*)$ are regular for all $i \in J$, this defines ψ_n as a combination of $\{s \mapsto \varepsilon\} \cup \{h_i \mid i \in J\}$ by regular conditionals, so ψ_n is regular. Finally, we set $f'(s) = \rho_0(s)\psi_1(s)\rho_1(s)\dots\psi_N(s)\rho_N(s) \in (\Sigma \cup I \setminus J)^*$; the function f' thus defined is regular by closure under concatenation (use a product construction on copyless SSTs). Observe that $f'(s)$ is obtained by substituting each occurrence of a letter $i \in J$ in $g(s)$ by $h_i(s)$ (thus, it is equal to $g(s)$ when $J = \varnothing$, and to $f(s)$ when $J = I$).

What remains to do is to substitute the letters of $I \setminus J$ to get f. To do so, let us define $L_{\vec{w}} = \{s \in \Gamma^* \mid \forall i \in I \setminus J,\ h_i(s) = w_i\}$ for $\vec{w} = (w_i)_{i\in I\setminus J} \in \prod_{i\in I\setminus J} h_i(\Gamma^*)$. The function f coincides on $L_{\vec{w}}$ with f' postcomposed with the morphism that replaces each $i \in I \setminus J$ by w_i; this is regular by closure under composition. Furthermore, the factors of $\prod_{i\in I\setminus J} h_i(\Gamma^*)$ are finite by definition of J, and $I \setminus J$ itself is a subset of the finite alphabet I. So there are finitely many $L_{\vec{w}}$, and they partition Σ^*; they are also all regular, as finite intersections of preimages of singletons by regular functions. Therefore, f is obtained by combining regular functions by a regular conditional, so it is regular, i.e. $\mathrm{rk}(f) = 0$ as we wanted.

This being done, let us move on to the case $k \geq 1$. For $i \in J$, let $h_i = \mathrm{CbS}(g'_i, (h'_{i,x})_{x \in X_i})$ where all the g'_i are regular and the $h'_{i,x}$ are of rank at most $k-1$, choosing the X_i to be pairwise disjoint as well as disjoint from I. Let $f'(s)$ be obtained from $g(s)$ by substituting each occurrence of a letter $i \in J$ by $g'_i(s)$. For the same reasons as those exposed in the first paragraph of the case $k = 0$, this defines a regular function f'. By taking its composition by substitution with the disjoint union of the families $(h_i)_{i \in I \setminus J}$ and $(h'_{i,x})_{x \in X_i}$ for $i \in J$, we recover f. Since the functions involved in this union family are all of rank at most $k-1$ (by definition of J), this means that $\mathrm{rk}(f) \leq k$. □

Lemma 3.3.21. *Let $f : \Gamma^* \to \Sigma^*$ be a comparison-free polyregular function. Let $\varphi : \Gamma^* \to M$ be a morphism to a finite monoid and let $r \in \mathbb{N}_{\geq 1}$. Then there exists a regular language $\widehat{L}(f, \varphi, r) \subseteq \Gamma^*$ such that:*

- *the function which maps $\widehat{L}(f, \varphi, r)$ to ε and coincides with f on $\Gamma^* \setminus \widehat{L}(f, \varphi, r)$*
  - *is regular and takes finitely many values if $\mathrm{rk}(f) = 0$ i.e. f is regular;*
  - *is comparison-free polyregular with rank strictly lower than $\mathrm{rk}(f)$ otherwise;*
- *for any $s \in \widehat{L}(f, \varphi, r)$, there exist $k = \mathrm{rk}(f) + 1$ r-splits according to φ – let us write them as $s = u^{(m)} v_1^{(m)} \dots v_r^{(m)} w^{(m)}$ for $m \in \{1, \dots, k\}$ – such that, for any factorization $s = \alpha_0 \beta_1 \alpha_1 \dots \beta_k \alpha_k$ where, for some* permutation σ *of $\{1, \dots, k\}$, each β_m coincides with some $v_l^{(\sigma(m))}$ (in the sense that their positions as substrings of s are equal), we have*

$$\forall n \in \mathbb{N},\ |f(\alpha_0 \beta_1^n \alpha_1 \dots \beta_k^n \alpha_k)| \geq n^k$$

(note that in general, such factorizations $s = \alpha_0 \beta_1 \alpha_1 \dots \beta_k \alpha_k$ might not exist, for instance when $r = 1$ and all the substrings $v_1^{(m)}$ overlap)

Proof. We proceed by induction on $\mathrm{rk}(f)$.

Base case: $\mathrm{rk}(f) = 0$. In this case, f is regular. Let $\psi : \Gamma^* \to M \times \mathcal{N}(f)$ be the monoid morphism obtained by pairing φ (given in the lemma statement) with ν_f (given in Lemma 3.3.15). Then, using Lemma 3.3.15, one can see that taking $\widehat{L}(f, \varphi, r) = L(f, \Sigma, \psi, r)$ works.

Inductive case: $\mathrm{rk}(f) \geq 1$. In this case, $f = \mathrm{CbS}(g, (h_i)_{i \in I})$ for some regular $g : \Gamma^* \to I^*$ and some comparison-free polyregular $h_i : \Gamma^* \to \Sigma^*$ with $\mathrm{rk}(h_i) \leq \mathrm{rk}(f) - 1$ for all $i \in I$. Let φ and r be as given in the lemma statement. Let J be defined as in Lemma 3.3.19:

$$J = \begin{cases} \{i \in I \mid \mathrm{rk}(h_i) = \mathrm{rk}(f) - 1\} & \text{when } \mathrm{rk}(f) \geq 2 \\ \{i \in I \mid |h_i(\Gamma^*)| = \infty\} & \text{when } \mathrm{rk}(f) = 1 \end{cases}$$

For $i \in J$, let $\psi_i : \Gamma^* \to M \times \mathcal{N}(g) \times \mathcal{N}'(h_i)$ be obtained by combining φ with the morphisms given by Lemmas 3.3.13 and 3.3.18. We shall consider the regular languages $\widehat{L}(h_i, \psi_i, r)$ provided by the inductive hypothesis.

Let us take a copy $\underline{J} = \{\underline{i} \mid i \in J\}$ of J such that $\underline{J} \cap I = \varnothing$. We define the regular function $g' : \Gamma^* \to (I \cup \underline{J})^*$ as follows: for any input $s \in \Gamma^*$, to build the output $g'(s)$, we start from $g(s)$ and then, for each $i \in J$ *such that* $s \notin \widehat{L}(h_i, \psi_i, r)$, we replace all the occurrences of i by $\underline{i}$. For $i \in J$, we also define $h_{\underline{i}}$ to be the function which maps $\widehat{L}(h_i, \psi_i, r)$ to ε and coincides with h_i on $\Gamma^* \setminus \widehat{L}(h_i, \psi_i, r)$. By construction, $f = \mathrm{CbS}(g', (h_i)_{i \in I \cup \underline{J}})$.

Note that g' is regular: it is indeed generated from g using regular conditionals and postcomposition by letter-to-letter morphisms. We can therefore build a morphism

$$\chi : \Gamma^* \to M \times \mathcal{N}(g') \times \prod_{i \in J} \mathcal{N}'(h_i)$$

in the expected way, and define the language provided by the lemma statement as

$$\widehat{L}(f,\varphi,r) = L(g',J,\chi,r)$$

According to Lemma 3.3.15, it is indeed a regular language. Concerning the first item of the lemma statement, the function that it considers can be expressed as

$$\mathrm{CbS}(g'',(h_i)_{i\in I\cup\underline{J}}) \qquad \text{where} \qquad g'' : s \mapsto \begin{cases} \varepsilon & \text{when } s \in L(g',J,\chi,r) \\ g'(s) & \text{otherwise} \end{cases}$$

We want to show that $\mathrm{rk}(\mathrm{CbS}(g'',(h_i)_{i\in I\cup\underline{J}})) \leq \mathrm{rk}(f)-1$. The shape of this statement fits with the conclusion of Lemma 3.3.19, so we just have to check the corresponding assumptions.

- g'' is regular, by closure of regular functions under regular conditionals.
- for $i \in I \cup \underline{J}$, the function h_i is comparison-free polyregular of rank at most $\mathrm{rk}(f)-1$:
  - for $i \in I$, this was required in our choice of expression for $f = \mathrm{CbS}(g,(h_i)_{i\in I})$ (and such a choice was possible by definition of rank);
  - for $i = \underline{j} \in \underline{J}$, we get this by applying the first item of the inductive hypothesis to h_j (indeed, the function introduced by this item is none other than $h_{\underline{j}} = h_i$).
- We also get that, *with the same J as before,*

$$J = \begin{cases} \{i \in I\cup\underline{J} \mid \mathrm{rk}(h_i) = \mathrm{rk}(f)-1\} & \text{when } \mathrm{rk}(f) \geq 2 \text{ i.e. } \forall i \in J,\ \mathrm{rk}(h_i) \geq 1 \\ \{i \in I\cup\underline{J} \mid |h_i(\Gamma^*)| = \infty\} & \text{when } \mathrm{rk}(f) = 1 \text{ i.e. } \forall i \in J,\ \mathrm{rk}(h_i) = 0 \end{cases}$$

  using again the first item of the inductive hypothesis to handle the case of indices in $\underline{J}$.
- Finally, by definition of g'' and by Lemma 3.3.15, using the convention $\sup \varnothing = 0$,

$$\sup_{s\in\Gamma^*} |g''(s)|_J = \sup\{|g'(s)|_J \mid s \in \Gamma^* \setminus L(g',J,\chi,r)\} < \infty$$

Let us now check the second item concerning splits and factorizations. Let $s \in \widehat{L}(f,\varphi,r)$. By definition, there exists $i \in J$ such that $s \in L(g',i,\chi,r)$. In particular, $|g'(s)|_i \geq 1$, which entails that $s \in \widehat{L}(h_i,\psi_i,r)$ by definition of g'. The inductive hypothesis gives us a family of r-splits $s = u^{(m)}v_1^{(m)}\dots v_r^{(m)}w^{(m)}$ according to ψ for $m \in \{1,\dots,k-1\}$ – recall that $\mathrm{rk}(h_i)+1 = \mathrm{rk}(f) = k-1$. We complete it by taking $(u^{(k)},v_1^{(k)},\dots,v_r^{(k)},w^{(k)})$ to be a producing r-split of s with respect to (g',i,χ), whose existence is guaranteed by definition of $L(g',i,\chi,r)$. Since φ factors through both ψ_i and χ by construction, this indeed gives us a family of k r-splits according to φ.

Now, let $s = \alpha_0\beta_1\alpha_1\dots\beta_k\alpha_k$ be a factorization and σ be a permutation of $\{1,\dots,k\}$ such each β_m coincides with some $v_l^{(\sigma(m))}$ for some l. Note that from the original expression of f as a composition by substitution, we have

$$\forall s' \in \Gamma^*, \quad |f(s')| \geq |g(s')|_i \cdot |h_i(s')|$$

Therefore, our desired inequality will follow once we prove the ones below:

$$\forall n \in \mathbb{N}, \quad |g(\alpha_0\beta_1^n\alpha_1\dots\beta_k^n\alpha_k)|_i \geq n \quad \text{and} \quad |h_i(\alpha_0\beta_1^n\alpha_1\dots\beta_k^n\alpha_k)| \geq n^{k-1}$$

To illustrate the idea, we assume $\sigma(k) = k$, so that $\beta_k = v_l^{(k)}$ for some l, and we invite the reader to convince themself that this is merely a matter of notational convenience for the rest of the proof.

Let us start with h_i. Since ν'_{h_i} factors through χ, the triple

$$(\alpha_0\beta_1\dots\beta_{k-1}\alpha_{k-1},\beta_k,\alpha_k) = (u^{(k)}v_1^{(k)}\dots v_{l-1}^{(k)}, v_l^{(k)}, v_{l+1}^{(k)}\dots v_r^{(k)}w^{(k)})$$

is a 1-split according to ν'_{h_i}. Using the fact that ν'_{h_i} factors through ψ_i, one can show that $(\alpha_0\beta_1^n \dots \beta_{k-1}^n\alpha_{k-1}, \beta_k, \alpha_k)$ is still a 1-split according to ν'_{h_i}. Therefore, for $n \in \mathbb{N}$,

$$|h_i(\alpha_0\beta_1^n \dots \beta_{k-1}^n\alpha_{k-1}\beta_k^n\alpha_k)| \geq |h_i(\alpha_0\beta_1^n \dots \beta_{k-1}^n\alpha_{k-1}\beta_k\alpha_k)| \geq n^{k-1}$$

where β_k is *not* raised to the n-th power in the middle; the left inequality comes from Lemma 3.3.18, while the right inequality is part of the induction hypothesis applied to h_i.

The case of g requires an additional step. We know that $(\alpha_0\beta_1 \dots \beta_{k-1}\alpha_{k-1}, \beta_k, \alpha_k)$ is a producing triple with respect to (g', i, χ); therefore, by Lemma 3.3.13,

$$\forall n \in \mathbb{N},\ |g'(\alpha_0\beta_1 \dots \beta_{k-1}\alpha_{k-1}\beta_k^n\alpha_k)|_i \geq n$$

To replace g' by g in the above inequality, recall that by definition of g', since $i \in J$,

$$\forall s' \in \Gamma^*,\ \big(|g'(s')|_i \neq 0 \implies |g'(s')|_i = |g(s')|_i\big)$$

One can then conclude by Proposition 3.3.22 below, taking $l = k - 1$. There is a subtlety here: our definitions ensure that ν_g factors through ψ_i, but this might not be the case for $\nu_{g'}$ (because ψ_i had to be defined before g'). So for this final step, we must work with the function g, whereas to leverage the producing triple, we had to use g'. □

The following proposition, which we used at the end of the above proof, will also be useful to prove Theorem 3.5.3.

Proposition 3.3.22. *Let $g : \Gamma^* \to \Sigma^*$ be a regular function and $s = \alpha_0\beta_1\alpha_1 \dots \beta_l\alpha_l \in \Gamma^*$ such that every triple $(\alpha_0\beta_1 \dots \alpha_m, \beta_{m+1}, \alpha_{m+1}\beta_{m+2} \dots \alpha_l)$ is a 1-split according to ν_g. Then for every $c \in \Sigma$, the function*

$$(n_1, \dots, n_l) \mapsto |g(\alpha_0\beta_1^{n_1}\alpha_1 \dots \beta_l^{n_l}\alpha_l)|_c$$

is monotone according to the product partial order on $\mathbb{N}^l$.

Proof idea. In order to apply Lemma 3.3.15, the key observation is that the triple

$$(\alpha_0\beta_1^{n_1} \dots \alpha_m\beta_{m+1}^{n_{m+1}}, \beta_{m+1}, \alpha_{m+1}\beta_{m+2}^{n_{m+2}} \dots \alpha_l)$$

is also a 1-split. This is because we have, by definition of 1-split, $\nu_g(\alpha_0\beta_1^{n_1}) = \nu_g(\alpha_0\beta_1)$, then $\nu_g(\alpha_0\beta_1\alpha_1\beta_2^{n_2}) = \nu_g(\alpha_0\beta_1\alpha_1\beta_2)$, etc., and similarly on the right side. □

After having established Lemma 3.3.21, we can use it to finally wrap up this section.

Proof of Theorem 3.3.9. We apply Lemma 3.3.21 to get a language $\widehat{L}(f, \varphi, \mathrm{rk}(f)+1)$ where φ does not matter (take for instance the morphism from Γ^* to the trivial monoid). It must be non-empty (or else we would have the contradiction $\mathrm{rk}(f) < \mathrm{rk}(f)$), so we can choose an arbitrary element $s \in \widehat{L}(f, \varphi, \mathrm{rk}(f) + 1)$.

Let $k = \mathrm{rk}(f) + 1$. Lemma 3.3.21 gives us k factorizations $s = u^{(m)}v_1^{(m)} \dots v_k^{(m)}w^{(m)}$ satisfying certain properties. Note that k plays two roles here that were distinct in the lemma. We claim that thanks to this, there exists a factorization $s = \alpha_0\beta_1\alpha_1 \dots \beta_k\alpha_k$ as described in Lemma 3.3.21. This entails that setting $s_n = \alpha_0\beta_1^n\alpha_1 \dots \beta_k^n\alpha_k$ proves the theorem.

Our task is therefore to select one element in each of the k sets $\{v_l^{(m)} \mid l \in \{1, \dots, k\}\}$ of substrings of s for $m \in \{1, \dots, k\}$, such that the selected substrings are pairwise non-overlapping. There is a strategy for this which is similar to the classical greedy algorithm for computing a maximum independent set in an interval graph. We take β_1 to be the substring of s among the $v_1^{(m)}$ whose right endpoint is leftmost. One can check that β_1 cannot overlap with any $v_l^{(m)}$ for $l \geq 2$. Thus, by discarding the set to which β_1 belongs, as well as each $v_1^{(m)}$ in the other sets, we reduce the remainder of the task to our original goal with k being decremented by 1. At this stage, an induction suffices to conclude the proof. □

3.3.3. **Proofs of Theorems 3.3.1 and 3.3.2.** Now that we have shown that cfp functions are closed under composition and that their asymptotic growth are tightly linked to their ranks, we have the essential ingredients to prove Theorems 3.3.1 and 3.3.2. There are a couple of preliminary lemmas helpful for both that we first prove here.

Lemma 3.3.23. *For any comparison-free polyregular function $f : \Gamma^* \to \Sigma^*$ and $k \geq \mathrm{rk}(f)$, there exists a regular function $f' : (\{0, \dots, \mathrm{rk}(f)\} \times \Gamma)^* \to \Sigma^*$ such that $f = f' \circ \mathtt{cfpow}_\Gamma^{(k+1)}$.*

Proof. By induction on $\mathrm{rk}(f)$ (with an inductive hypothesis that quantifies over k).

Base case (f regular). Consider the unique $\varphi \in \mathrm{Hom}((\{0, \dots, \mathrm{rk}(f)\} \times \Gamma)^*, \Gamma^*)$ such that for every $c \in \Gamma$, $\varphi(k, c) = c$ and $\varphi(m, c) = \varepsilon$ when $m < k$. Since regular functions are closed under composition, $f' = f \circ \varphi$ is regular, and the desired equation follows from the fact that $\varphi \circ \mathtt{cfpow}_\Gamma^{(k+1)} = \mathrm{id}_{\Gamma^*}$.

Inductive case. Let $f = \mathrm{CbS}(g, (h_i)_{i \in I})$ with $g : \Gamma^* \to I^*$ regular and $h_i : \Gamma^* \to \Sigma^*$ cfp such that $\mathrm{rk}(h_i) \leq \mathrm{rk}(f) - 1$ for all $i \in I$. Using the inductive hypothesis, we know that $h_i = h'_i \circ \mathtt{cfpow}_\Gamma^{(k)}$ for some family of regular functions $(h'_i)_{i \in I}$. Thus, let us assume we are given 2DFTs $\mathcal{T}$ and $(\mathcal{T}'_i)_{i \in I}$ corresponding to g and the family $(h'_i)_{i \in I}$. Without loss of generality, let us assume further that $\mathcal{T}$ always output at most one letter at each transition, never outputs a letter upon reading $\triangleleft$, and that the $\mathcal{T}_i$ always terminate on the marker $\triangleright$ by a transition that does not move the reading head. With these assumptions, let us describe informally a 2DFT $\mathcal{T}''$ corresponding to the function f' such that

$$\mathrm{CbS}(g, (h'_i \circ \mathtt{cfpow}_\Gamma^{(k)})_{i \in I}) = f' \circ \mathrm{CbS}(g, (h'_i)_{i \in I}) \circ \mathtt{cfpow}_\Gamma^{(k+1)}$$

Assuming that the state space of $\mathcal{T}$ is Q and the state space of $\mathcal{T}'_i$ is Q'_i, with Q and the Q'_is all pairwise disjoint, we take the state space of $\mathcal{T}''$ to be

$$Q'' \;=\; Q \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \times \left(\{\bullet\} \sqcup \bigcup_{i \in I} Q'_i \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \right)$$

with initial state $(q_0, \mathsf{R}, \bullet)$, if q_0 is the initial state of $\mathcal{T}$ and final states the triples $(q_f, \mathsf{M}, \bullet)$ such that q_f is a final state of $\mathcal{T}$. To guide intuitions, the elements L, R and S should be respectively read as "left", "right" and "stay". With this in mind, the high-level description of computations carried out by $\mathcal{T}''$ over words $\mathtt{pow}_\Gamma^{(k)}$ is as follows.

- When in a state $(q_0, \mathsf{M}, \bullet)$, $\mathcal{T}''$ essentially acts as $\mathcal{T}$ on letters of the shape (k, a) or end-markers and ignores letters (l, a) for $l < k$; the central component M then determines whether to seek the next relevant position to the left or to the right when reading such an irrelevant letter. This continues up until upon reading a letter (k, a) in state $(q, \mathsf{M}, \bullet)$ such that $\mathcal{T}$ would ouput i when reading a in q, $\mathcal{T}''$ moves into the state $(r, \mathsf{M}', (q'_{0,i}, \mathsf{R}))$ where $q'_{0,i}$ is the initial state of $\mathcal{T}'_i$ and (r, M') is determined by the transition in $\mathcal{T}$.
- When in a state $(q, \mathsf{M}, (q'_i, \mathsf{M}'))$ for $q'_i \in Q'_i$, $\mathcal{T}$ behaves exactly as $\mathcal{T}_i$ as long as the current transition does not reach the final state, treating letters outside of its input alphabets as end markers; this is possible because of the component M' of the state, that we use to keep track of the last move of the reading head. Meanwhile the components $q \in Q$ and M are untouched. When a final transition is taken, by our assumption we return control to $\mathcal{T}$ by going to state $(q, \mathsf{M}, \bullet)$ and moving in the direction prescribed by M (recall that, by assumption, we are moving away from (or staying in) the position at which $\mathcal{T}_i$ started running).

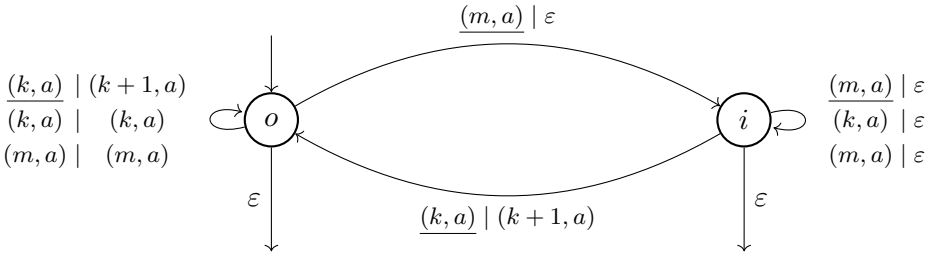
We leave formalizing this definition and checking that $\mathcal{T}''$ has a desirable behaviour to the reader. □

Lemma 3.3.24. *For every $k \in \mathbb{N}$, $\mathtt{cfpow}_\Gamma^{(k)}$ is equal to a a composition of sequential functions and squaring functions $\mathtt{cfsquaring}_\Delta$.*

PROOF. We proceed by induction over k. The cases of $k = 0, 1, 2$ are immediate as $\mathtt{cfpow}_\Gamma^{(k)}$ then corresponds, up to isomorphism of output alphabet, to a constant function, the identity and $\mathtt{cfsquaring}_\Gamma$ respectively, so we focus on the inductive step. To achieve the desired result, it suffices to show that there exists a sequential function

$$f : (\{0, \dots, k\} \times \Gamma) \cup \underline{(\{0, \dots, k\} \times \Gamma)}^* \to (\{0, \dots, k+1\} \times \Gamma)^*$$

such that $\mathtt{cfpow}_\Gamma^{(k+1)} = f \circ \mathtt{cfsquaring}_{\{0,\dots,k\}\times\Gamma} \circ \mathtt{cfpow}_\Gamma^{(k+1)}$. In fact, the sequential transducer pictured below computes such an f:



where m designates any element of $\{0, \dots, k-1\}$. □

Now we turn to the proofs of our main theorems.

PROOF OF THEOREM 3.3.1. The direct implication is obtained by combining the two lemmas above: every cfp function can be written as a composition $f \circ \mathtt{cfpow}_\Gamma^{(k)}$ for some $k \in \mathbb{N}$ and f regular by Lemma 3.3.23, and Lemma 3.3.24 guarantees that in turn, $\mathtt{cfpow}_\Gamma^{(k)}$ is a composition of sequential (and a fortiori regular) functions and squarings. Conversely, that cfp functions are closed under composition is proven in Section 3.3.1, which is enough to conclude as regular functions and $\mathtt{cfsquaring}_\Gamma$ are cfp. □

PROOF OF THEOREM 3.3.2. We prove the circle of implications (i) ⇒ (iii) ⇒ (ii) ⇒ (i). (The claim after this equivalence has already been established in Theorem 3.3.9.)

The first implication (i) ⇒ (iii) corresponds exactly to Lemma 3.3.23 we just proved.

The implication (iii) ⇒ (ii) is also relatively easy: $\mathtt{cfpow}_\Gamma^{(k)}$ is comparison-free polyregular (this is a consequence of Lemma 3.3.24 and Theorem 3.3.1, although $\mathtt{cfpow}_\Gamma^{(k)}$ can also be shown to fit Definition 3.1.2 in a more elementary way) and so is $f \circ \mathtt{cfpow}_\Gamma^{(k+1)}$ by Theorem 3.3.1 for f regular. Furthermore, $|\mathtt{cfpow}_\Gamma^{(k+1)}(w)| = O(|w|^{k+1})$ and, since f is regular, $|f(u)| = O(|u|)$, so we have, as expected, $|(f \circ \mathtt{cfpow}_\Gamma^{(k+1)})(w)| = O(|w|^{k+1})$.

The final implication (ii) ⇒ (i) is technically the hardest as it relies on Theorem 3.3.9. Let $f : \Gamma^* \to \Sigma^*$ be cfp and $k \in \mathbb{N}$ such that $|f(w)| = O(|w|^{k+1})$. If f is regular, then $\mathrm{rk}(f) = 0 \leq k$. Otherwise, by Theorem 3.3.9, there exists a sequence $(w_n)_{n\in\mathbb{N}}$ of inputs such that $|w_n| = O(n)$ and $|f(w_n)| \geq n^{\mathrm{rk}(f)+1}$. So $n^{\mathrm{rk}(f)+1} = O(n^{k+1})$, hence $\mathrm{rk}(f) \leq k$. □

3.4. COMPARISON-FREE POLYREGULAR SEQUENCES

Now that we have given general characterizations of cfp functions, it can be interesting to consider degenerate cases. We shall look here at functions with *unary input alphabet*. (Dually, Douéneau-Tabot has recently studied polynomial growth transductions – including cfp functions – with unary *output* [Dou21].) By identifying $\mathbb{N}$ with the set of words $\{a\}^*$, via the canonical isomorphism $n \mapsto a^n$, functions $\{a\}^* \to \Gamma^*$ can be seen as *sequences*.

We shall perform this identification implicitly and speak freely, for instance, of word sequences $\mathbb{N} \to \Gamma^*$ that are comparison-free polyregular. It turns out that the latter admit a characterization as finite combinations of what we call "poly-pumping sequences".

Definition 3.4.1. A *poly-pumping sequence* is a function of the form $[\![e]\!] : \mathbb{N} \to \Sigma^*$ where
- e is a *polynomial word expression* generated by $e ::= w \mid e \cdot e' \mid e^*$ where $w \in \Sigma^*$;
- $[\![w]\!](n) = w$, $[\![e \cdot e']\!](n) = [\![e]\!](n)[\![e']\!](n)$ and $[\![e^*]\!](n) = ([\![e]\!](n))^n$.

The *star-height* of a polynomial word expression is defined in the usual way.

Theorem 3.4.2. *Let $s : \mathbb{N} \to \Sigma^*$ and $k \in \mathbb{N}$. The sequence s is comparison-free polyregular with $\mathrm{rk}(s) \leq k$ if and only if there exists $p > 0$ such that, for any $m < p$, there is a polynomial word expression e of star-height at most $k+1$ such that $\forall n \in \mathbb{N},\ s((n+1)p+m) = [\![e]\!](n)$.*

In short, the comparison-free polyregular sequences are exactly the *ultimately periodic combinations* of poly-pumping sequences. Our proof strategy is an induction on k. The base case $k = 0$ characterizes regular sequences as ultimately periodic combinations:

Lemma 3.4.3. *A sequence of words $s : \mathbb{N} \to \Sigma^*$ is regular if and only if there is $m > 0$ such that for every $k < m$, there are words $u_0, \dots, v_l, v_1, \dots, v_l$ such that for every $n \in \mathbb{N}$, we have*

$$\forall n \in \mathbb{N},\ s((n+1)m+k) = u_0(v_1)^n \dots (v_l)^n u_l$$

A result that is essentially equivalent to the above lemma is stated with a proof sketch using 2DFTs by Choffrut in [Cho17, p. 90];[8] we propose an alternative proof using copyless SSTs. (Non-deterministic two-way transducers (2NFTs) taking unary inputs have also been studied [Gui16]; furthermore, the notion of "k-iterative language" that appears in a pumping lemma for general 2NFTs [Smi14] (which actually dates back to [Roz86]) is related to the shape of the above pumping sequences.)

The proof of the base case of Theorem 3.4.2 concentrates the bulk of the reasoning. To make the inductive step go through, it is enough to synchronize the periods of the different poly-pumping sequences involved and to observe that $\mathrm{CbS}([\![e]\!], ([\![e'_i]\!])_{i \in I})$ is realized by an expression obtained by substituting the e'_i for i in e.

Thanks to Theorem 3.4.2, we can perform induction over expressions to work with cfp sequences. One example of application, involving the **map** operation from §2.5.3, is the following (contrast with our later Corollary 3.5.5):

Corollary 3.4.4. *If $f : \Gamma^* \to \Sigma^*$ and $s : \mathbb{N} \to (\Gamma \cup \{\#\})^*$ are cfp, so is $\mathbf{map}(f) \circ s$.*

Theorem 3.4.2 will also be applied later in §3.5 to prove a separation result. Before that, the next 3 subsections prove Lemma 3.4.3, Theorem 3.4.2 and Corollary 3.4.4 respectively.

3.4.1. **Proof of Lemma 3.4.3.** The "if" direction is straightforward, so we only prove the "only if" part of the statement. To keep notations harmonized, let us work with $f : \{a\}^* \to \Sigma^*$ such that $s(n) = f(a^n)$ for every $n \in \mathbb{N}$ and fix a copyless SST computing f whose set of states, set of registers and transition function we call Q, R and δ respectively. We use the monoid $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q$ introduced in Section 2.3.4, which contains $\mu = \mathtt{erase}_\Sigma(\delta(-, a))$. Since $\mathcal{M}^{\mathrm{cl}}_{R,\varnothing} \wr Q$ is finite (Proposition 2.3.24), there is an exponent $m \in \mathbb{N} \setminus \{0\}$ such that $\mu^{\bullet m} = \mu \bullet \dots (m \text{ times}) \dots \bullet \mu$ is idempotent, i.e. $\mu^{\bullet m} = \mu^{\bullet 2m}$. This m is the one put forth in the lemma statement.

Let us fix $k < m$. Let $(q, \alpha) = \mu^{\bullet m}(q_0)$ where q_0 is the initial state of the SST. We have $\mu^{\bullet(m+k)} \bullet \mu^{\bullet m} = \mu^{\bullet(2m+k)} = \mu^{\bullet k} \bullet \mu^{\bullet 2m} = \mu^{\bullet k} \bullet \mu^{\bullet m} = \mu^{\bullet(m+k)}$ as usual. Therefore,

[8] In [Cho17], the result is attributed to "Eugenia", a hobbyist Italian researcher with a day job at a software company. Disappointingly, she is actually a fictional character (personal communication from Christian Choffrut, relayed by Charles Paperman).

$\mu^{\bullet m}(q) = (q, \beta)$ with $\alpha \bullet \beta = \alpha$ and $\beta \bullet \beta = \beta$ (the latter is because of $\mu^{\bullet 2m} = \mu^{\bullet m}$). Thus, q is the state reached by the SST after reading $a^{m(n+1)+k}$ for any $n \in \mathbb{N}$. We also have $(\delta(-, a))^{\bullet m}(q) = (q, \gamma)$ with $\gamma \in \mathcal{M}^{\mathrm{cl}}_{R,\Sigma}$ and $\mathtt{erase}_\Sigma(\gamma) = \beta$.

Given $r \in R$, we distinguish two cases.

- First, suppose that $\beta(r) = \varepsilon$ or equivalently that $\gamma(r) \in \Sigma^*$ (in general, the codomain of γ is $(\Sigma \cup R)^*$). When the SST is in state q and reads a^m, it executes the assignment γ; when $\beta(r) = \varepsilon$, the new value of the register r is this $\gamma(r) \in \Sigma^*$ which does not depend on the old value of any register. Therefore, for all $n \in \mathbb{N}$, the content of the register r after having read $a^{m(n+1)+k}$ (starting from the initial configuration) is the constant $\gamma(r)$.
- We now treat the case where $\beta(r)$ is non-empty. By definition, $\beta \bullet \beta = \beta^* \circ \beta$ where $\beta^* \in \mathrm{Hom}(R^*, R^*)$ extends $\beta : R \to R^*$. Since we know, as a consequence of the idempotency of $\mu^{\bullet m}$, that $\beta \bullet \beta = \beta$, we have $\beta^*(\beta(r)) = \beta(r) \neq \varepsilon$.

  Let us study in general the situation $\beta^*(\rho) = \beta(r) \neq \varepsilon$ for $\rho \in R^*$. A first observation is that the letters in $\beta(r)$ cannot be found in any other $\beta(r')$ for $r' \in R \setminus \{r\}$ because β is copyless, so $\rho \notin (R \setminus \{r\})^*$. We therefore have $n \geq 1$ occurrences of r in ρ, so $\rho = \rho_0 r \dots r \rho_n$ with $\rho_0, \dots, \rho_n \notin (R \setminus \{r\})^*$. By coming back to $\beta^*(\rho) = \beta(r)$, into which we plug this expression for ρ, and using the fact that $\beta(r)$ has non-zero length, we can see that $n = 1$ and $\beta^*(\rho_0) = \beta^*(\rho_1) = \varepsilon$.

  Let us apply this to $\rho = \beta(r) = \mathtt{erase}_\Sigma(\gamma)(r)$ and lift the result to $\gamma(r)$:

$$\gamma(r) = u_r r v_r \quad \text{for some} \quad u_r, v_r \in (\Sigma \cup \beta^{-1}(\{\varepsilon\}))^*$$

  In the previous case ($\beta(r') = \varepsilon$ for $r' \in R$), we saw that $\gamma(\beta^{-1}(\{\varepsilon\})) \subseteq \Sigma^*$. Therefore $\gamma^\star(u_r), \gamma^\star(v_r) \in \Sigma^*$, where $\gamma^\star \in \mathrm{Hom}((\Sigma \cup R)^*, (\Sigma \cup R)^*)$ extends $\gamma : R \to (\Sigma \cup R)^*$ by being the identity on Σ. Since Σ^* is fixed by $\gamma^\star$, when we iterate, we obtain

$$\gamma^{\bullet(n+1)}(r) = (\gamma^\star)^n \circ \gamma(r) = (\gamma^\star(u_r))^n \cdot u_r r v_r \cdot (\gamma^\star(v_r))^n$$

Now, let F be the final output function of the SST that computes f, and $\vec{w}_{m+k}$ be the register values after it has read a prefix a^{m+k}. Then after reading $a^{m(n+1)+k}$, the new register values are $(\gamma^{\bullet(n+1)})^\dagger(\vec{w}_{m+k})$. More precisely, the register r contains:

- $\gamma(r) \in \Sigma^*$ if $\beta(r) = \varepsilon$;
- $(\gamma^\star(u_r))^n \cdot ((u_r r v_r)^\dagger(\vec{w}_{m+k})) \cdot (\gamma^\star(v_r))^n$ otherwise.

These values are combined by $F(q)^\dagger$ – where q is the recurrent state we have been working with all along, and F is the final output function – to produce the output $f(a^{m(n+1)+k})$. This yields the desired shape: an interleaved concatenation of finitely many factors that are either constant, $(\gamma^\star(u_r))^n$ or $(\gamma^\star(v_r))^n$ for some $r \in R$.

3.4.2. **Proof of Theorem 3.4.2.** We proceed by induction on the rank of the sequence $s : \mathbb{N} \to \Sigma^*$ under consideration. If the rank of s is 0, it is regular and we apply Lemma 3.4.3 and the desired polynomial word expression is of the shape $u_0 \cdot (v_1)^* \dots (v_l)^* \cdot u_l$.

If the rank of s is $k+1$, thanks to the induction hypothesis and the base case above, it can be written as $\mathrm{CbS}([\![e]\!], ([\![e'_i]\!])_{i \in I})$ where e is an expression over the alphabet I with star-height at most one and the e'_is expressions over Σ with star-height at most k. Without loss of generality, we may assume that that terminal nodes of polynomial word expressions are words of length at most one. For such an expression over alphabet I, one may define inductively the following *substitution operation* to obtain an expression of Σ^*:

$$\begin{aligned} j[(e'_i)_{i\in I}] &= e'_j & \varepsilon[(e'_i)_{i\in I}] &= \varepsilon \\ (f \cdot f')[(e'_i)_{i\in I}] &= f[(e'_i)_{i\in I}] \cdot f'[(e'_i)_{i\in I}] & f^*[(e'_i)_{i\in I}] &= (f[(e'_i)_{i\in I}])^* \end{aligned}$$

One can then check by induction on the structure of e that $[\![e[(e'_i)_{i\in I}]]\!] = \mathrm{CbS}([\![e]\!], ([\![e'_i]\!])_{i\in I})$ and that $e[(e'_i)_{i\in I}]$ has star-height bounded by $k+1$.

3.4.3. **Proof of Corollary 3.4.4.** We first prove the result for poly-pumping sequences.

Lemma 3.4.5. *If $[\![e]\!] : \mathbb{N} \to (\Gamma \cup \{\#\})^*$ is a poly-pumping sequence and $f : \Gamma^* \to \Sigma^*$ is comparison-free polyregular, then* $\mathbf{map}(f) \circ [\![e]\!]$ *is a cfp sequence.*

For the rest of this subsection, we write S for the successor function $n \mapsto n+1$ over $\mathbb{N}$. We will use the fact that s is a cfp sequence iff $s \circ \mathsf{S}$ also is.

Proof. We first note that if the separator $\#$ does not occur at any leaf of e, then the result is immediate as we would have $\mathbf{map}(f) \circ [\![e]\!] = f \circ [\![e]\!]$. We thus focus on the cases when it does occur, and proceed inductively over e.

- If $e = w \in (\Gamma \cup \{\#\})^*$, then $\mathbf{map}(f) \circ [\![e]\!]$ is a constant sequence, which is obviously cfp.
- If $e = (e')^*$, with $\#$ occuring in e', let $h_{\mathrm{l}}, h_{\mathrm{r}} : \mathbb{N} \to \Gamma^*$ and $h_{\mathrm{c}} : \mathbb{N} \to (\Gamma \sqcup \{\#\})^*$ be the sequences such that
$$[\![e']\!] \circ \mathsf{S} \circ \mathsf{S} = h_{\mathrm{l}} \cdot \# \cdot h_{\mathrm{c}} \cdot \# \cdot h_{\mathrm{r}}$$
with $h_{\mathrm{l}}(n)$ being the largest $\#$-free prefix of $[\![e']\!](n+2)$ and $h_{\mathrm{r}}(n)$ the largest $\#$-free suffix of $[\![e']\!](n+2)$. There is a regular function
$$\begin{array}{rccl} f' : & (\Gamma \sqcup \{\#\})^* & \to & (\Gamma \sqcup \{\#\})^* \\ & w_0 \# w_1 \# \dots w_{n-1} \# w_n & \mapsto & w_1 \# \dots w_{n-1} \quad (w_0, \dots, w_n \in (\Gamma \sqcup \underline{\Gamma})^*) \end{array}$$
stripping away the first and last component of its input, so that it satisfies
$$f' \circ \mathbf{map}(f) \circ [\![e']\!] \circ \mathsf{S} \circ \mathsf{S} \;=\; \mathbf{map}(f) \circ h_{\mathrm{c}}$$
By the inductive hypothesis, we know that $\mathbf{map}(f) \circ [\![e']\!]$ is comparison-free polyregular. We may therefore conclude by composition (cf. Theorem 3.3.1) that $\mathbf{map}(f) \circ h_{\mathrm{c}}$ is cfp. One can check analogously that h_{l} and h_{r} are also cfp. Then observe that
$$\begin{array}{rcl} ([\![e]\!] \circ \mathsf{S} \circ \mathsf{S})(n) & = & (h_{\mathrm{l}} \cdot \# \cdot h_{\mathrm{c}} \cdot \# \cdot h_{\mathrm{r}})(n)^{n+2} \\ & = & (h_{\mathrm{l}} \cdot (\# \cdot h_{\mathrm{c}} \cdot \# \cdot h_{\mathrm{r}} \cdot h_{\mathrm{l}})^{n+1} \cdot \# \cdot h_{\mathrm{c}} \cdot \# \cdot h_{\mathrm{r}})(n) \end{array}$$
$$\text{which means that} \quad \mathbf{map}(f) \circ [\![e]\!] \circ \mathsf{S} \circ \mathsf{S} = \left\{ \begin{array}{c} (f \circ h_{\mathrm{l}}) \\ \cdot \\ (\# \cdot (\mathbf{map}(f) \circ h_{\mathrm{c}}) \cdot \# \cdot (f \circ (h_{\mathrm{r}} \cdot h_{\mathrm{l}})))^* \\ \cdot \\ \# \cdot (\mathbf{map}(f) \circ h_{\mathrm{c}}) \cdot \# \cdot (f \circ (h_{\mathrm{r}} \cdot h_{\mathrm{l}})) \\ \cdot \\ \# \cdot (\mathbf{map}(f) \circ h_{\mathrm{c}}) \cdot \# \cdot (f \circ h_{\mathrm{r}}) \end{array} \right.$$
Thanks again to the closure under composition, each component of this expression is cfp, so $\mathbf{map}(f) \circ [\![e]\!] \circ \mathsf{S} \circ \mathsf{S}$ is also cfp. Hence, so is $\mathbf{map}(f) \circ [\![e]\!]$.
- The last case where $e = e' \cdot e''$ is handled similarly after a case analysis determining whether $\#$ occurs only in e', e'' or in both; we leave it to the reader. □

To finish proving Corollary 3.4.4, suppose that we are given cfp functions $f : \Gamma^* \to \Sigma^*$ and $s : \mathbb{N} \to (\Gamma \cup \{\#\})^*$. By Theorem 3.4.2, there are $m > 0$ and some expressions $e_0, \dots, e_{m-1}$ such that $s(m(n+1)+k) = [\![e_k]\!](n)$ for every $k < m$. By Lemma 3.4.5, every $\mathbf{map}(f) \circ [\![e_k]\!]$ is cfp. The set $L_k = \{m(n+1)+k \mid n \in \mathbb{N}\}$ is semi-linear, i.e., corresponds to a regular language, and there are regular sequences $r_k : \mathbb{N} \to \mathbb{N}$ such that $r_k(m(n+1)+k) = n$. Further, $\mathbb{N} = \{n \mid n < m\} \cup \bigcup_{k<m} L_k$, so we may use the regular conditional provided by Proposition 3.1.8 to show that the combination of the $\mathbf{map}(f) \circ [\![e_k]\!] \circ r_k$ and the first m values of $\mathbf{map}(f) \circ s$, which corresponds exactly to $\mathbf{map}(f) \circ s$, is indeed cfp.

3.5. Separation results

Let us now give concrete functions that witness the separations claimed at the beginning of the chapter: the class of comparison-free polyregular functions is incomparable with the class of HDT0L transductions and is a strict subclass of polyregular functions.

Theorem 3.5.1. *There exist comparison-free polyregular functions which are not HDT0L:*

(i) $a^n \in \{a\}^* \mapsto (a^n b)^{n+1} \in \{a,b\}^*$ *for* $a \neq b$;
(ii) $w \in \Sigma^* \mapsto w^{|w|}$ *for* $|\Sigma| \geq 2$ *(a simplification of Example 3.1.3);*
(iii) $a^n \# w \in \Sigma^* \mapsto (w\#)^n$ *for* $a, \# \in \Sigma$, $a \neq \#$ (from [DFG20, Section 6])

Note that in the last example, we have not fully specified the function, but we claim that a function that obeys this condition cannot be an HDT0L transduction, and that there exists a cfp function that satisfies it.

Remark 3.5.2. The first example in [DFG20, §5] shows that $a^n \mapsto a^{n \times n}$ is HDT0L (via the equivalent model of marble transducers), hence the necessity of $|\Sigma| \geq 2$ above. More generally, Douéneau-Tabot has shown very recently that *every polyregular function with unary output is HDT0L* [Dou21]. So polyregular functions with unary output coincide with *polynomial growth* $\mathbb{N}$*-rational series* (cf. Remark 2.3.15), and the latter admit several algebraic characterizations in the literature (see [Reu79] and [BR10, Chapter 9, Exercise 1.2]).

Theorem 3.5.3. *Some HDT0L transductions are polyregular but not comparison-free:*

(i) $f : a^n \in \{a\}^* \mapsto ba^{n-1}b \dots baabab$ *(with* $f(\varepsilon) = \varepsilon$ *and* $f(a) = b$*);*
(ii) $\mathbf{map}(a^n \mapsto a^{n \times n}) : a^{n_1} \# \dots \# a^{n_k} \mapsto a^{n_1 \times n_1} \# \dots \# a^{n_k \times n_k}$ *(cf. Definition 2.5.8).*

Remark 3.5.4. The function $a^{n_1} \# \dots \# a^{n_k} \mapsto a^{n_1 \times n_1 + \dots + n_k \times n_k}$ obtained by erasing the #s in the output of $\mathbf{map}(a^n \mapsto a^{n \times n})$ is also *not* comparison-free. This result implies the second item of Theorem 3.5.3 by composition with the erasing morphism; we do not prove it here, but it appears in Douéneau-Tabot's aforementioned paper [Dou21]. Therefore, according to [Dou21], *not* every polyregular function with unary output is comparison-free.

To see why the first of the two functions in Theorem 3.5.3 is HDT0L, observe that it is Example 2.3.4 for $\Gamma = \{a\}$ (taking $b = \underline{a}$); as for the second one, combine Remark 3.5.2 and Proposition 2.5.9.

The non-membership parts of Theorems 3.5.1 and 3.5.3 require more work. For the former, we use pumping arguments on HDT0L systems; the proofs are detailed in Section 3.5.1. Item (ii) of Theorem 3.5.3 is handled in §3.5.2 by first appealing to Theorem 3.3.2 to reduce to showing that $\mathbf{map}(a^n \mapsto a^{n \times n}) \neq \mathrm{CbS}(g, (h_i)_{i \in I})$ when g and all the h_i are *regular* functions; a combination of pumping and of a combinatorial argument then shows that inputs with $|I|$ occurrences of # suffice to discriminate the two sides of the inequality. This result also has the following immediate consequence:

Corollary 3.5.5. *Comparison-free polyregular functions are not closed under* **map**.

Compare with Corollary 3.4.4, and see the end of Section 2.5.3 for a discussion of the significance of this fact.

As for item (i) of Theorem 3.5.3, we apply our characterization of cfp sequences from the previous section. We show that $a^n \mapsto ba^{n-1}b \dots bab$ is not comparison-free polyregular by proving that its subsequences are *not* poly-pumping: for every poly-pumping sequence $s : \mathbb{N} \to \{a,b\}^*$, there is a uniform bound on the number of distinct contiguous subwords of the shape $baa \dots ab$ occuring in each $s(n)$ for $n \in \mathbb{N}$. Let us formalize this.

Definition 3.5.6. Let Σ be a finite alphabet and $c \in \Sigma$. Call $\beta_c : \Sigma^* \to \mathcal{P}(\mathbb{N})$ the function assigning to a word w the set of lengths of its maximal factors lying in $\{c\}^*$ (including ε):

$$\beta_c(w) \;=\; \{k \in \mathbb{N} \mid w \in (\Sigma^* \setminus (\Sigma^* \cdot c)) \cdot c^k \cdot (\Sigma^* \setminus (c \cdot \Sigma^*))\}$$

We say that a sequence $s : \mathbb{N} \to \Sigma^*$ is *poly-uniform* if for every $c \in \Sigma$ there exists a *finite* set of polynomials $A_{s,c} \subseteq \mathbb{Q}[X]$ such that, for every $n \in \mathbb{N}$,

$$\beta_c(s(n)) \;\subseteq\; A_{s,c}(n) = \{P(n) \mid P \in A_{s,c}\}$$

Lemma 3.5.7. *Every comparison-free polyregular sequence is poly-uniform.*

Proof. First, observe that any ultimately periodic combination of poly-uniform sequences is poly-uniform. Indeed, assume that we have such a sequence s and $m > 0$ so that $n \mapsto s(m(n+1)+k)$ is poly-uniform for every k, and finite sets $A_{k,c} \subseteq \mathbb{Q}[X]$ so that $\beta_c(s(m(n+1)+k)) \subseteq A_{k,c}(n)$. Then the poly-uniformity of s is witnessed by

$$A_{s,c} \;=\; \bigcup_{l<m} \left\{ P\left(\frac{X-l}{m}\right) \mid P \in A_{k,c} \right\} \cup \beta_c(s(l))$$

Hence, by Theorem 3.4.2, it suffices to show that poly-pumping sequences are all poly-uniform. We proceed by induction over polynomial word expressions e, defining suitable finite sets of polynomials $A_{e,c}$ for $c \in \Sigma$ such that $\beta_c([\![e]\!](n)) \subseteq A_{e,c}(n)$ and $0 \in A_{e,c}$:

$$\begin{array}{lcl@{\qquad\qquad}lcl} A_{e\cdot e',c} & = & \{P+Q \mid (P,Q) \in A_{e,c} \times A_{e',c}\} & A_{w,c} & = & \beta_c(w) \cup \{0\} \\ A_{e^*,c} & = & A_{e,c} \cup \{XP \mid P \in A_{e,c}\} & & & \end{array}$$

□

Proof of Theorem 3.5.3 item (i). Observe that for $f : a^n \mapsto ba^{n-1}b \dots bab$, the sequence of sets $\beta_c(f(a^n)) = \{0, \dots, n-1\}$ had unbounded cardinality, and thus cannot be covered by a finite set of functions, let alone polynomials in $\mathbb{Q}[X]$. □

As announced before, the rest of this section gives the remaning proofs.

3.5.1. Proof of Theorem 3.5.1.

These examples are comparison-free. We have seen in Example 3.1.3 that $w \mapsto w^{|w|}$ is a comparison-free polyregular function. For the other examples:

- $(a^n \mapsto (a^n b)^{n+1}) = \mathrm{CbS}((a^n \mapsto a^{n+1}), (a^n \mapsto a^n b)_{i \in \{a\}})$ is obtained as a composition by substitution of sequential functions (cf. Section 2.2), which are in particular regular;
- for an alphabet Σ with $a, \# \in \Sigma$, there exist sequential functions $f : \Sigma^* \to \{a\}^*$ and $g : \Sigma^* \to \Sigma^*$ such that $f(a^n \# w) = a^n$ and $g(a^n \# w) = w\#$ for $n \in \mathbb{N}$ and $w \in \Sigma^*$, so that $\mathrm{CbS}(f, (g)_{i \in \{a\}})(a^n \# w) = (w\#)^n$.

(i) is not HDT0L. Let us fix a HDT0L system $(\{a\}, \{a,b\}, \Delta, d, (h)_{i \in \{a\}}, h')$ and show that it does not compute $a^n \mapsto (a^n b)^{n+1}$. Let $\texttt{letters}(w)$ be the set of letters occurring in the string w at least once. By the infinite pigeonhole principle, there exists an infinite $X \subseteq \mathbb{N}$ such that $\texttt{letters}(h^n(d))$ has the same value Δ' for all $n \in X$. Let us do a case analysis:

- Suppose first that for some $r \in \Delta'$ and some $m \in \mathbb{N}$, the letter b appears twice in $h' \circ h^m(r)$; in other words, that the latter contains a factor $ba^k b$ for some $k \in \mathbb{N}$. Then for all $n \in X$, $h' \circ h^{m+n}(d) \in \Sigma^* ba^k b \Sigma^*$. Since X is infinite, this holds for some n such that $m+n > k$, so that this word – i.e. the output of the HDT0L system for a^{m+n} – is different from $(a^{m+n}b)^{m+n+1} \notin \Sigma^* ba^k b \Sigma^*$.

- Otherwise, for all $r \in \Delta'$ (that includes the degenerate case $\Delta' = \varnothing$) and all $m \in \mathbb{N}$, there is at most one occurrence of b in $h' \circ h^m(r)$. Then for all $m \in \mathbb{N}$, the length of $h^{\min(X)}(d)$ bounds the number of occurrences of b in $h' \circ h^{m+\min(X)}(d)$, and this bound is independent of m. On the contrary, in the sequence $((a^n b)^{n+1})_{n \geq m+\min(X)}$, the number of occurrences of b is unbounded.

(ii) is not HDT0L. The second counterexample, namely $w \mapsto w^{|w|}$, reduces to the first one: indeed, $(a^n b)^{n+1} = (a^n b)^{|a^n b|}$ for all $n \in \mathbb{N}$, which can also be expressed as

$$(w \mapsto w^{|w|}) \circ (u \in \{a\}^* \mapsto ub) = (a^n \mapsto (a^n b)^{n+1})$$

Suppose for the sake of contradiction that there is a HDT0L system $(\Sigma, \Sigma, \Delta, d, (h_c)_{c \in \Sigma}, h')$ that computes $w \mapsto w^{|w|}$ with $|\Sigma| \geq 2$; we may assume without loss of generality that $a, b \in \Sigma$. Then $(\{a\}, \{a, b\}, \Delta, h_b(d), (h_a)_{c \in \{a\}}, h')$ computes $a^n \mapsto (a^n b)^{n+1}$.

(iii) is not HDT0L. (This is claimed without proof in [DFG20, Section 6].)

Let $\Sigma \supseteq \{a, \#\}$ be an alphabet and let $(\Sigma, \Sigma, \Delta, d, (h_c)_{c \in \Sigma}, h')$ be a HDT0L system. We reuse a similar argument to our treatment of the counterexample **(i)**. Let the sets $\Delta' \subseteq \Delta$ and $X \subseteq \mathbb{N}$ with X infinite be such that $\texttt{letters}(h_a^n(d)) = \Delta'$ for all $n \in X$.

- Suppose first that for some $r \in \Delta'$ and some $m \in \mathbb{N}$, the string $h' \circ h_a^m \circ h_\#(r)$; contains a factor $\# \cdot a^k \cdot \#$ for some $k \in \mathbb{N}$. Then for all $n \in X$, the given HDT0L system maps $a^m \# a^n$ to a string in $\Sigma^* \cdot \# \cdot a^k \cdot \# \cdot \Sigma^*$. For $n > k$, this language does not contain $(a^n \#)^m$; such a $n \in X$ exists because X is infinite.
- Otherwise, for any $m \in \mathbb{N}$, since $\#$ occurs at most once in $h' \circ h_a^m \circ h_\#(r)$ for $r \in \Delta'$, the output of the HDT0L system has at most $|h^{\min(X)}(d)|$ occurrences of $\#$ on input $a^m \# a^{\min(X)}$. Therefore, for large enough m, this output is different from $(a^{\min(X)} \#)^m$.

3.5.2. **Proof of Theorem 3.5.3 item (ii).** Suppose for the sake of contradiction that $f = \mathbf{map}(a^n \mapsto a^{n \times n})$ is comparison-free. Using Theorem 3.3.2, it must then have rank 1 since $|f(w)| = O(|w|^2)$. Thus, we may write $f = \mathrm{CbS}(g, (h_i)_{i \in I})$ where $g : \{a, \#\}^* \to I^*$ and all the $h_i : \{a, \#\}^* \to \{a, \#\}^*$ are regular.

For each $J \subseteq I$ and $k \in \{0, \ldots, |I|\}$ (though the definition would make sense for $k \in \mathbb{N}$), let $\rho_{J,k} : \{a^*\} \to (I \setminus J)^*$ be uniquely defined by the condition

$$\forall w \in \{a, \#\}^*,\ \rho_{J,k}(w) = \begin{cases} s & \text{when} \quad g(w) \in ((I \setminus J)^* J)^k \cdot s \cdot (\{\varepsilon\} \cup J I^*) \\ \varepsilon & \text{when} \quad |g(w)|_J < k \end{cases}$$

(recall from Section 3.3.2 the notation $|\cdot|_J$). To put it plainly, $\rho_{J,k}(w)$ is the k-th block of letters from $I \setminus J$ that appears in $g(w)$ (the block may be the empty string if there are consecutive letters from J), or the empty string if this k-th block does not exist. The function $\rho_{J,k}$ is regular because it is the composition of a sequential function with g.

We reuse some tools from Section 3.3.2, especially the notion of producing 1-split from Lemma 3.3.15. There is a unique sensible way to combine the morphisms $\nu_{f'} : \{a, \#\}^* \to \mathcal{N}(f')$ given by this lemma into a morphism

$$\varphi : \{a, \#\}^* \to \prod_{f' \in \mathcal{F}} \mathcal{N}(f') \quad \text{for} \quad \mathcal{F} = \{g\} \cup \{h_i \mid i \in I\} \cup \{\rho_{J,k} \mid J \subseteq I,\ k \in \{0, \ldots, |I|\}\}$$

Note that the codomain above is a finite monoid: this allows us to apply Proposition 3.3.16 to this morphism φ and $r = 1$, which gives us some $N \in \mathbb{N}$. Let $s = a^N (\# a^N)^{|I|}$. For each $m \in \{0, \ldots, |I|\}$, we apply the proposition to the factorization $s = u_k v_k w_k$ with $u_k = (a^N \#)^k$, $v_k = a^N$ and $w_k = (\# a^N)^{|I|-k}$ to get a 1-split $s = u'_k v'_k w'_k$ according to φ where u_k is a prefix of u'_k and w_k is a suffix of w'_k. Let $p_k = |v_k| \neq 0$ and $q_k = N - |v_k|$.

For $f' \in \mathcal{F}$ (the finite set of functions introduced above), we then define

$$\widetilde{f'} : (n_0, \dots, n_{|I|}) \in \mathbb{N}^{|I|+1} \mapsto f'\left(a^{n_0p_0+q_0}\# \dots \# a^{n_{|I|}p_{|I|}+q_{|I|}}\right)$$

Thanks to Proposition 3.3.22 and to the 1-split conditions that we made sure to get previously, we see that for each letter c in the codomain of f' (either $\{a, \#\}$ or I), the map $|\widetilde{f'}|_c : \mathbb{N}^{|I|+1} \to \mathbb{N}$ is monotone for the product partial order. Since $f = \mathbf{map}(a \mapsto a^{n\times n}) = \mathrm{CbS}(g, (h_i)_{i\in I})$,

$$\forall x \in \mathbb{N}^{|I|+1},\ \sum_{i\in I} |\widetilde{g}(x)|_i \cdot |\widetilde{h_i}(x)|_\# = |\widetilde{f}(x)|_\# = |I|$$

$$\text{where} \qquad \widetilde{f} : (n_0, \dots, n_{|I|}) \in \mathbb{N}^{|I|+1} \mapsto f\left(a^{n_0p_0+q_0}\# \dots \# a^{n_{|I|}p_{|I|}+q_{|I|}}\right)$$
$$= a^{(n_0p_0+q_0)^2}\# \dots \# a^{(n_{|I|}p_{|I|}+q_{|I|})^2}$$

Since $|\widetilde{g}|_i$ and $|\widetilde{h_i}|_\#$ are monotone for all $i \in I$, and $\mathbb{N}^{|I|+1}$ admits a minimum $(0, \dots, 0)$, the fact that the above sum is constant means that, for each $i \in I$,

- either one of $|\widetilde{g}|_i$ and $|\widetilde{h_i}|_\#$ is constant equal to 0,
- or both are non-zero constant.

Let $J^\# \subseteq I$ be the set of indices that fit the second case. We claim that for $i \in J^\#$, the constant value taken by $|\widetilde{h_i}|_\#$ must be 1. If this were not the case, then for all $n \in \mathbb{N}$, there would be a substring of the form $\#a \dots a\#$ in $|\widetilde{h_i}(n, \dots, n)|_\#$, and since $f = \mathrm{CbS}(g, (h_i)_{i\in I})$ and $|\widetilde{g}(n, \dots, n)|_i \neq 0$, it would also be a substring of $\widetilde{f}(n, \dots, n)$ with length at most $|\widetilde{h_i}(n, \dots, n)|_\# = O(n)$ (since h_i is regular). This is impossible: for $k \in \{0, \dots, |I|\}$, the k-th substring of this form in $\widetilde{f}(n, \dots, n)$ has length $(np_k + q_k)^2 + 2 = \Theta(n^2)$.

Combining this with the above equation for $|\widetilde{f}|_\#$, we see that $|\widetilde{g}|_{J^\#}$ is the constant function equal to $|I|$. Let us abbreviate $\rho_k = \rho_{J^\#,k} \in \mathcal{F}$ (recall that we defined it at the beginning of this proof) for $k \in \{0, \dots, |I|\}$; then

$$\forall x \in \mathbb{N}^{|I|+1},\ \exists! \iota_1(x), \dots, \iota_{|I|}(x) \in J^\# : \widetilde{g}(x) = \widetilde{\rho_0}(x)\iota_1(x)\widetilde{\rho_1}(x) \dots \iota_{|I|}(x)\widetilde{\rho_{|I|}}(x)$$

Using this, we define $h'_k(x) = \widetilde{h}_{\iota_k(x)}(x)$ for $x \in \mathbb{N}^{|I|+1}$ and $k \in \{1, \dots, |I|\}$, plus two edge cases $h'_0 : x \mapsto \varepsilon$ and $h'_{|I|+1} : x \mapsto \varepsilon$.

Let $k \in \{0, \dots, |I|\}$. Write $\vec{e}_k = (0, \dots, 0, 1, 0, \dots, 0) \in \mathbb{N}^{|I|+1}$ for the k-th vector of the canonical basis of $\mathbb{Q}^{|I|+1}$. By looking again at the k-th substring of the form $\#a \dots a\#$ in $\widetilde{f}(x)$, with $x = n\vec{e}_k$ here, we get

$$\forall n \in \mathbb{N},\ (np_k + q_k)^2 + 2 \leq |h'_k(n\vec{e}_k)| + \sum_{i\in I} |\widetilde{\rho_k}(n\vec{e}_k)|_i \cdot |\widetilde{h_i}(n\vec{e}_k)| + |h'_{k+1}(n\vec{e}_k)|$$

Note that all the lengths involved in the right-hand side above are linearly bounded in n because of the regularity of the functions involved. So there must exist $i_k \in I$ such that both $|\widetilde{\rho_k}(n\vec{e}_k)|_{i_k}$ and $|\widetilde{h_{i_k}}(n\vec{e}_k)|$ are unbounded: otherwise, the whole RHS would be $O(n)$, contradicting the $\Omega(n^2)$ lower bound induced by the above inequality.

We thus get a finite sequence of indices $i_0, \dots, i_{|I|} \in I$. By the pigeonhole principle, there must exist $k, l \in \{0, \dots, |I|\}$ such that $k \neq l$ and $i_k = i_l$; we call i this common value. Let $m \in \mathbb{N}$ be such that $|\widetilde{\rho_k}(m\vec{e}_k)|_i \geq 1$. By monotonicity (since $\rho_k, h_i \in \mathcal{F}$):

- $|\widetilde{\rho_k}(m\vec{e}_k + n\vec{e}_l)|_i \geq 1$ for all $n \in \mathbb{N}$;
- $|\widetilde{h_i}(m\vec{e}_k + n\vec{e}_l)|$ is unbounded when $n \to +\infty$.

The product of those two quantities is a lower bound for the length of the k-th substring of the form $\#a \dots a\#$ in $\widetilde{f}(m\vec{e}_k + n\vec{e}_l)$, which contradicts the fact that this length does not depend on n (it is equal to $(mp_k + q_k)^2 + 2$).

CHAPTER 4

Streaming transducers meet categorical semantics

This chapter is devoted to the material on categorical automata theory for which we gave a high-level overview in Section 1.2.3 of the introduction. Since it is the first place where category theory appears, it also contains some preliminaries on the topic (§4.1).

We start by introducing in Section 4.2 a general notion of automaton over strings parameterized by a structure $\mathfrak{C}$ that we call a *(string) streaming setting* and whose main component is a category $\mathcal{C}$. From the point of view of expressiveness, $\mathfrak{C}$ can be thought of as a gadget delimiting a class of transition monoids which may be used for computations on top of finite structure of what we call a $\mathfrak{C}$-*SST* – a name meant to[1] emphasize an intuition of these categorical automata as generalizations of streaming string transducers ($\mathcal{C}$ should then be thought of as the collection of possible memory updates). We give a few examples of streaming settings, including in §4.2.1 the one that corresponds to usual copyless SSTs, which we name $\mathfrak{SR}$ (and we call $\mathcal{SR}$ its underlying category of string-valued registers).

Then, in Section 4.3, we study the free finite coproduct completion of categories $(-)_{\oplus}$, which readily extends to streaming settings. In particular, properties of $\mathfrak{SR}_{\oplus}$ are explored. Section 4.4 deals with the dual construction $(-)_{\&}$, the free product completion. A tight link between the expressiveness of $\mathfrak{C}_{\&}$-SSTs and *non-deterministic* $\mathfrak{C}$-SSTs is established. Section 4.5 then combines those results to study the composition $((-)_{\&})_{\oplus}$ of those two completions (which we describe as a direct construction $(-)_{\oplus\&}$ – the order $\oplus\&$ rather than $\&\oplus$ in the notation is deliberate). In particular, it is shown that the underlying category of $\mathfrak{SR}_{\oplus\&}$ is *monoidal closed*; as already said in the introduction, this last property will turn out in the next chapter to be crucial to prove the characterizations of string transduction classes in the $\lambda\ell^{\oplus\&}$-calculus.

The rest of this chapter deals with more technical material that our "implicit automata" results do not depend on, but which is interesting in itself from the standpoint of categorical automata theory. First, in Section 4.6 we give a kind of categorical justification of the definition of $\mathcal{SR}$, by claiming a universal property and proving the existence part of this claim. The latter is a convenient technical tool that we apply in Section 4.7 to establish a general theorem on preservation by precomposition by regular functions. This was one of the two main illustrations of the relevance of monoidal closure for automata explained in Chapter 1 (see §1.2.3); the other one, a determinization (or more precisely uniformization) result, is the topic of the last section of the present chapter (§4.8).

4.1. Categorical preliminaries

Our use of category theory, while absolutely essential, stays at a fairly elementary level. We assume familiarity with the notions of category, functor, natural transformation, (cartesian) product and coproduct (and their nullary cases, terminal and initial objects),

[1] In retrospect, it would have been wiser to speak of $\mathcal{C}$-automata and to make our definitions fully compatible with the previous work of Colcombet and Petrisan (cf. §1.1.9), which we were unaware of when we started working on this; see Remark 4.2.3 for the slight differences. Since this is merely a cosmetic concern, I have chosen here not to perform this harmonization in the limited time available to me to write this manuscript, but a future journal version of this chapter will probably incorporate this modification.

but not much more than that; the remaining categorical prerequisites are summed up here for convenience. The reader familiar with monoidal closed categories can safely skip directly to Section 4.1.4. We also wish to stress that while we motivate some definitions by referring to the $\lambda\ell^{\oplus\&}$-calculus, which will only be formally introduced in Chapter 5, all the technical content can be followed while ignoring those justifications.

4.1.1. **Notations on categories.** Given a category $\mathcal{C}$, we write $\mathsf{Obj}(\mathcal{C})$ for its class of objects and $\mathsf{Hom}_{\mathcal{C}}(A, B)$ for the set of arrows (or morphisms) from A to B (for $A, B \in \mathsf{Obj}(C)$). The composition of two morphisms $f \in \mathsf{Hom}_{\mathcal{C}}(A, B)$ and $g \in \mathsf{Hom}_{\mathcal{C}}(B, C)$ is denoted by $g \circ f$. Following the traditional notations of linear logic, products and coproducts will be customarily written using '&' and '$\oplus$' respectively – except in the category of sets where we use the notations '$\times$' and '$+$' as usual – and we reserve $\top$ for the terminal object. We sometimes use basic combinators such as $\langle -\rangle/[-]$ for pairing/copairing and π_i/in_i for projections/coprojections.

Finally, if we are given a binary operation $\square$ over the objects of a category, we freely use the corresponding "I-ary" operation, with a notation of the form $\square_{i\in I} A_i$, over families indexed by a finite set I. Concretely speaking, this depends on a fixed total order over $I = \{i_1 < \ldots < i_{|I|}\}$ to unfold as $A_{i_1} \square (A_{i_2} \square (\ldots \square A_{i_{|I|}}) \ldots)$ – for convenience, the reader may consider that a choice of such an order for every finite set is fixed once and for all for the rest of the manuscript. In practice, the particular order does not matter since we will deal with operations $\square \in \{\oplus, \&, \otimes, \ldots\}$ that are symmetric in a suitable sense. Those operations also have units (i.e. identity elements), giving a canonical meaning to $\bigoplus_{i\in\varnothing} A_i$, $\mathop{\&}_{i\in\varnothing} A_i$, etc.

Finally, as is usual when dealing with categories, we sometimes allow ourselves to implicitly use the axiom of choice for classes to pick objects determined by their universal properties to build functors (for instance, given an object A in a category $\mathcal{C}$ with cartesian products, we shall speak of the functor $- \,\&\, A$ without first mentioning that a choice of cartesian products $X \,\&\, A$ exists for for every X in $\mathcal{C}$). This is merely for convenience; the reader may check that in all of our concrete examples of interest, canonical choices can be made without appealing to choice.

4.1.2. **Monoidal categories, symmetry and functors.** The idea of categorical semantics is to interpret the types of a programming language – in our case, the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus – as objects, and the programs (terms) as morphisms. (A formal statement tailored to our purposes will be given later in Lemma 5.2.6.) In this perspective, the additive conjunction '&' of the $\lambda\ell^{\oplus\&}$-calculus is interpreted as a categorical *cartesian product*, while the additive disjunction '$\oplus$' corresponds to a *coproduct*; this justifies our use of the notations $\&/\oplus$ for products/coproducts. We now define monoidal products, which are meant to interpret the multiplicative conjunction '$\otimes$'.

Definition 4.1.1 ([Mel09, Sections 4.1 to 4.4])**.** Let $\mathcal{C}$ be a category. A *monoidal product* $\otimes$ over $\mathcal{C}$ is given by the combination of

- a bifunctor $- \otimes - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$
- a distinguished object $\mathbf{I}$
- natural isomorphisms
  - $\lambda_A : \mathbf{I} \otimes A \to A$ (*left unitor*),
  - $\rho_A : A \otimes \mathbf{I} \to A$ (*right unitor*),
  - and $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$ (*associator*)

  such that the diagrams in Figure 4.1.1 commute.

Such a monoidal product is said to be *symmetric* if it comes with a natural isomorphism (called the *symmetry*) $\gamma_{A,B} : A \otimes B \to B \otimes A$ subject to the conditions of Figure 4.1.2.

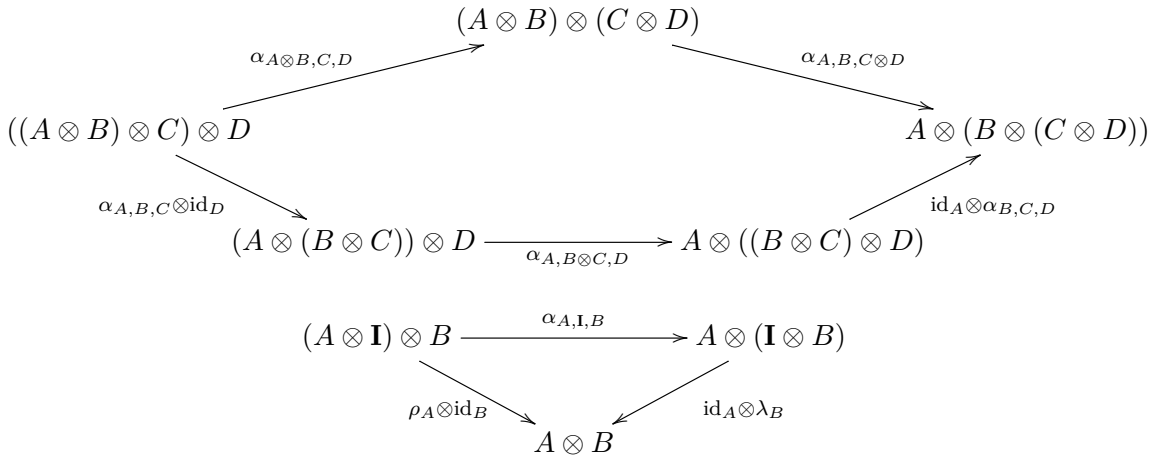


FIGURE 4.1.1. Coherence conditions for monoidal categories

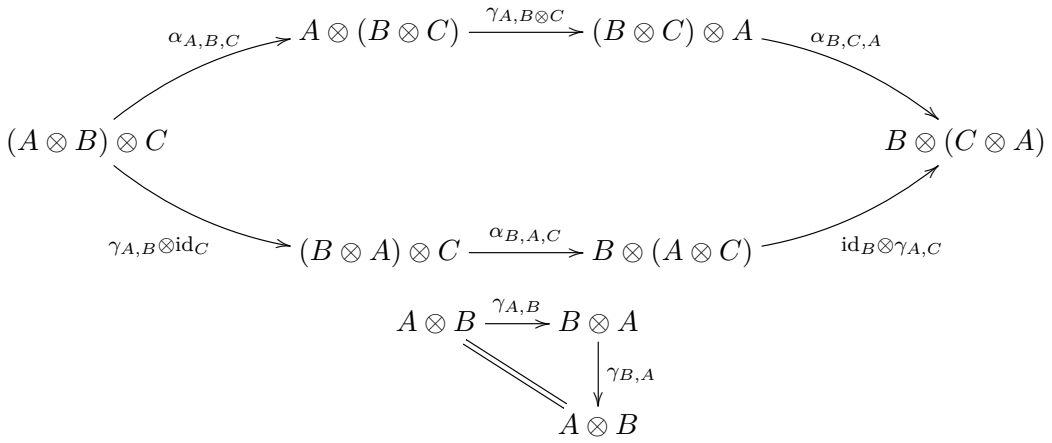


FIGURE 4.1.2. Coherence conditions for symmetries

In the sequel, we use the name *(symmetric) monoidal category* for a category $\mathcal{C}$ that comes equipped with a (symmetric) monoidal structure $\otimes, \mathbf{I}, \ldots$ and write it $(\mathcal{C}, \otimes, \mathbf{I})$ for short.[2] Of course, if a category $\mathcal{C}$ has binary products & and a terminal object $\top$, then $(\mathcal{C}, \&, \top)$ is a symmetric monoidal category, and similarly for coproducts and intial objects.

We shall sometimes need to refer to morphisms between monoidal categories, which are essentially functors together with natural transformations witnessing that the monoidal structure is preserved.

Definition 4.1.2 ([Mel09, Section 5.1])**.** Let $(\mathcal{C}, \otimes, \mathbf{I})$ and $(\mathcal{D}, \widehat{\otimes}, \widehat{\mathbf{I}})$ be two monoidal categories. A *lax monoidal functor* is given by a functor $F : \mathcal{C} \to \mathcal{D}$ together with natural transformations

$$m_0 : \widehat{\mathbf{I}} \to F(\mathbf{I}) \qquad\qquad m_{A,B} : F(A) \mathbin{\widehat{\otimes}} F(B) \to F(A \otimes B)$$

making the following diagrams commute.

[2] This is slightly abusive, as λ, ρ, α and γ are also part of the structure and not uniquely determined from the triple $(\mathcal{C}, \otimes, \mathbf{I})$.

$$\begin{array}{ccc}
(F(A) \mathbin{\widehat{\otimes}} F(B)) \mathbin{\widehat{\otimes}} F(C) & \xrightarrow{\alpha_{F(A),F(B),F(C)}} & F(A) \mathbin{\widehat{\otimes}} (F(B) \mathbin{\widehat{\otimes}} F(C)) \\
\downarrow{\scriptstyle m_{A,B}\widehat{\otimes}\mathrm{id}_{F(C)}} & & \downarrow{\scriptstyle \mathrm{id}_{F(A)}\widehat{\otimes}m_{B,C}} \\
F(A \otimes B) \mathbin{\widehat{\otimes}} F(C) & & F(A) \mathbin{\widehat{\otimes}} F(B \otimes C) \\
\downarrow{\scriptstyle m_{A\otimes B,C}} & & \downarrow{\scriptstyle m_{A,B\otimes C}} \\
F((A \otimes B) \otimes C) & \xrightarrow{F(\alpha_{A,B,C})} & F(A \otimes (B \otimes C))
\end{array}$$

$$\begin{array}{ccc}
F(A) \mathbin{\widehat{\otimes}} \widehat{\mathbf{I}} & \xrightarrow{\rho_{F(A)}} & F(A) \\
\downarrow{\scriptstyle \mathrm{id}_{F(A)}\widehat{\otimes}m_0} & & \uparrow{\scriptstyle F(\rho_A)} \\
F(A) \mathbin{\widehat{\otimes}} F(\mathbf{I}) & \xrightarrow{m_{A,\mathbf{I}}} & F(A \otimes \mathbf{I})
\end{array}
\qquad
\begin{array}{ccc}
\widehat{\mathbf{I}} \mathbin{\widehat{\otimes}} F(A) & \xrightarrow{\lambda_{F(A)}} & F(A) \\
\downarrow{\scriptstyle m_0\widehat{\otimes}\mathrm{id}_{F(A)}} & & \uparrow{\scriptstyle F(\lambda_A)} \\
F(\mathbf{I}) \mathbin{\widehat{\otimes}} F(A) & \xrightarrow{m_{\mathbf{I},A}} & F(\mathbf{I} \otimes A)
\end{array}$$

A lax monoidal functor is called *strong monoidal* if the natural transformations m_0 and $m_{A,B}$ are isomorphisms.

Let us note that while every concrete instance of monoidal functor in the paper is also going to be a *symmetric* monoidal functor (i.e., satisfy additional coherence diagrams involving γ), we do not make use of that fact.

4.1.3. **Function spaces and monoidal closure.** Our next definition serves to interpret the linear function arrow '$\multimap$'. (Since we will only need a semantics for the *purely linear* fragment of the $\lambda\ell^{\oplus\&}$-calculus, we do not discuss the non-linear arrow '$\to$' here.)

Definition 4.1.3 ([Mel09, Sections 4.5 to 4.7])**.** Let $(\mathcal{C}, \otimes, \mathbf{I})$ be a (symmetric) monoidal category and $A, B \in \mathsf{Obj}(\mathcal{C})$. An *internal homset* from A to B is an object $A \multimap B \in \mathsf{Obj}(\mathcal{C})$ with a prescribed arrow $\mathrm{ev}_{A,B} : (A \multimap B) \otimes A \to B$ (the *evaluation map*) such that, for every other arrow $f : C \otimes A \to B$, there is a unique map $\Lambda(f)$ (called the *currying of* f) making the following diagram commute:

$$\begin{array}{ccc}
(A \multimap B) \otimes A & \xrightarrow{\mathrm{ev}_{A,B}} & B \\
{\scriptstyle \Lambda(f)\otimes\mathrm{id}}\uparrow & \nearrow{\scriptstyle f} & \\
C \otimes A & &
\end{array}$$

When there exists an internal homset for every pair objects in $\mathcal{C}$, we say that $(\mathcal{C}, \otimes, \mathbf{I})$ is a *(symmetric) monoidal closed* category.

As for (co)products, internal homsets are determined up to unique isomorphism, so we may talk somewhat loosely about *the* internal homset later on. While we work with the universal property given in Definition 4.1.3 when the construction of an internal homset involves a bit of combinatorics, we will also sometimes use the following characterization.

Proposition 4.1.4. *The object $A \multimap B$ is an internal homset for $A, B \in \mathsf{Obj}(\mathcal{C})$ if and only if there is a family of isomorphisms*

$$\mathsf{Hom}_{\mathcal{C}}(C \otimes A,\, B) \quad \cong \quad \mathsf{Hom}_{\mathcal{C}}(C,\, A \multimap B)$$

which is natural *in the parameter C varying contravariantly over $\mathcal{C}$ (in other words, if $\mathsf{Hom}_{\mathcal{C}}(- \otimes A,\, B)$ and $\mathsf{Hom}_{\mathcal{C}}(-,\, A \multimap B)$ are naturally isomorphic as functors $\mathcal{C}^{\mathrm{op}} \to \mathsf{Set}$).*

Proof. This is an instance of [Mac98, Chapter III, Section 2, Proposition 1]. □

4.1.4. **Affineness and quasi-affineness.** Given a monoidal product $\otimes$, morphisms from A to $A \otimes A$ need not exist in general; this accounts for the linearity constraints in $\lambda\ell^{\oplus\&}$. But monoidal categories do not incorporate the ability of register transitions in SSTs to discard the content of a register, a behavior more aligned with the *affine* λ-calculi. This notion thus plays a role in our development, so we discuss its incarnation in categorical semantics.

Definition 4.1.5. A (symmetric) monoidal category $(\mathcal{C}, \otimes, \mathbf{I})$ is called *affine* if $\mathbf{I}$ is a terminal object of $\mathcal{C}$. (Such categories are also sometimes called *semi-cartesian* [nLa20].)

Most symmetric monoidal categories are not affine. However, there is a generic way of building an affine monoidal category from a monoidal category. Recall that if $\mathcal{C}$ is a category and X is an object of $\mathcal{C}$, one may consider the *slice category* $\mathcal{C}/_X$

- whose objects are morphisms $A \to X$ ($A \in \mathsf{Obj}(\mathcal{C})$),
- and such that $\mathsf{Hom}_{\mathcal{C}/_X}(f : A \to X, g : B \to X) = \{h \in \mathsf{Hom}_{\mathcal{C}}(A, B) \mid g \circ h = f\}$.

If $\mathcal{C}$ has a monoidal structure $(\otimes, \mathbf{I})$, this structure can be lifted to $\mathcal{C}/_{\mathbf{I}}$ by taking the identity $\mathbf{I} \to \mathbf{I}$ as the unit and

$$\left(A \xrightarrow{f} \mathbf{I}\right) \otimes \left(B \xrightarrow{g} \mathbf{I}\right) = \left(A \otimes B \xrightarrow{f \otimes g} \mathbf{I} \otimes \mathbf{I} \xrightarrow{\lambda_{\mathbf{I}} = \rho_{\mathbf{I}}} \mathbf{I}\right)$$

as the monoidal product. This gives rise to an affine monoidal structure over $\mathcal{C}/_{\mathbf{I}}$, and a strong monoidal structure for the forgetful functor $\mathsf{dom} : \mathcal{C}/_{\mathbf{I}} \to \mathcal{C}$.

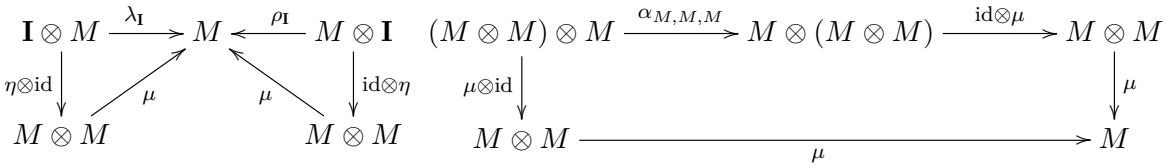
In the converse direction, one can sometimes turn an object A from $\mathcal{C}$ into one of $\mathcal{C}/_{\mathbf{I}}$. This is the case when A admits a cartesian product with $\mathbf{I}$, which may be written $A \,\&\, \mathbf{I}$ (note that if $\mathcal{C}$ is affine, A itself is such a cartesian product). We are then led to consider the projection $\pi_2 : A \,\&\, \mathbf{I} \to \mathbf{I}$ as an object of the slice category.

Definition 4.1.6. A (symmetric) monoidal category $(\mathcal{C}, \otimes, \mathbf{I})$ is called *quasi-affine* if every $A \in \mathsf{Obj}(\mathcal{C})$ has a cartesian product $A \,\&\, \mathbf{I}$ with the monoidal unit.

Remark 4.1.7. We have a map $A \in \mathsf{Obj}(\mathcal{C}) \mapsto \left(A \,\&\, \mathbf{I} \xrightarrow{\pi_2} \mathbf{I}\right) \in \mathsf{Obj}\left(\mathcal{C}/_{\mathbf{I}}\right)$ in any quasi-affine category, according to the above discussion. It turns out that it extends to a functor J which embeds $\mathcal{C}$ into this affine slice category; moreover, J is *right adjoint* to the forgetful functor dom. The interested reader may even check (although we will not make use of this) that the existence of a right adjoint to dom is *equivalent* to quasi-affineness.

4.1.5. **Monoids.** Since we are interested in string transductions, the free monoids Σ^* are going to make an appearance. Let us thus conclude this section by recalling the notion of monoid *internal to a monoidal category*.

Definition 4.1.8 ([Mel09, Section 6.1])**.** Given a monoidal category $(\mathcal{C}, \otimes, \mathbf{I})$, an *internal monoid* (or a *monoid object*) is a triple (M, μ, η) where $M \in \mathsf{Obj}(\mathcal{C})$ and $\mu : M \otimes M \to M$, $\eta : \mathbf{I} \to M$ are morphisms making the following unitality and associativity diagrams commute



A useful example of this notion is the "internalization" of the monoid of endomorphisms of A when A is part of a monoidal *closed* category.

Proposition 4.1.9. *Let $(\mathcal{C}, \otimes, \mathbf{I})$ be a monoidal category. Any internal homset $A \multimap A$ (with $A \in \mathsf{Obj}(C)$) that exists in $\mathcal{C}$ has an internal monoid structure $(A \multimap A,\ \eta,\ \mu)$ such that*

$$\eta = \Lambda'(\mathrm{id}_A) \qquad \mu \circ (\Lambda'(f) \otimes \Lambda'(g)) \circ \lambda_{\mathbf{I}} = \Lambda'(f \circ g) \quad \text{for } f, g \in \mathsf{Hom}_{\mathcal{C}}(A, A)$$

where $\Lambda' : \mathsf{Hom}_{\mathcal{C}}(A, A) \xrightarrow{\sim} \mathsf{Hom}_{\mathcal{C}}(\mathbf{I},\ A \multimap A)$ is defined as $\Lambda' : h \mapsto \Lambda(h \circ \lambda_A)$ from the curryfication Λ and the left unitor λ_A.

Proof sketch. One can define the monoid multiplication

$$\mu : (A \multimap A) \otimes (A \multimap A) \to (A \multimap A)$$

as the curryfication $\mu = \Lambda(\mathsf{app2})$ of the morphism $\mathsf{app2}$ built by composing the sequence

$$((A \multimap A) \otimes (A \multimap A)) \otimes A \xrightarrow{\alpha} (A \multimap A) \otimes ((A \multimap A) \otimes A) \xrightarrow{\mathrm{id} \otimes \mathrm{ev}} (A \multimap A) \otimes A \xrightarrow{\mathrm{ev}} A$$

and check that it satisfies the coherence diagrams for internal monoids (that also involve the unit η defined in the proposition statement) and the equation relating μ to Λ'. □

Let us conclude our categorical preliminaries on the following.

Proposition 4.1.10. *Let $(\mathcal{C}, \otimes, \mathbf{I})$ be a monoidal category and let $M \,\&\, \mathbf{I} \in \mathsf{Obj}(C)$ be a cartesian product of some $M \in \mathsf{Obj}(C)$ with the monoidal unit $\mathbf{I}$. Suppose that (M, μ, η) is a monoid object. Then $M \,\&\, \mathbf{I}$ has an internal monoid structure defined by*

$$\langle \mu \circ (\pi_1 \otimes \pi_1),\ \lambda_{\mathbf{I}} \circ (\pi_2 \otimes \pi_2) \rangle \ :\ (M \,\&\, \mathbf{I}) \otimes (M \,\&\, \mathbf{I}) \to M \,\&\, \mathbf{I} \qquad \langle \eta, \mathrm{id}_{\mathbf{I}} \rangle \ :\ \mathbf{I} \to M \,\&\, \mathbf{I}$$

where $\pi_1 : M \,\&\, \mathbf{I} \to M$ and $\pi_2 : M \,\&\, \mathbf{I} \to \mathbf{I}$ are the projections and $\langle -, - \rangle$ is the pairing given by the universal property of the cartesian product.

Furthermore, this makes $\left(M \,\&\, \mathbf{I} \xrightarrow{\pi_2} \mathbf{I}\right) \in \mathsf{Obj}\left(\mathcal{C}/_{\mathbf{I}}\right)$ into a monoid object of $\mathcal{C}/_{\mathbf{I}}$.

The routine verification of the required commutations of diagrams is left to the reader.

Remark 4.1.11. Our applications of this proposition will take place in *quasi-affine* monoidal categories. For those, it admits a more conceptual proof: the right adjoint $J : \mathcal{C}/_{\mathbf{I}} \to \mathcal{C}$ to the forgetful functor dom (cf. Remark 4.1.7) is *lax monoidal*, and therefore so is $\mathsf{dom} \circ J$ which maps A to $A \,\&\, \mathbf{I}$ on objects; furthemore, the image of a monoid object by a lax monoidal functor is itself a monoid object in a canonical way [Mel09, Section 6.2].

4.2. A categorical framework for automata: streaming settings

We now introduce string streaming settings, which can be understood for our purposes as a sort of memory framework for transducers iterating performing a single left-to-right pass over a word. (As we argued in the introduction (§1.1.9), they are in fact more general than that since they can express two-way automata, but this is beyond our scope here.) This is the abstract notion that will allow us to generalize streaming string transducers:

Definition 4.2.1. Let X be a set. A *string streaming setting* with output X is a tuple $\mathfrak{C} = (\mathcal{C}, \Top, \Bot, (\!|-|\!))$ where

- $\mathcal{C}$ is a category
- $\Top$ and $\Bot$ are arbitrary objects of $\mathcal{C}$
- $(\!|-|\!)$ is a set-theoretic map $\mathsf{Hom}_{\mathcal{C}}(\Top, \Bot) \to X$

Since the properties of the underlying category of a streaming setting will turn out to be the most crucial thing in the sequel, we shall abusively apply adjectives befitting categories (such as "affine symmetric monoidal") to streaming settings in the sequel.

The notion of streaming setting is a convenient tool motivated by our subsequent development rather than our primary object of study. A closely related framework in which some of our abstract results can be formulated is defined in [CP20] (see Remark 4.2.3).

For the rest of this section, we will refer to string streaming setting simply as streaming settings; we also fix two alphabets Σ and Γ for the rest of this section.

Definition 4.2.2. Let $\mathfrak{C} = (\mathcal{C}, \top, \bot\!\!\!\bot, (\!|-|\!))$ be a streaming setting with output X. A $\mathfrak{C}$-SST with input alphabet Σ and output X is a tuple $(Q, q_0, R, \delta, i, o)$ where

- Q is a finite set of states and $q_0 \in Q$
- R is an object of $\mathcal{C}$
- δ is a function $\Sigma \times Q \to Q \times \mathsf{Hom}_{\mathcal{C}}(R, R)$
- $i \in \mathsf{Hom}_{\mathcal{C}}(\top, R)$ is an initialization morphism
- $(o_q)_{q \in Q} \in \mathsf{Hom}_{\mathcal{C}}(R, \bot\!\!\!\bot)^Q$ is a family of output morphisms – alternatively, we will sometimes consider it as a map $o : Q \to \mathsf{Hom}_{\mathcal{C}}(R, \bot\!\!\!\bot)$.

We write $\mathcal{T} : \Sigma^* \to_{\mathfrak{C}\text{-SST}} X$ to mean that $\mathcal{T}$ is a $\mathfrak{C}$-SST with input alphabet Σ and output X (the latter depends only on $\mathfrak{C}$).

The corresponding function $[\![\mathcal{T}]\!] : \Sigma^* \to X$ is then computed as for standard SSTs (cf. Definition 2.3.3): an input word w generates a sequence of states $q_0, \dots, q_{|w|} \in Q$ and a sequence of morphisms $f_i : R \to R$ in $\mathcal{C}$, and the output is then $(\!|o_{q_{|w|}} \circ f_{|w|} \circ \cdots \circ f_1 \circ i|\!) \in X$.

An important class of $\mathfrak{C}$-SSTs are those for which the set of states Q is a singleton, significantly simplifying the above data. They are called *single-state* $\mathfrak{C}$*-SSTs*.

Remark 4.2.3. Single-state $\mathfrak{C}$-SSTs are very close to the $\mathcal{C}$-automata over words defined by Colcombet and Petrişan [CP20, Section 3], or more precisely $(\mathcal{C}, \top, \bot\!\!\!\bot)$-automata with our notations. The main difference is that the latter's output would juste be an element of $\mathsf{Hom}_{\mathcal{C}}(\top, \bot\!\!\!\bot)$: there is no post-processing $(\!|-|\!)$ to produce an output.

As for the addition of finite states, ultimately, it does not increase the framework's expressive power: we shall see in Remark 4.3.13 that $\mathfrak{C}$-SSTs are equivalent to single-state SSTs over a modified category. We chose to incorporate states into our definition for convenience.

Example 4.2.4. Let $\mathfrak{Set}_X = (\mathsf{Set}, \{\bullet\}, X, (\!|-|\!))$ where $(\!|-|\!)$ is the canonical isomorphism between $\mathsf{Hom}_{\mathsf{Set}}(\{\bullet\}, X) = X^{\{\bullet\}}$ and X. Then *any* function $\Sigma^* \to X$ can be "computed" by a single-state $\mathfrak{Set}_X$-SST by taking $R = \Sigma^*$.

Example 4.2.5. Let $\mathfrak{Finset}_2 = (\mathsf{FinSet}, \{\bullet\}, \{0,1\}, (\!|-|\!))$ with $(\!|-|\!)$ the canonical isomorphism $\mathsf{Hom}_{\mathsf{FinSet}}(\{\bullet\}, \{0,1\}) \cong \{0,1\}$. Single-state $\mathfrak{Finset}_2$-SST are essentially the usual notion of deterministic finite automata[3]. Therefore, the functions they compute are none other than the indicator functions of regular languages.

Example 4.2.6. Consider the category $\mathcal{POL}_{\mathbb{Q}}$ whose objects are natural numbers, whose morphisms are tuples of multivariate polynomials over $\mathbb{Q}$ with matching arities (so that $\mathsf{Hom}_{\mathcal{POL}_{\mathbb{Q}}}(n,k) = (\mathbb{Q}[X_1, \dots, X_n])^k$) and where composition is lifted from the composition of polynomials in the usual way, making $\mathcal{POL}_{\mathbb{Q}}$ into a category with (strict) cartesian products. Then, taking $\mathfrak{Pol}_{\mathbb{Q}} = (\mathcal{POL}_{\mathbb{Q}}, 0, 1, (\!|-|\!))$ where $(\!|-|\!)$ is the isomorphism identifying $\mathbb{Q}$ and polynomials without variables ($n = 0$), we can recover the definition of *polynomial automata* from [Ben+17] as single-state $\mathfrak{Pol}_{\mathbb{Q}}$-SSTs.

Given two streaming settings $\mathfrak{C}$ and $\mathfrak{D}$ with a common output set X, $\mathfrak{C}$-SSTs are said to *subsume* $\mathfrak{D}$-SSTs if for every $\mathfrak{D}$-SST $\mathcal{T}$ there is a $\mathfrak{C}$-SST $\mathcal{T}'$ with $[\![\mathcal{T}]\!] = [\![\mathcal{T}']\!]$. We say that $\mathfrak{C}$-SSTs and $\mathfrak{D}$-SSTs are *equivalent* if both classes subsume one another. There is also a straightforward notion of morphism of streaming settings with common output.

[3] Actually, *complete* DFA, i.e. DFA with total transition functions.

Definition 4.2.7. Let $\mathfrak{C} = (\mathcal{C}, \top_{\mathfrak{C}}, \bot\!\!\!\bot_{\mathfrak{C}}, (\!|-|\!)_{\mathfrak{C}})$ and $\mathfrak{D} = (\mathcal{D}, \top_{\mathfrak{D}}, \bot\!\!\!\bot_{\mathfrak{D}}, (\!|-|\!)_{\mathfrak{D}})$ be streaming settings with the same output set X. A morphism of streaming settings is given by a functor $F : \mathcal{C} \to \mathcal{D}$ and $\mathcal{D}$-arrows $i : \top_{\mathfrak{D}} \to F(\top_{\mathfrak{C}})$ and $o : F(\bot\!\!\!\bot_{\mathfrak{C}}) \to \bot\!\!\!\bot_{\mathfrak{D}}$ such that

$$\forall f \in \mathsf{Hom}_{\mathcal{C}}(\top_{\mathfrak{C}}, \bot\!\!\!\bot_{\mathfrak{C}}), \ (\!|o \circ F(f) \circ i|\!)_{\mathfrak{D}} = (\!|f|\!)_{\mathfrak{C}}$$

This notion is useful to compare the expressiveness of classes of generalized SSTs because of the following lemma.

Lemma 4.2.8. *If there is a morphism of streaming settings $\mathfrak{C} \to \mathfrak{D}$, then $\mathfrak{D}$-SSTs subsume $\mathfrak{C}$-SSTs and single-state $\mathfrak{D}$-SSTs subsume single-state $\mathfrak{C}$-SSTs.*

Proof sketch. Given a $\mathfrak{C}$-SST $(Q, q_0, R, \delta, i, (o_q)_{q\in Q})$ (where we use the notations of Definition 4.2.2) and a morphism $(F : \mathcal{C} \to \mathcal{D},\ i' : \top_{\mathfrak{D}} \to F(\top_{\mathfrak{C}}),\ o' : F(\bot\!\!\!\bot_{\mathfrak{C}}) \to \bot\!\!\!\bot_{\mathfrak{D}})$ of streaming settings, one builds a $\mathfrak{D}$-SST that computes the same function as follows. The set of states and initial state are unchanged (so our proof applies both to the stateful and the single-state case). The memory object becomes $F(R)$, and the $\mathsf{Hom}_{\mathcal{C}}(R, R)$ component of the transition function δ is passed through the functor F to yield a $\mathfrak{D}$-endomorphism of $F(R)$. The new initialization morphism is $F(i) \circ i'$ and the new output morphisms are $(o' \circ F(o_q))_{q\in Q}$. □

Remark 4.2.9. For any streaming setting $\mathfrak{C}$, the functor $\mathsf{Hom}_{\mathcal{C}}(\top, -)$ is a morphism of streaming settings $\mathfrak{C} \to \mathfrak{Set}$ with $i = \mathrm{id}$ and $o = (\!|-|\!)_{\mathfrak{C}}$.

In the sequel, we will omit giving the morphisms $i : \top_{\mathcal{D}} \to F(\top_{\mathcal{C}})$ and $o : F(\bot\!\!\!\bot_{\mathcal{C}}) \to \bot\!\!\!\bot_{\mathcal{D}}$ most of the time, as they will be isomorphisms deducible from the context.

4.2.1. **The category $\mathcal{SR}(\Gamma)$ of Γ-register transitions.** We shall now formulate copyless streaming string transducers in our framework, using a category whose morphisms we call *register transitions*. Those are a variant of copyless register assignments (cf. §2.3) where we use a coproduct rather than a union in the formalism. This choice would have been more cumbersome for the proofs of Chapters 2 and 3, but it is better if we want to build a category whose objects are *all* finite sets.

Definition 4.2.10. Fix a finite alphabet Γ and let R and S be finite sets.

A Γ-*register transition* from R to S is a function $t : S \to (\Gamma + R)^*$ satisfying the following *copylessness* condition: for every $r \in R$, there is at most one occurrence of $\mathsf{in}_2(r)$ among all the words $t(s)$ for $s \in S$. (Compare with Definitions 2.3.1 and 2.3.5.)

We write $[R \to_{\mathcal{SR}(\Gamma)} S]$ for the set of Γ-register transitions from R to S, or $[R \to_{\mathcal{SR}} S]$ when Γ is clear from context. For any $t \in [R \to_{\mathcal{SR}(\Gamma)} S]$, we define $t^\dagger : (\Sigma^*)^R \to (\Sigma^*)^S$ by transposing the definition of $(-)^\dagger$ in Definition 2.3.1 in the only sensible way.

To arrange register transitions into a category, we must be able to *compose* any two of them. Moreover, this composition should be compatible with the action of register transitions on tuples of strings, i.e. the latter should be *functorial*: for $t \in [R \to_{\mathcal{SR}(\Gamma)} S]$ and $t' \in [S \to_{\mathcal{SR}(\Gamma)} T]$, we expect that $(t' \circ_{\mathcal{SR}} t)^\dagger = t'^\dagger \circ t^\dagger$. In fact, we have already seen how to do this in Section 2.3.4 for the case $R = S$: composition then corresponds to the monoid multiplication of Definition 2.3.16, while functoriality is stated in Proposition 2.3.19. All this is extended below to an arbitrary choice of finite R and S.

Definition 4.2.11. Let $t \in [R \to_{\mathcal{SR}(\Gamma)} S]$ and $t' \in [S \to_{\mathcal{SR}(\Gamma)} T]$; recall that t and t' are defined as maps between sets $t : S \to (\Gamma + R)^*$ and $t' : T \to (\Gamma + S)^*$.

We define the *composition of register transitions* $t' \circ_{\mathcal{SR}} t : T \to (\Gamma + R)^*$ to be the set-theoretic composition $t^\star \circ t'$ where $t^\star : (\Gamma + S)^* \to (\Gamma + R)^*$ is the unique monoid morphism extending the copairing of in_1 and t (i.e. $(\mathsf{in}_1(c) \mapsto \mathsf{in}_1(c),\ \mathsf{in}_2(s) \mapsto t(s)) : \Gamma + S \to (\Gamma + R)^*$).

(The notation $(-)^\star$ is consistent with the one used throughout previous chapters, see for instance Definition 2.3.16.)

Proposition 4.2.12. *There is a* category $\mathcal{SR}(\Gamma)$ *(given a finite alphabet* Γ *which we will often omit in the notation) whose objects are finite sets of registers, whose morphisms are register transitions* – $\mathsf{Hom}_{\mathcal{SR}(\Gamma)}(R,S) = [R \to_{\mathcal{SR}(\Gamma)} S]$ – *and whose composition is given by the above definition. This means in particular that, with the above notations,* $t' \circ_{\mathcal{SR}(\Gamma)} t \in [R \to_{\mathcal{SR}(\Gamma)} T]$*, i.e. copylessness is preserved by composition. Furthermore:*

- *This category admits the empty set of registers as the terminal object:* $\top = \varnothing$*.*
- *The action of register transitions on tuples of strings gives rise to a functor* $(-)^\dagger : \mathcal{SR} \to \mathsf{Set}$*, with* $X^\dagger = (\Gamma^*)^X$ *on objects.*

This can be verified purely mechanically from the definitions; basically, nothing new happens compared to what already appeared in the literature on usual SSTs (again, we refer to §2.3.4). Let us now introduce the streaming setting for copyless SSTs.

Definition 4.2.13. We write $\mathfrak{SR}(\Gamma)$ for $(\mathcal{SR}(\Gamma), \top = \varnothing, \perp\!\!\!\perp = \{\bullet\}, (\!|-|\!))$ where the map $(\!|-|\!) : [\varnothing \to_{\mathcal{SR}(\Gamma)} \{\bullet\}] \to \Gamma^*$ is the canonical isomorphism $((\Gamma + \varnothing)^*)^{\{\bullet\}} \cong \Gamma^*$.

Fact 4.2.14. Standard copyless SSTs $\Sigma^* \to_{\mathrm{SST}} \Gamma^*$ are the same thing as $\mathfrak{SR}$-SSTs $\Sigma^* \to \Gamma^*$.

Remark 4.2.15. The functor $\mathsf{Hom}_{\mathcal{C}}(\top, -)$ mentioned in Remark 4.2.9 is, in the case $\mathcal{C} = \mathcal{SR}$, naturally isomorphic to $(-)^\dagger$. Therefore, the latter can be extended to a morphism $\mathfrak{SR} \to \mathfrak{Set}$ of string streaming settings.

Proposition 4.2.16. *The category* $\mathcal{SR}$ *can be endowed with a symmetric monoidal structure, where the monoidal product* $R \otimes S$ *is the disjoint union of register sets* $R + S$ *and the unit is the empty set of registers. Since the latter is also the terminal object of* $\mathcal{SR}$*, this defines an* affine *symmetric monoidal category.*

Note that given $t \in [R \to_{\mathcal{SR}(\Gamma)} S]$ and $t' \in [T \to_{\mathcal{SR}(\Gamma)} U]$, there is only one sensible way to define a set-theoretic map $t \otimes t' : U + S \to (\Gamma + (R + T))^*$. The above proposition means, among other things, that $t \otimes t' \in [R + T \to_{\mathcal{SR}(\Gamma)} S + U]$. Checking this, as well as the requisite coherence diagrams for monoidal categories, is left to the reader.

This wraps up our initial presentation of $\mathcal{SR}(\Gamma)$. The next few sections will study the categories obtained from it by applying various completions, but we will return to the "naked" $\mathcal{SR}(\Gamma)$ in Section 4.6 where its definition will be motivated by intrinsic category-theoretic considerations (in other words, we shall argue that the connection with transducers is not strictly necessary to justify its existence).

4.3. The free coproduct completion (or finite states)

We give here an elementary definition of the free finite coproduct completion $\mathcal{C}_\oplus$ of a category $\mathcal{C}$ and some of its basic properties. We have said in the introduction that it amounts to adding finite states, but the true way to "freely add states" would actually be to consider a certain full subcategory of $\mathcal{C}_\oplus$ (cf. Remark 4.3.13). However, we will see that the coproduct completion leads to equivalently expressive streaming settings in most cases, while being better behaved – for instance, our monoidal closure result (Theorem 1.2.4) depends on having all finite coproducts.

The $(-)_\oplus$ construction consists essentially in considering finite families of objects of $\mathcal{C}$ as "formal coproducts". (Alternatively, one could use finite lists as in e.g. [Gal20, Definition 3] to get an equivalent category.)

Definition 4.3.1. Let $\mathcal{C}$ be a category. The *free finite coproduct completion* $\mathcal{C}_\oplus$ is the category defined as follows.

- An object of $\mathcal{C}_\oplus$ is a pair $(U, (C_u)_{u\in U})$ consisting of a finite set U and a family of objects of $\mathcal{C}$ over U. We write those as formal sums $\bigoplus_{u\in U} C_u$ in the sequel.
- A morphism $\bigoplus_{u\in U} C_u \to \bigoplus_{v\in V} C_v$ is a U-indexed family of pairs $(v_u, g_u)_{u\in U}$ with $v_u \in V$ and $g_u : C_u \to C_{v_u}$ in $\mathcal{C}$. In short,

$$\mathsf{Hom}_{\mathcal{C}_\oplus}\left(\bigoplus_{u\in U} C_u\,,\ \bigoplus_{v\in V} C_v\right) \;=\; \prod_{u\in U}\sum_{v\in V} \mathsf{Hom}_{\mathcal{C}}\left(C_u, C_v\right)$$

- The identity at object $\bigoplus_{u\in U} C_u$ is the family $(u, \mathrm{id}_{C_u})_{u\in U}$. Given two composable maps

$$(w_v, h_v)_{v\in V} : \bigoplus_{v\in V} C_v \to \bigoplus_{w\in W} C_w \qquad \text{and} \qquad (v_u, g_u)_{u\in U} : \bigoplus_{u\in U} C_u \to \bigoplus_{v\in V} C_v$$

  the composite is defined to be the family

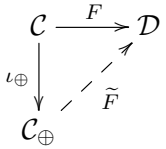
$$(w_{v_u}, h_{v_u} \circ g_u) : \bigoplus_{u\in U} C_u \to \bigoplus_{w\in W} C_w$$

There is a full and faithful functor $\iota_\oplus : \mathcal{C} \to \mathcal{C}_\oplus$ taking an object $C \in \mathsf{Obj}(\mathcal{C})$ to the one-element family $\bigoplus_1 C \in \mathsf{Obj}(\mathcal{C}_\oplus)$. Objects lying in the image of this functor will be called *basic* objects of $\mathcal{C}_\oplus$. The formal sum notation reflects that families $\bigoplus_{u\in U} C_u$ should really be understood as coproducts of those basic objects C_u. More generally, it is straightforward to check that, for any finite set I and family $\bigoplus_{u\in U_i} C_u$ over $i \in I$, canonical coproducts in $\mathcal{C}_\oplus$ can be computed as follows

$$\bigoplus_{i\in I}\bigoplus_{u\in U_i} C_{i,u} \quad = \quad \bigoplus_{(i,u)\in\sum_{i\in I} U_i} C_{i,u}$$

Remark 4.3.2. Observe that here, we have two slightly different meanings for the $\bigoplus$ operator: $\bigoplus_{u\in U_i} C_{i,u}$ is a notation for a finite family of objects, and thus unambiguously refers to an object of $\mathcal{C}_\oplus$, while $\bigoplus_{i\in I}(\dots)$ is a coproduct in the usual sense, which is therefore defined only up to unique isomorphism. We will freely mix both uses in the rest of this chapter, as there is no harm risked by confusing the two.

As advertised, this is a free finite coproduct completion in the following sense: for any functor $F : \mathcal{C} \to \mathcal{D}$ to a category $\mathcal{D}$ with finite coproducts, there is an extension $\widetilde{F} : \mathcal{C}_\oplus \to \mathcal{D}$ preserving finite coproducts making the following diagram commute:



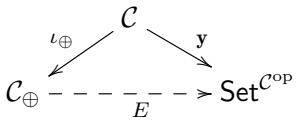
and it is unique up to unique natural isomorphism under those conditions.

Finally, suppose that we are a monoidal structure on $\mathcal{C}$. Then, it is possible to extend it to a monoidal structure over $\mathcal{C}_\oplus$ in a rather canonical way: we require that $\otimes$ distributes over $\oplus$, i.e., that $A \otimes (B \oplus C) \cong (A \otimes B) \oplus (A \otimes C)$. Formally speaking, we set

$$\left(\bigoplus_{u\in U} C_u\right) \otimes \left(\bigoplus_{v\in V} C_v\right) \quad = \quad \bigoplus_{(u,v)\in U\times V} C_u \otimes C_v$$

If $\mathbf{I}$ is the unit of the tensor product in $\mathcal{C}$, then the basic object $\bigoplus_1 \mathbf{I}$ is taken to be the unit of the tensor product in $\mathcal{C}_\oplus$. An affine symmetric monoidal structure on $\mathcal{C}$ can be lifted in a satisfactory manner to this new tensor product (in particular $\iota_\oplus(\top)$ is still terminal).

Remark 4.3.3. For readers more familiar with the free cocompletion $\mathsf{Set}^{\mathcal{C}^{\mathrm{op}}}$ of $\mathcal{C}$, note that the coproduct-preserving functor E determined by



is full and faithful, as well as strong monoidal when $\mathsf{Set}^{\mathcal{C}^{\mathrm{op}}}$ is equipped with the *Day convolution* as monoidal product.

The following property, which is arguably obvious from the definition of $\mathcal{C}_\oplus$, will turn out to be quite important later.

Proposition 4.3.4. *For any $A, B \in \mathsf{Obj}(\mathcal{C}_\oplus)$, there is a natural isomorphism*

$$\mathsf{Hom}_{\mathcal{C}_\oplus}(\iota_\oplus(-), A \oplus B) \quad \cong \quad \mathsf{Hom}_{\mathcal{C}_\oplus}(\iota_\oplus(-), A) + \mathsf{Hom}_{\mathcal{C}_\oplus}(\iota_\oplus(-), B)$$

Remark 4.3.5. Compare with the following natural isomorphism, which holds for any category $\mathcal{D}$ and $A, B \in \mathsf{Obj}(\mathcal{D})$:

$$\mathsf{Hom}_{\mathcal{D}}(A \oplus B, -) \quad \cong \quad \mathsf{Hom}_{\mathcal{D}}(A, -) \times \mathsf{Hom}_{\mathcal{D}}(B, -)$$

In fact, this characterizes the coproduct of A and B. Just like Proposition 4.1.4, this is an instance of a general equivalence between universal properties and natural bijections involving homsets.

4.3.1. **Conservativity over affine monoidal settings.** First, note that the coproduct completion can be lifted at the level of streaming settings.

Definition 4.3.6. Given a streaming setting $\mathfrak{C} = (\mathcal{C}, \top, \bot, (\!|-|\!)_{\mathcal{C}})$, define $\mathfrak{C}_\oplus$ as the tuple

$$(\mathcal{C}_\oplus, \iota_\oplus(\top), \iota_\oplus(\bot), (\!|-|\!)_{\mathcal{C}_\oplus})$$

where $(\!|-|\!)_{\mathcal{C}_\oplus}$ is obtained by precomposing the canonical isomorphism (recalling that $\iota_\oplus$ is full and faithful)

$$\mathsf{Hom}_{\mathcal{C}_\oplus}(\iota_\oplus(\top), \iota_\oplus(\bot)) \cong \mathsf{Hom}_{\mathcal{C}}(\top, \bot)$$

Before moving on, let us make the following definition: an object A in a monoidal category $(\mathcal{C}, \otimes, \mathbf{I})$ is said to have *unitary support*if there exists a map $\mathbf{I} \to A$. This is quite useful in affine categories for transductions, as it ensures the following.

Lemma 4.3.7. *Let $\mathcal{C}$ be a symmetric affine monoidal category. Then, for any pair of finite families $(C_u)_{u \in U}$ and $(C_v)_{v \in V}$ of objects of $\mathcal{C}$ such that all C_u and C_v have unitary support, we have a $U \times V$-indexed family of embeddings*

$$\mathit{padwithjunk}_{u,v} : \mathsf{Hom}_{\mathcal{C}}(C_u, C_v) \to \mathsf{Hom}_{\mathcal{C}}\left(\bigotimes_{u \in U} C_u, \bigotimes_{v \in V} C_v\right)$$

The basic idea behind Lemma 4.3.7 can be pictured using string diagrams as in Figure 4.3.1: a morphism $C_u \to C_v$ can be pictured as a single string, which is to be embedded in a diagram with U-many inputs and V-many outputs. The fact that $\mathcal{C}$ is affine allows us to cut all input strings for $u' \neq u$ using a weakening node, and unitary support allow us to create some "junk" strings with no input to connect to those $v' \neq v$. This might fail for arbitrary symmetric affine monoidal categories: take for instance the category of finite sets and surjections between them, with the coproduct as a monoidal product.

We are now ready to state our first theorem asserting that, in those favorable circumstances, coproduct completions do not give rise to more expressive SSTs.

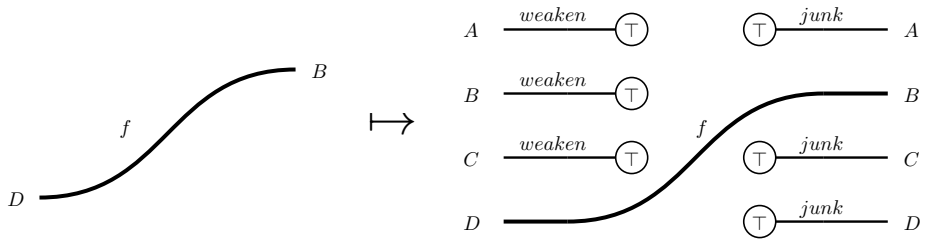


FIGURE 4.3.1. $\texttt{padwithjunk}_{D,B} : \mathsf{Hom}_{\mathcal{C}}(D, C) \to \mathsf{Hom}_{\mathcal{C}}(A \otimes B \otimes C \otimes D, A \otimes B \otimes C \otimes D)$

Theorem 4.3.8. *Let $\mathfrak{C}$ be an affine symmetric monoidal streaming setting where all objects C such that $\mathsf{Hom}_{\mathcal{C}}(\top\!\!\top, C) \neq \varnothing \neq \mathsf{Hom}_{\mathcal{C}}(C, \perp\!\!\!\perp)$ have unitary support.*

$\mathfrak{C}$-SSTs are equivalent to $\mathfrak{C}_{\oplus}$-SSTs.

PROOF. Since $\iota_{\oplus}$ extends to a morphism of streaming settings, $\mathfrak{C}_{\oplus}$-SSTs subsume $\mathfrak{C}$-SSTs.

Conversely, let $\mathcal{T} = \left(Q, q_0, \bigoplus_{u\in U} C_u, \delta, i, o\right)$ be a $\mathfrak{C}_{\oplus}$-SST with input Σ^* and C_u basic objects. Then, we construct a $\mathfrak{C}$-SST

$$\mathcal{T}' = \left(Q \times U,\ (q_0, u_0),\ \bigotimes_{u\in U} C_u,\ \delta',\ i_{u_0},\ o'\right)$$

such that $[\![\mathcal{T}]\!] = [\![\mathcal{T}']\!]$. We define successively (u_0, i_{u_0}), δ' and o'.

- We have $i \in \mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\iota_{\oplus}(\top), \bigoplus_{i\in I} C_i\right)$ which can be rewritten as a factorization

$$\iota_{\oplus}(\top) = \bigoplus_{1} \top\!\!\top \xrightarrow{(\mathsf{in}_{u_0}, * \mapsto \mathrm{id})} \bigoplus_{U} \top\!\!\top \xrightarrow{(\mathrm{id}, u \mapsto i_u)} \bigoplus_{u\in U} C_u$$

  for some $u_0 \in U$ (the i_u for $u \neq u_0$ are taken arbitrarily thanks to the assumption that the C_u have unitary support). This u_0 is the second component of the initial state of $\mathcal{T}'$.
- We set $\delta'(a, (q, u)) = (q', \alpha_u(f'))$ if $\delta(a, q) = (q', f')$, where

$$(\alpha_u)_{u\in U} : \prod_{u\in U}\left[\mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\bigoplus_{u\in U} C_u, \bigoplus_{u\in U} C_u\right) \to \mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\bigotimes_{u\in U} C_u, \bigotimes_{u\in U} C_u\right)\right]$$

  is defined by taking the pointwise composite of

$$\begin{aligned}
\widetilde{\alpha}_u :&\quad \mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\bigoplus_{u\in U} C_u, \bigoplus_{u'\in U} C_{u'}\right) &&\to \sum_{u'\in U} \mathsf{Hom}_{\mathcal{C}_{\oplus}}(C_u, C_{u'})\\
\beta_u :&\quad \sum_{u'\in U} \mathsf{Hom}_{\mathcal{C}_{\oplus}}(C_u, C_{u'}) &&\to U \times \mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\bigotimes_{u\in U} C_u, \bigotimes_{u\in U} C_u\right)\\
\pi :&\quad U \times \mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\bigotimes_{u\in U} C_u, \bigotimes_{u\in U} C_u\right) &&\to \mathsf{Hom}_{\mathcal{C}_{\oplus}}\left(\bigotimes_{u\in U} C_u, \bigotimes_{u\in U} C_u\right)
\end{aligned}$$

  where $\widetilde{\alpha}_u$ is obtained by evaluating its input $f \in \prod_{u\in U}\sum_{u'\in U} \mathsf{Hom}_{\mathcal{C}_{\oplus}}(C_u, C_{u'})$ at u, $\beta_u = \sum_{u'} \texttt{padwithjunk}_{u,u'}$ (with $\texttt{padwithjunk}$ given as per Lemma 4.3.7) and π taken to be the second projection.
- Finally, we set $o'(q, u) \in \mathsf{Hom}_{\mathcal{C}}\left(\bigotimes_{v\in U} C_v, \perp\!\!\!\perp\right)$ to $\widetilde{o}_u \in \mathsf{Hom}_{\mathcal{C}}(C_u, \perp\!\!\!\perp)$ precomposed with the projection $\pi_u \in \mathsf{Hom}_{\mathcal{C}}\left(\bigotimes_{v\in U} C_v, C_u\right)$.

To conclude, one can check by induction that $[\![\mathcal{T}]\!](w) = [\![\mathcal{T}']\!](w)$ for every $w \in \Sigma^*$. □

Corollary 4.3.9. *$\mathfrak{SR}_{\oplus}$-SSTs are equivalent to $\mathfrak{SR}$-SSTs.*

PROOF. All objects of $\mathcal{SR}$ have unitary support via an induction: tensor with the map $\varepsilon : \mathbf{I} \to \Gamma^*$ corresponding to the empty word at the recursive step. □

4.3.2. **State-dependent memory SSTs.** The free coproduct completion encourages us to define the notion of *state-dependent memory SST*, generalizing usual copyless SSTs as follows: instead of taking a single object $C \in \mathsf{Obj}(\mathcal{C})$ as an abstract infinitary memory, we allow to take a family $(C_q)_{q\in Q} \in \mathsf{Obj}(\mathcal{C})^Q$ indexed by the states of the SST.

Definition 4.3.10. A *state-dependent memory* $\mathfrak{C}$*-SST* (henceforth abbreviated sdm-$\mathfrak{C}$-SST or sdmSST when $\mathfrak{C}$ is clear form context) with input Σ^* is a tuple $(Q, q_0, \delta, (C_q)_{q\in Q}, i, o)$ where

- Q is a finite set of states
- $q_0 \in Q$ is some initial state
- $\delta : \Sigma \to \prod_{q\in Q} \sum_{r\in Q} \mathsf{Hom}_{\mathcal{C}}(C_q, C_r)$ is a transition function
- $i \in \mathsf{Hom}_{\mathcal{C}}(\top, C_{q_0})$ is the initialization morphism
- $o \in \prod_{q\in Q} \mathsf{Hom}_{\mathcal{C}}(C_q, \bot)$ is the output family of morphism

In the sequel, we shall often use sdmSSTs because we find them convenient to give more elegant constructions that produce little "junk", as is encoded in Lemma 4.3.7. They essentially give the full power of coproducts in any given situation as shown below.

Lemma 4.3.11. *Let $\mathfrak{C}$ be a streaming setting. State-dependent memory $\mathfrak{C}$-SSTs are exactly as expressive as $\mathfrak{C}_\oplus$-SSTs.*

PROOF SKETCH. Given a $\mathfrak{C}_\oplus$-SST $\left(Q, q_0, \bigoplus_{u\in U} C_u, \delta, i, o\right)$ where $i(*) = \mathsf{in}_{u_0}(i')$ one may check that the following sdm-$\mathfrak{C}$-SST computes the same function:

$$\left(Q \times U, (q_0, u_0), (C_u)_{(q,u)\in Q\times U}, \delta', i', (o(q)_u)_{(q,u)\in Q\times U}\right)$$

where $\delta'(a)_{q,u} = ((r, u'), f)$ if and only if $\delta(a, q) = (r, v)$ and $v_u = (u', f)$.

Conversely, letting $(Q, q_0, (C_q)_{q\in Q}, \delta, i, o)$ be a sdm-$\mathfrak{C}$-SST, an equivalent $\mathfrak{C}_\oplus$-SST is given by $(Q, q_0, \bigoplus_{q\in Q} C_q, \delta', \mathsf{in}_{q_0}(i), o')$, where it is sufficient to define $o'(q)$ as $(o_q)_q$ and to ensure that if $\delta'(a, q) = (r, (r_{q'}, f_{q'})_{q'\in Q})$, then $\delta(a)_q = (r, f_q)$ and $r_q = r$. This can be done. □

Finally, let us remark that the notions of single-state, "normal" and state-dependent memory $\mathfrak{C}$-SSTs coincide if $\mathfrak{C}$ has all coproducts.

Lemma 4.3.12. *If $\mathfrak{C}$ is a streaming setting with coproducts, single-state $\mathfrak{C}$-SSTs are as expressive as general $\mathfrak{C}$-SSTs and sdm-$\mathfrak{C}$-SSTs.*

PROOF. Take a sdmSST $(Q, q_0, (C_q)_{q\in Q}, \delta, i, o)$ to the single-state SST

$$\left(\{\bullet\}, \bullet, \bigoplus_{q\in Q} C_q,\ \delta',\ \mathsf{in}_{q_0} \circ i,\ [o(q)]_{q\in Q}\right)$$

where δ' is defined from δ through the maps

$$\left[\prod_{q\in Q}\sum_{r\in Q} \mathsf{Hom}_{\mathcal{C}}(C_q, C_r)\right]^\Sigma \to \left[\prod_{q\in Q} \mathsf{Hom}_{\mathcal{C}}\left(C_q, \bigoplus_{r\in Q} C_r\right)\right]^\Sigma \xrightarrow{\sim} \left[\mathsf{Hom}_{\mathcal{C}}\left(\bigoplus_{q\in Q} C_q, \bigoplus_{r\in Q} C_r\right)\right]^\Sigma$$

□

Remark 4.3.13. The comparison between single-state, standard and state-dependent memory $\mathfrak{C}$-SSTs can be summed up in terms of completion with the following "equalities":

$$\mathfrak{C}\text{-SSTs} = \text{single-state } \mathfrak{C}_{\oplus\text{const}}\text{-SSTs} \qquad \text{sdm-}\mathfrak{C}\text{-SSTs} = \text{single-state } \mathfrak{C}_{\oplus}\text{-SSTs}$$

where $\mathfrak{C}_{\oplus\text{const}}$ designates the following restriction of $\mathfrak{C}_{\oplus}$: the category $\mathcal{C}_{\oplus}$ is restricted to the full subcategory $\mathcal{C}_{\oplus\text{const}}$ whose objects are constant formal sums $\bigoplus_{i\in I} C$ for some $C \in \mathsf{Obj}(\mathcal{C})$.

4.3.3. **Some function spaces in $\mathcal{SR}_{\oplus}$.** Now we study $\mathcal{SR}_{\oplus}$ in some more detail. This category is unfortunately not able to interpret even the $\otimes/\multimap$ fragment of $\lambda\ell^{\oplus\&}$, because, like $\mathcal{SR}$, it lacks internal homsets $A \multimap B$ for every pair $(A, B) \in \mathsf{Obj}(\mathcal{SR}_{\oplus})^2$. However, they exist when A lies in the image of $\iota_{\oplus}$. This will turn out to be very useful later on.

The reason this works is already present in the proof of Lemma 3.3.7: copyless register assignments may be effectively coded using a combination of finite state – given by the coproduct completion – and a larger set of registers, in such a way that certain operations on those assignments can represented themselves as a copyless assignment combined with a function on states. What follows can be seen as a slight extension, using categorical terminology, of the arguments that we gave to prove Lemma 3.3.7.

Lemma 4.3.14. *Let $R, S \in \mathsf{Obj}(\mathcal{SR})$. There is an internal homset $\iota_{\oplus}(R) \multimap \iota_{\oplus}(S)$ in $\mathcal{SR}_{\oplus}$.*

In Example 4.3.15, we work through the proof below in a concrete case.

Proof. First, recall that $\iota_{\oplus}$ is full and faithful, and that it is thus pertinent to focus our preliminary analysis on morphisms in $\mathcal{SR}$. Recall that a register transition $f : R \to S$, which is a set-theoretical map $S \to (\Gamma + R)^*$ where for every $r \leq R$, $\sum_{s\in S} |f(s)|_{\mathsf{in}_2(r)} \leq 1$ (i.e., it is copyless). Consider the map $(\Gamma + R)^* \to R^*$ erasing the letters of Γ. Then, the image of the induced map $p : \mathsf{Hom}_{\mathcal{SR}}(R, S) \to [S \to R^*]$ is clearly finite because of copylessness. In fact, letting $\mathsf{LO}(X)$ be the set of all total orders over some set X, we have an isomorphism between the image of $\mathsf{Hom}_{\mathcal{SR}}(R, S)$ under p and the following dependent sum

$$\mathcal{O}(R, S) \quad = \quad \sum_{\hat{f}:R\rightharpoonup S} \prod_{s\in S} \mathsf{LO}(\hat{f}^{-1}(s))$$

The intuition is that $\hat{f}$ tracks where register variables in R get affected and the additional data encode in which order they appear in an affectation. Once this crucial finitary information $(\hat{f}, (<_s)_{s\in S})$ is encoded in the internal homset using coproducts, it only remains to recover the information we erased with p, i.e. what words in Γ^* located between occurrences of register variables. This information cannot be bounded by the size of R and S, but the number of intermediate words can; we may index them by $S + \mathsf{dom}(\hat{f})$.

Putting everything together, it means that we take

$$\iota_{\oplus}(R) \multimap \iota_{\oplus}(S) \quad = \quad \bigoplus_{(\hat{f},<)\in\mathcal{O}(R,S)} \iota_{\oplus}(S + \mathsf{dom}(\hat{f}))$$

Now, we need to define the evaluation map $\mathrm{ev}_{R,S} : [\iota_{\oplus}(R) \multimap \iota_{\oplus}(S)] \otimes \iota_{\oplus}(R) \to \iota_{\oplus}(S)$. Recall that the tensor distributes over $\oplus$, so we really need to exhibit $\mathrm{ev}_{R,S}$ in

$$\mathsf{Hom}_{\mathcal{SR}_{\oplus}}\left(\bigoplus_{(\hat{f},<)} \iota_{\oplus}(S + \mathsf{dom}(\hat{f}) + R)\,,\ \iota_{\oplus}(S)\right) \cong \prod_{(\hat{f},<)} \mathsf{Hom}_{\mathcal{SR}}\left(S + \mathsf{dom}(\hat{f}) + R\,,\ S\right)$$

where the indices $(\hat{f}, <)$ ranges over $\mathcal{O}(R, S)$ on both sides. Call $\mathrm{ev}_{R,S,\hat{f},<}$ the corresponding family of $\mathcal{SR}$-morphisms, whose members are set-theoretic maps $S \to (S + \mathsf{dom}(\hat{f}) + R)^*$.

Calling $\{r_1, \dots, r_k\}$ the subset of $\mathsf{dom}(\hat{f})$ ordered by $r_1 < \dots < r_k$, we set

$$\mathrm{ev}_{R,S,\hat{f},<}(s) \;=\; \mathsf{in}_0(s)\mathsf{in}_2(r_1)\mathsf{in}_1(r_1)\dots\mathsf{in}_2(r_k)\mathsf{in}_1(r_k)$$

This concludes the definition of ev. We now leave checking that this satisfies the required universal property to the reader. □

Example 4.3.15. Let us illustrate this construction in a simple case. Consider the following register transition for the concrete base alphabet $\{a, b\}$ and register names x, y, z, u, r, s:

$$\begin{array}{rcl} r & \leftarrow & zaxabyaa \\ s & \leftarrow & bab \end{array}$$

Up to the evident isomorphism $\{x, y, z, u\} \cong \{x\} \otimes \{y, z, u\}$, this determines a morphism $f \in \mathsf{Hom}_{\mathcal{SR}}\left(\{x\} \otimes \{y, z, u\}, \{r, s\}\right)$. Let us describe the unique map $\Lambda(f)$ in

$$\mathsf{Hom}_{\mathcal{SR}_\oplus}\left(\iota_\oplus(\{x\}),\ \iota_\oplus(\{y, z, u\}) \multimap \iota_\oplus(\{r, s\})\right) \cong \sum_{\hat{f},<} \mathsf{Hom}_{\mathcal{SR}}\left(\{x\}, \{r, s\} + \mathsf{dom}(\hat{f})\right)$$

such that $\mathrm{ev} \circ (\Lambda(f) \otimes \mathrm{id}) = f$. On its first component, we set $\hat{f}(y) = \hat{f}(z) = r$ and leave it undefined on u; as for the order, we set $z <_r y$. The last component corresponds to the register transition h depicted below on the left, where, for legibility, we write $\mathbf{r}, \mathbf{s}, \mathbf{z}$ and $\mathbf{y}$ for $\mathsf{in}_1(r), \mathsf{in}_1(s), \mathsf{in}_2(z)$ and $\mathsf{in}_2(y)$. Using the same notations, we also display on the right the relevant component of $\mathrm{ev}_{\{y,z,u\},\{r,s\},\hat{f},<} \in \mathsf{Hom}_{\mathcal{SR}}\left((\{r, s\} + \{z, y\}) + \{z, y\}, \{r, s\}\right)$, so that the reader may convince themself that the composite $\mathrm{ev}_{\{y,z,u\},\{r,s\},\hat{f},<} \circ (h \otimes \mathrm{id}_{\{y,z,u\}})$ is indeed f.

$$\begin{array}{rcl} \mathbf{r} & \leftarrow & \varepsilon \\ \mathbf{z} & \leftarrow & axab \\ \mathbf{y} & \leftarrow & aa \\ \mathbf{s} & \leftarrow & bab \end{array} \qquad\qquad \begin{array}{rcl} r & \leftarrow & \mathsf{in}_1(\mathbf{r})\ \mathbf{y}\ \mathsf{in}_1(\mathbf{y})\ \mathbf{z}\ \mathsf{in}_1(\mathbf{z}) \\ s & \leftarrow & \mathsf{in}_1(\mathbf{s}) \end{array}$$

Lemma 4.3.14 can be extended to define internal homsets $\iota_\oplus(R) \multimap C$ for arbitrary $C \in \mathsf{Obj}(\mathcal{SR}_\oplus)$ through the natural isomorphism of Proposition 4.3.4. However, extending this to all homsets (i.e. allowing any object of $\mathcal{SR}_\oplus$ in the left-hand side) seems impossible: the lack of *products* prevents us from doing so.

4.4. The product completion (or non-determinism)

The above point (among others) leads us to study the *free finite product completion* of streaming settings. As for coproducts, we first discuss the categorical construction before turning to the expressiveness. Here, the situation is more intricate as it turns out that sdm-$\mathfrak{C}_\&$-SSTs of interest will roughly have the power of *non-deterministic* sdm-$\mathfrak{C}$-SSTs. We make that connection precise.

Thankfully, a determinization theorem for usual copyless SSTs, i.e. $\mathfrak{SR}$-SSTs, exists in the literature [AD11] (the proof in the given reference is indirect and goes through monadic second-order logic). It could be applied without difficulty to show that non-determinism does not increase the power of *sdm*-$\mathfrak{SR}$-SSTs. Nevertheless, to keep the exposition self-contained *and* illustrate our framework, we give in Section 4.8 a direct determinization argument generalized to our setting by using, crucially, the concept of internal homsets (and Lemma 4.3.14 for the desired application).

Without further ado, let us formally introduce the product completion.

Definition 4.4.1. Let $\mathcal{C}$ be a category. Its *free finite product completion* is $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$.

In other words, it is the "dual" of the coproduct completion. While conceptually immaculate, this definition merits a bit of unfolding. The objects of $\mathcal{C}_{\&}$ are still finite families $(C_x)_{x\in X}$ – in this context, we write them as formal products $\&_{x\in X} C_x$. As for homsets, we have the dualized situation:

$$\mathsf{Hom}_{\mathcal{C}_{\&}}\left(\mathop{\&}_{x\in X} C_x\,,\ \mathop{\&}_{y\in Y} C_y\right) \quad\cong\quad \prod_{y\in Y}\sum_{x\in X} \mathsf{Hom}_{\mathcal{C}_{\&}}\left(C_x, C_y\right)$$

We also have a full and faithful functor $\iota_{\&} : \mathcal{C} \to \mathcal{C}_{\&}$ with a similar universal property as for the coproduct completion.

As with the coproduct completion, one may want to produce a tensor product in $\mathcal{C}_{\oplus}$ if the underlying category $\mathcal{C}$ has one. The very same recipe can be applied: we define the tensor so that the distributivity $A \otimes (B \,\&\, C) \cong (A \otimes B) \,\&\, (A \otimes C)$ holds.

$$\left(\mathop{\&}_{x\in X} C_x\right) \otimes \left(\mathop{\&}_{y\in Y} C_y\right) \quad=\quad \mathop{\&}_{(x,y)\in X\times Y} C_x \otimes C_y$$

Remark 4.4.2. One might be disturbed by this distributivity of $\otimes$ over &, which goes against the non-linear intuition of thinking of & and $\otimes$ as "morally the same" This feeling may also be exacerbated by the familiar iso

$$\mathsf{Hom}_{\mathcal{SR}_{\oplus}}\left(\top, R \otimes S\right) \cong \mathsf{Hom}_{\mathcal{SR}_{\oplus}}\left(\top, R\right) \times \mathsf{Hom}_{\mathcal{SR}_{\oplus}}\left(\top, S\right)$$

This indeed becomes false when going from $\mathcal{SR}$ to $\mathcal{SR}_{\&}$.

Remark 4.4.3. While $\mathcal{C}_{\&}$ inherits a monoidal product from $\mathcal{C}$ much like with the coproduct completion, it does *not* preserve the *affineness* of monoidal products. The product completion indeed adds a new terminal object, namely the empty family, which can *never* be isomorphic to the singleton family $\iota_{\&}(\top)$. More generally, $\iota_{\&}$ preserves colimits rather than limits.

It is also straightforward to extend the product completion to streaming settings.

4.4.1. **Relationship with non-determinism.** At this juncture, our goal is to prove the equivalence between sdm-$\mathfrak{SR}_{\&}$-SSTs and sdm-$\mathfrak{SR}$-SSTs. One direction is trivial; for the other, we actually prove that sdm-$\mathfrak{SR}_{\&}$-SSTs are subsumed by sdm-$\mathfrak{SR}_{\oplus}$-SSTs.

This result involves some non-trivial combinatorics. We prove it via *uniformization* of non-deterministic sdm-$\mathfrak{SR}$-SSTs, a mild generalization of determinization[4].

Our non-deterministic devices make *finitely branching* choices, following the case of the usual non-deterministic SSTs [AD11, Section 2.2]. To express this, we use the notation $\mathcal{P}_{<\infty}(X)$ for the set of *finite* subsets of a set X. (Note that if Q is some finite set of states, we have the simplification $\mathcal{P}(X) = \mathcal{P}_{<\infty}(X)$.)

Definition 4.4.4. Let $\mathfrak{C}$ be streaming setting with output X. A *non-deterministic* sdm-$\mathfrak{C}$-SST with input Σ^* and output X is a tuple $(Q, I, (C_q)_{q\in Q}, \Delta, i, o)$ where

- Q is a finite set of states and $I \subseteq Q$
- $(C_q)_{q\in Q}$ is a family of objects of $\mathcal{C}$
- Δ is a finite transition relation: $\Delta \in \mathcal{P}_{<\infty}\left(\Sigma \times \sum_{(q,r)\in Q^2} \mathsf{Hom}_{\mathcal{C}}\left(C_q, C_r\right)\right)$
- $i \in \prod_{q_0\in I} \mathsf{Hom}_{\mathcal{C}}\left(\top, C_{q_0}\right)$ is a family of input morphisms

[4] We work with uniformization here we find it slightly more convenient to handle. This choice of uniformization over determinization is rather inessential.

- $o \in \prod_{q \in Q} (\mathsf{Hom}_{\mathcal{C}}(C_q, \perp\!\!\!\perp) + 1)$ is a family of partial output morphisms

A *partial* sdm-$\mathfrak{C}$-SST $(Q, q_0, (C_q)_{q\in Q}, \delta, i, o)$ has the same definition as a (deterministic) sdm-$\mathfrak{C}$-SST (Definition 4.3.10), except for the output o, which is allowed to be partial, just as in the last item above.

By transposing the usual notion of run of a non-deterministic finite automaton, one sees that a non-deterministic sdm-$\mathfrak{C}$-SST $\mathcal{T}$ gives rise to a function $[\![\mathcal{T}]\!] : \Sigma^* \to \mathcal{P}_{<\infty}(X)$ (for an input alphabet Σ and output set X). Similarly, a partial sdm-$\mathfrak{C}$-SST $\mathcal{T}'$ is interpreted as a partial function $[\![\mathcal{T}]\!] : \Sigma^* \rightharpoonup X$.

Remark 4.4.5. In line with Remark 4.3.13, we may describe non-deterministic sdm-$\mathfrak{C}$-SSTs as single-state (deterministic) SSTs over an enriched streaming setting.

Let $\mathfrak{C} = (\mathcal{C}, \top\!\!\!\top_{\mathfrak{C}}, \perp\!\!\!\perp_{\mathfrak{C}}, (\!|-|\!)_{\mathfrak{C}})$, with output X. We first define the category $\mathsf{NFA}(\mathcal{C})$:

- its objects consist of a finite set Q with a family $(C_q)_{q\in Q} \in \mathsf{Obj}(\mathcal{C})^Q$;
- its morphisms are $\mathsf{Hom}_{\mathsf{NFA}(\mathcal{C})}\left((C_q)_{q\in Q}, (C'_r)_{r\in R}\right) = \mathcal{P}_{<\infty}\left(\sum_{(q,r)\in Q\times R} \mathsf{Hom}_{\mathcal{C}}(C_q, C_r)\right)$
- the composition of morphisms extends that of binary relations:

$$\varphi \circ \psi = \{(q,\ s,\ f \circ g) \mid (q,\ r,\ f : C_q \to C'_r) \in \varphi,\ (r,\ s,\ g : C'_r \to C''_s) \in \psi\}$$

To lift this to a construction $\mathsf{NFA}(\mathfrak{C})$ on streaming settings, we take:

- $\top\!\!\!\top_{\mathsf{NFA}(\mathfrak{C})}$ and $\perp\!\!\!\perp_{\mathsf{NFA}(\mathfrak{C})}$ are the $\{\bullet\}$-indexed families containing respectively $\top\!\!\!\top_{\mathfrak{C}}$ and $\perp\!\!\!\perp_{\mathfrak{C}}$, so that $\mathsf{Hom}_{\mathsf{NFA}(\mathcal{C})}\left(\top\!\!\!\top_{\mathsf{NFA}(\mathfrak{C})}, \perp\!\!\!\perp_{\mathsf{NFA}(\mathfrak{C})}\right) = \mathcal{P}_{<\infty}\left(\{(\bullet,\bullet)\} \times \mathsf{Hom}_{\mathcal{C}}\left(\top\!\!\!\top_{\mathfrak{C}}, \perp\!\!\!\perp\mathfrak{C}\right)\right)$
- $(\!|\varphi|\!)_{\mathsf{NFA}(\mathfrak{C})} = \{(\!|f|\!)_{\mathfrak{C}} \mid (\bullet,\bullet,f) \in \varphi\} \subseteq X$, so that the output set of a $\mathsf{NFA}(\mathfrak{C})$-SST is $\mathcal{P}_{<\infty}(X)$.

There is a slight mismatch between the above definition of non-deterministic sdm-$\mathfrak{C}$-SSTs and single-state $\mathsf{NFA}(\mathfrak{C})$-SSTs: in the latter, two distinct input (resp. output) morphisms may correspond to the same initial (resp. final) state. However, the former can encode such situations by enlarging the set of states (to keep it finite, the use of $\mathcal{P}_{<\infty}(-)$ in the definitions is crucial).

One can also give an analogous account of partial sdm-$\mathfrak{C}$-SSTs; we leave the interested reader to work out the details.

Coming back to our main point, we now state the slight variation of the determinization theorem for copyless SSTs [AD11] that fits our purposes.

Definition 4.4.6. Given an arbitrary function $F : X \to \mathcal{P}(Y)$, we say that $f : X \rightharpoonup Y$ *uniformizes* F if and only if $\mathsf{dom}(f) = X \setminus F^{-1}(\varnothing)$ and $\forall x \in \mathsf{dom}(f).\ f(x) \in F(x)$.

Theorem 4.4.7 (variant of [AD11] for state-dependent-memory copyless SSTs)**.** *For every non-deterministic sdm-$\mathfrak{SR}$-SSTs $\mathcal{T}$, there exists a partial deterministic sdm-$\mathfrak{SR}$-SST $\mathcal{T}'$ such that $[\![\mathcal{T}']\!]$ uniformizes $[\![\mathcal{T}]\!]$.*

We now show that studying the uniformization property between classes of sdmSSTs parameterized by streaming setting $\mathfrak{C}$ and $\mathfrak{D}$ is morally the same as comparing the expressiveness of sdm-$\mathfrak{C}_{\&}$-SSTs and sdm-$\mathfrak{D}$-SSTs.

Lemma 4.4.8. *Non-deterministic sdm-$\mathfrak{C}$-SSTs are uniformizable by partial sdm-$\mathfrak{D}$-SSTs if and only if sdm-$\mathfrak{D}$-SSTs subsume sdm-$\mathfrak{C}_{\&}$-SSTs.*

Proof. First, let us assume that sdm-$\mathfrak{C}$-SSTs uniformize sdm-$\mathfrak{D}$-SSTs and let

$$\mathcal{T} = (Q, q_0, (A_q)_{q\in Q}, \delta, i, o) \qquad \text{with} \qquad A_q = \bigotimes_{x\in X_q} C_{q,x}$$

be a $\mathfrak{C}_{\&}$-SST with input Σ. We first define a non-deterministic sdm-$\mathfrak{C}$-SST $\mathcal{T}'$ by setting

$$\mathcal{T}' = \left(I + Q', I, (C_{p(m)})_{m \in I+Q'}, \Delta, i', o'\right) \quad \text{where}$$

$$\begin{array}{rcl} Q' & = & \sum_{q \in Q} X_q \\ I & = & \sum_{x \in X_{q_0}} \{f \mid i_* = (x, f)\} \end{array} \qquad \begin{array}{rccc} p: & I + Q' & \to & Q' \\ & \mathsf{in}_1(x, f) & \mapsto & (q_0, x) \\ & \mathsf{in}_2(q, x) & \mapsto & (q, x) \end{array}$$

$$i'_{\mathsf{in}_1(x,f)} = f \qquad o'_m = \mathsf{in}_1\left(\pi_2\left((o_{\pi_1(p(m))})_{\pi_2(p(m))}\right)\right)$$

$$\Delta = \left\{ (a, ((m, \mathsf{in}_2(r, y)), f)_m) \;\middle|\; \begin{array}{rcl} \forall (x, q) \in Q'. \forall m \in p^{-1}(x, q). & & \\ \pi_1(\delta(a)_q) & = & r \\ \wedge \; \pi_2(\delta(a)_q)_y & = & (x, f) \end{array} \right\}$$

Taking $\mathcal{T}''$ to be a sdm-$\mathfrak{D}$-SST uniformizing $\mathcal{T}'$, we have $\{-\} \circ [\![\mathcal{T}'']\!] = [\![\mathcal{T}']\!] = \{-\} \circ [\![\mathcal{T}]\!]$, so we are done.

For the converse, assume that sdm-$\mathfrak{D}$-SSTs subsume sdm-$\mathfrak{C}_{\&}$-SSTs and suppose we have some non-deterministic sdm-$\mathfrak{C}$-SST $\mathcal{T} = (Q, I, (A_q)_{q \in Q}, \Delta, i, o)$ to uniformize. Fix a total order $\preceq$ over the morphisms of $\mathcal{C}$ occurring in Δ (recall that there are finitely many of them). Consider a partial deterministic sdm-$\mathfrak{C}_{\&}$-SST $\mathcal{T}'$ obtained from $\mathcal{T}$ by a powerset construction

$$\mathcal{T}' = \left(\mathcal{P}(Q),\ I,\ \left(\bigotimes_{r \in R} A_r\right)_{R \subseteq Q},\ \delta,\ i,\ o'\right)$$

where $\delta(a)_R = (S, (r_s, f_s)_{s \in S})$ if and only if $\forall s \in S \left[\begin{array}{l} (a, ((r_s, s), f_s)) \in \Delta \\ \forall g. (a, ((r_s, s), g)) \in \Delta \Rightarrow f_s \preceq g \end{array} \right.$

and $o'_R = o \restriction r$ for some arbitrary r such that the right hand-side is defined; if there is no such r, o'_R is undefined and we call R a *dead* set of states. By padding o' with some arbitrary values on such dead states, we may extend $\mathcal{T}'$ to a non-partial deterministic $\mathfrak{C}$-SST $\mathcal{T}''$ so that $[\![\mathcal{T}']\!] \subseteq [\![\mathcal{T}'']\!]$. We may then consider a sdm-$\mathfrak{D}$-SST $\mathcal{T}''' = (Q', q_0, (D_q)_{q \in Q'}, \delta, i'', o'')$ such that $[\![\mathcal{T}']\!](w) = [\![\mathcal{T}'']\!](w) = [\![\mathcal{T}''']\!]$ for $w \in \mathsf{dom}([\![\mathcal{T}']\!])$. This $\mathcal{T}'''$ is almost our uniformizer; we only need to restrict the domain of its output function. This can be achieved by adding a $\mathcal{P}(Q)$ component to the state space corresponding to the set of states reached by $\mathcal{T}$ and forcing the output function to be undefined if this component contains a dead set of states. □

Putting Lemma 4.4.8 together with Theorem 4.4.7 yields the desired result.

Theorem 4.4.9. *sdm-$\mathfrak{SR}_{\&}$-SSTs are subsumed by sdm-$\mathfrak{SR}$-SSTs.*

Note that, conversely, Theorem 4.4.9 also implies Theorem 4.4.7 through Lemma 4.4.8. We will provide a direct self-contained proof of a categorical generalization of Theorem 4.4.9 in Section 4.8.

4.5. The ⊕&-completion (a Dialectica-like construction)

We now consider the composition $\mathcal{C} \mapsto (\mathcal{C}_{\&})_{\oplus}$ of the coproduct completion with the product completion. Unraveling the formal definition and distributing sums and products at the right spots, we define an isomorphic category $\mathcal{C}_{\oplus\&}$ which is a bit less cumbersome to manipulate in practice. (The proof is by mechanical unfolding of the definitions).

Theorem 4.5.1. *Given an arbitrary category $\mathcal{C}$, there is an* isomorphism of categories *(not just an equivalence) between $(\mathcal{C}_{\&})_{\oplus}$ and the category $\mathcal{C}_{\oplus\&}$ defined below.*

- *The objects of* $\mathcal{C}_{\oplus\&}$ *are triples* $(U, (X_u)_u, (C_{u,x})_{(u,x)})$ *where* U *is a finite set,* $(X_u)_{u\in U}$ *is a family of finite sets and* $C_{u,x}$ *is a family of objects of* $\mathcal{C}$ *indexed by* $(u,x) \in \sum_{u\in U} X_u$*. We drop the first index* u *when it is determined by* $x \in X_u$ *from context and write those objects* $\bigoplus_{u\in U} \&_{x\in X_u} C_x$ *for short.*
- $\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\bigoplus_{u\in U} \&_{x\in X_u} C_x\,,\, \bigoplus_{v\in V} \&_{y\in Y_v} C_y\right) = \prod_{u\in U} \sum_{v\in V} \prod_{y\in Y_v} \sum_{x\in X_u} \mathsf{Hom}_{\mathcal{C}}(C_x, C_y)$
- *Its identities are maps* $\prod_{u\in U} \sum_{u'\in U} \prod_{x'\in X_u} \sum_{x\in X_{u'}} \mathsf{Hom}_{\mathcal{C}}(C_x, C_{x'})$
$$u \mapsto \big[u\ ,\ \big(x \mapsto (x\ ,\ \mathrm{id}_{C_x})\big)\big]$$
- *Composition is defined as in Figure 4.5.1. The interesting steps of this computation are those involving* v *and* y*; since they are similar, let us focus on* v*. A map of the form*
$$\begin{array}{ccccccc} \prod_{v\in V} A_v & \times & \sum_{v\in V} B_v & \longrightarrow & \sum_{v\in V} A_v \times B_v \\ (a_v)_{v\in V} & , & (v',\ b) & \mapsto & (v',\ (a_{v'}\ ,\ b)) \end{array}$$
is applied. This makes the two $\mathcal{C}_{\oplus\&}$*-morphisms interact (an interaction represented in Remark 4.5.2 below as a move in a game): the* v' *provided by the right one selects* $a_{v'}$ *among all the possibilities* $(a_v)_v$ *proposed by the left one.*

The reader may notice that composition in $\mathcal{C}_{\oplus\&}$ is very similar to the interpretation of cuts in Gödel's Dialectica interpretation [Göd58] and/or composition in categories of *polynomial functors* [GK13; MG18]. This intuition can be made formal. In particular, see [Hof11] for a decomposition of a general version of the categorical Dialectica construction into free completions with *simple* sums and products. In our context, the completion with simple coproducts would be the $(-)_{\oplus\mathrm{const}}$ of Remark 4.3.13; conversely, a "dependent Dialectica" could be defined in the fibrational setting of [Hof11] analogously to our $(-)_{\oplus\&}$-completion by removing the simplicity restriction.

Remark 4.5.2. For the uninitiated, it can be helpful to compute this completion on the trivial category $\mathbb{1}$ with one object and only its identity morphism. In this case, objects consists of a pair of a finite set U together with a family $(X_u)_{u\in U}$ of finite sets that can be regarded as a two-move sequential game (with no outcome) between player $\oplus$ and $\&$: first $\oplus$ plays some $u \in U$ and then $\&$ plays some $x \in X_u$. One can then consider *simulation games* between $(U, (X_u)_u)$ (the "left hand-side") and $(V, (Y_v)_v)$ (the "right hand-side") proceeding as follows:

- first, $\&$ plays some $u \in U$ on the left
- then, $\oplus$ plays some $v \in V$ on the right
- $\&$ answers with some $y \in Y_v$ on the right
- finally $\oplus$ answers with $x \in X_u$ on the left.

	$U\ ,\ (X_u)_u$	$\to$	$V,\ (Y_v)_v$
$\&$	u		
$\oplus$			v
$\&$			y
$\oplus$	x		

Morphisms in $\mathbb{1}_{\oplus\&}$ are $\oplus$-strategies in such games. Identities correspond to copycat strategies and composition is (a simple version) of an usual scheme in game semantics. As for $\mathcal{C}_{\oplus\&}$, one may consider that once this simulation game is played, $\oplus$ needs to provide a datum in some $\mathsf{Hom}_{\mathcal{C}}(C_x, C_y)$ which depends on the outcome of the game.

As the coproduct completion preserves limits, $\mathcal{C}_{\oplus\&}$ always boasts both binary cartesian products and coproducts. Concretely, products are computed by distributivity:
$$\&_{i\in I} \bigoplus_{j\in J_i} A_j \quad\cong\quad \bigoplus_{f\in\prod_{i\in I} J_i} \&_{i\in I} A_{f(i)}$$

$$\mathsf{Hom}\left(\bigoplus_{v}\,\&_{y}\, C_y\,,\ \bigoplus_{w}\,\&_{z}\, C_z\right) \times \mathsf{Hom}\left(\bigoplus_{u}\,\&_{x}\, C_x\,,\ \bigoplus_{v}\,\&_{y}\, C_y\right)$$

$$\Big\Vert$$

$$\prod_{v}\sum_{w}\prod_{z}\sum_{y}\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \quad\times\quad \prod_{u}\sum_{v}\prod_{y}\sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)$$

$$\downarrow$$

$$\prod_{u}\left(\prod_{v}\sum_{w}\prod_{z}\sum_{y}\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \times \sum_{v}\prod_{y}\sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)\right)$$

$$\downarrow$$

$$\prod_{u}\sum_{v}\left(\sum_{w}\prod_{z}\sum_{y}\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \quad\times\quad \prod_{y}\sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)\right)$$

$$\downarrow\sim$$

$$\prod_{u}\sum_{v,w}\left(\prod_{z}\sum_{y}\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \quad\times\quad \prod_{y}\sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)\right)$$

$$\downarrow$$

$$\prod_{u}\sum_{v,w}\prod_{z}\left(\sum_{y}\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \quad\times\quad \prod_{y}\sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)\right)$$

$$\downarrow$$

$$\prod_{u}\sum_{v,w}\prod_{z}\sum_{y}\left(\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \quad\times\quad \sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)\right)$$

$$\downarrow\sim$$

$$\prod_{u}\sum_{v,w}\prod_{z}\sum_{y,x}\left(\mathsf{Hom}_{\mathcal{C}}\left(C_y, C_z\right) \times \mathsf{Hom}_{\mathcal{C}}\left(C_x, C_y\right)\right)$$

$$\downarrow\text{composition in } \mathcal{C}$$

$$\prod_{u}\sum_{v,w}\prod_{z}\sum_{y,x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_z\right)$$

$$\downarrow\text{project away } v, y$$

$$\prod_{u}\sum_{w}\prod_{z}\sum_{x}\mathsf{Hom}_{\mathcal{C}}\left(C_x, C_z\right)$$

$$\Big\Vert$$

$$\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\bigoplus_{u}\,\&_{x}\, C_x\,,\ \bigoplus_{w}\,\&_{z}\, C_z\right)$$

FIGURE 4.5.1. Composition in $\mathcal{C}_{\oplus\&}$ ($- \in -$ are omitted from indices)

If $\mathcal{C}$ has a symmetric monoidal structure $(\otimes, \mathbf{I})$, the lifting is computed in $\mathcal{C}_{\oplus\&}$ as

$$\left(\bigoplus_{u\in U} \mathop{\&}_{x\in X_u} C_x\right) \otimes \left(\bigoplus_{v\in V} \mathop{\&}_{y\in Y_v} D_y\right) \quad = \quad \bigoplus_{(u,v)\in U\times V} \mathop{\&}_{(x,y)\in X_u\times Y_v} C_x \otimes D_y$$

This reflects the distributivity of $\otimes$ over & in $\mathcal{SR}_{\&}$ (Remark 4.4.2) – which however does *not* hold in general in $\mathcal{SR}_{\oplus\&}$, unlike the distribuvity of $\otimes$ over $\oplus$.

Finally, we write $\iota_{\oplus\&} : \mathcal{C} \to \mathcal{C}_{\oplus\&}$ for the functor sending an object C to $\bigoplus_1 \&_1 C$ (it is a full and faithful embedding).

4.5.1. **The monoidal closure theorem.** Recall from the introduction (§1.2.3) that a result of central importance in this dissertation is the fact that the category $(\mathcal{SR}_{\&})_{\oplus}$ is symmetric monoidal closed (Theorem 1.2.4). This will be a consequence of the general property below.

Theorem 4.5.3. *Let $(\mathcal{C}, \otimes, \mathbf{I})$ be a symmetric monoidal category. Assume that its coproduct completion $\mathcal{C}_{\oplus}$ admits internal homsets $\iota_{\oplus}(A) \multimap \iota_{\oplus}(B)$ for every $A, B \in \mathsf{Obj}(\mathcal{C})$. Then its Dialectica-like completion $\mathcal{C}_{\oplus\&}$ is monoidal closed.*

Indeed, since Lemma 4.3.14 gives us precisely the assumption on internal homsets in the above statement for $\mathcal{C} = \mathcal{SR}$, we immediately get:

Corollary 4.5.4 (equivalent to Theorem 1.2.4)**.** *$\mathcal{SR}_{\oplus\&}$ is monoidal closed.*

To prove Theorem 4.5.3, our strategy is to establish some isomorphism of the form $\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(C \otimes A, B) \cong \mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(C, \dots)$ which is *natural in C*. While we will not devote too much space to checking naturality down to low-level details, we have attempted to break down this verification into manageable chunks. In particular, our notations are going to stress the functoriality of the various intermediate expressions in our computation. Some lemmas will be useful for this.

Lemma 4.5.5. *We can lift every functor $F : \mathcal{C}^{\mathrm{op}} \to \mathsf{Set}$ to a functor $\langle\&\rangle F : (\mathcal{C}_{\&})^{\mathrm{op}} \to \mathsf{Set}$ such that for $(C_i) \in \mathsf{Obj}(C)^I$,*

$$\langle\&\rangle F\left(\mathop{\&}_{i\in I} C_i\right) \quad = \quad \sum_{i\in I} F(C_i)$$

and in such a way that if $F, G : \mathcal{C}^{\mathrm{op}} \to \mathsf{Set}$ are naturally isomorphic, so are $\langle\&\rangle F$ and $\langle\&\rangle G$.

Proof. Since $(\mathcal{C}_{\&})^{\mathrm{op}} = (\mathcal{C}^{\mathrm{op}})_{\oplus}$ by duality, the existence of $\langle\&\rangle F$ is simply the universal property of the *coproduct* completion! Concretely, consider a morphism

$$f = \left(\varphi_f : I \to J,\ \left(f_i : C'_{\varphi_f(i)} \to C_i\right)_{i\in I}\right) \in \mathsf{Hom}_{\mathcal{C}_{\&}}\left(\mathop{\&}_{j\in J} C'_j,\ \mathop{\&}_{i\in I} C_i\right)$$

Its functorial image is given by

$$\langle\&\rangle F(f) \quad : \quad (i, x) \in \sum_{i\in I} F(C_i) \quad \mapsto \quad (\varphi_f(i), F(f_i)(x)) \in \sum_{j\in J} F(C'_j)$$

Using this explicit expression, one can check that if $\eta_X : F(X) \xrightarrow{\sim} G(X)$ is a natural isomorphism (for X varying over $\mathcal{C}^{\mathrm{op}}$), then

$$(i, x) \in \sum_{i\in I} F(C_i) \quad \mapsto \quad (i, \eta_{C_i}(x)) \in \sum_{i\in I} G(C_i)$$

defines a natural isomorphism between $\langle\&\rangle F$ and $\langle\&\rangle G$. □

Lemma 4.5.6. *We can lift every functor $F : (\mathcal{C}_\&)^{\mathrm{op}} \to \mathsf{Set}$ to $\langle\oplus\rangle F : (\mathcal{C}_{\oplus\&})^{\mathrm{op}} \to \mathsf{Set}$ in such a way that if $F, G : (\mathcal{C}_\&)^{\mathrm{op}} \to \mathsf{Set}$ are naturally isomorphic, so are $\langle\oplus\rangle F$ and $\langle\oplus\rangle G$, and with the action on objects*

$$\langle\oplus\rangle F\left(\bigoplus_{u\in U} \mathop{\&}_{x\in X_u} C_x\right) \quad = \quad \prod_{u\in U} F\left(\mathop{\&}_{x\in X_u} C_x\right)$$

Proof. Write $\mathcal{C}_{\oplus\&} \cong \mathcal{D}_\oplus$ with $\mathcal{D} = \mathcal{C}_\&$. The statement is a special case of a property that works for any category $\mathcal{D}$; analogously (and dually) to the previous lemma, it corresponds to the universal property of the product completion. □

Before we can state the next lemma, we first need a few definitions. Let

$$\iota^\oplus_\& : \mathcal{C}_\oplus \to \mathcal{C}_{\oplus\&} \quad \text{such that} \quad \iota_{\oplus\&} = \iota^\oplus_\& \circ \iota_\oplus$$

be the full and faithful embedding be obtained by applying the universal property of $\mathcal{C}_\oplus$ to the coproduct-preserving functor $\iota_{\oplus\&}$. Furthermore, the isomorphism $\mathcal{C}_{\oplus\&} \cong (\mathcal{C}_\&)_\oplus$ entails the existence of another full and faithful functor

$$\iota'_\oplus : \mathcal{C}_\& \to \mathcal{C}_{\oplus\&} \quad \text{such that} \quad \iota_{\oplus\&} = \iota'_\oplus \circ \iota_\&$$

corresponding to the canonical embedding $\mathcal{D} \to \mathcal{D}_\oplus$ for $\mathcal{D} = \mathcal{C}_\&$. Both those functors have straightforward explicit constructions, whose actions on objects are

$$\iota^\oplus_\&\left(\bigoplus_{i\in I} C_i\right) \quad = \quad \bigoplus_{i\in I} \mathop{\&}_{j\in\{*\}} C_i \qquad\qquad \iota'_\oplus\left(\mathop{\&}_{i\in I} C_i\right) \quad = \quad \bigoplus_{j\in\{*\}} \mathop{\&}_{i\in I} C_i$$

(where the (co)product notation is syntactic sugar for indexed families).

Lemma 4.5.7. *For any $D \in \mathsf{Obj}(\mathcal{C}_\oplus)$, there is a natural isomorphism (with domain $\mathcal{C}_\&$)*

$$\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(-), \iota^\oplus_\&(D)\right) \quad \cong \quad \langle\&\rangle\left[\mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(-), D\right)\right]$$

Proof. Let $C = \mathop{\&}_{i\in I} C_i$ and $D = \bigoplus_{k\in K} D_k$ for $C_i, D_k \in \mathsf{Obj}(\mathcal{C})$. By definition,

$$\begin{aligned}
\mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota'_\oplus(C), \iota^\oplus_\&(D)\right) &= \prod_{p\in\{*\}} \sum_{k\in K} \prod_{q\in\{*\}} \sum_{i\in I} \mathsf{Hom}_{\mathcal{C}}\left(C_i, D_k\right) \\
&\cong \sum_{i\in I} \sum_{k\in K} \mathsf{Hom}_{\mathcal{C}}\left(C_i, D_k\right) \\
&\cong \sum_{i\in I} \mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(C_i), D\right)
\end{aligned}$$

This establishes the isomorphism for each object of $\mathcal{C}_\&$. The naturality condition can be recast as the commutation of the following diagram:

$$\begin{array}{ccccc}
\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(C), \iota^\oplus_\&(D)\right) & \xleftarrow{\sim} & \displaystyle\sum_{i,k} \mathsf{Hom}_{\mathcal{C}}\left(C_i, D_k\right) & \xrightarrow{\sim} & \displaystyle\sum_{i\in I} \mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(C_i), D\right) \\
\Big\downarrow{\scriptstyle -\circ f} & & & & \Big\downarrow{\scriptstyle (i,g)\mapsto(\varphi_f(i), g\circ f_i)} \\
\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(C'), \iota^\oplus_\&(D)\right) & \xrightarrow{\sim} & \displaystyle\sum_{j,k} \mathsf{Hom}_{\mathcal{C}}\left(C'_j, D_k\right) & \xleftarrow{\sim} & \displaystyle\sum_{j\in J} \mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(C'_j), D\right)
\end{array}$$

Fortunately, it can be checked from the definition of composition in $\mathcal{C}_\oplus$ and $\mathcal{C}_{\oplus\&}$ that both paths in this diagrams denote the same map, namely $(i, k, h) \mapsto (\varphi_f(i), k, h \circ f_i)$. □

PROOF OF THEOREM 4.5.3. Let $A, B, C \in \mathsf{Obj}(\mathcal{C}_{\oplus\&})$. By definition, we can write

$$A = \bigoplus_{u\in U} \mathop{\&}_{x\in X_u} A_x \qquad B = \bigoplus_{v\in V} \mathop{\&}_{y\in Y_v} B_y \qquad C = \bigoplus_{w\in W} \mathop{\&}_{z\in Z_w} C_z$$

where $A_x, B_y, C_z \in \mathsf{Obj}(\mathcal{C})$. Using the definition of $\otimes$ in $\mathcal{C}_{\oplus\&}$ and of $\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(-,-)$,

$$\begin{aligned}
\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(C \otimes A, B) &= \mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\bigoplus_{(w,u)\in W\times U} \mathop{\&}_{(z,x)\in Z_w\times X_u} C_z \otimes A_x \,,\; \bigoplus_{v\in V} \mathop{\&}_{y\in Y_v} B_y\right) \\
&= \prod_{w,u} \sum_{v\in V} \prod_{y\in Y_v} \sum_{z,x} \mathsf{Hom}_{\mathcal{C}}(C_z \otimes A_x,\; B_y) \\
&\cong \prod_{w\in W} \prod_{u\in U} \sum_{v\in V} \prod_{y\in Y_v} \sum_{x\in X_u} \sum_{z\in Z_w} \mathsf{Hom}_{\mathcal{C}}(C_z \otimes A_x,\; B_y)
\end{aligned}$$

Using the operators from Lemmas 4.5.5 and 4.5.6, the last expression above can be turned into a functor in C, and the isomorphism is then natural in C, that is:

$$\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(- \otimes A, B) \quad\cong\quad \langle\oplus\rangle \left[\prod_{u\in U} \sum_{v\in V} \prod_{y\in Y_v} \sum_{x\in X_u} \langle\&\rangle\left[\mathsf{Hom}_{\mathcal{C}}(- \otimes A_x,\; B_y)\right]\right]$$

(Any reader who is not convinced that rearranging the indexing of sums/products is innocuous with respect to the morphisms in $\mathcal{C}_{\oplus\&}$ can check the naturality by a brute-force computation.)

For any $A', B' \in \mathsf{Obj}(\mathcal{C})$, we have a sequence of natural isomorphisms

$$\begin{aligned}
\mathsf{Hom}_{\mathcal{C}}\left(- \otimes A', B'\right) &\cong \mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(- \otimes A'), \iota_\oplus(B')\right) \\
&\cong \mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(-) \otimes \iota_\oplus(A'), \iota_\oplus(B')\right) \\
&\cong \mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(-), \iota_\oplus(A') \multimap \iota_\oplus(B')\right)
\end{aligned}$$

by using the strong monoidality and full faithfulness of $\iota_\oplus$ as well as the assumption on internal homsets in $\mathcal{C}_\oplus$. Lemma 4.5.5 allows us to lift natural isomorphisms through $\langle\&\rangle$; combined with Lemma 4.5.7, this gives us

$$\begin{aligned}
\langle\&\rangle\left[\mathsf{Hom}_{\mathcal{C}}\left(- \otimes A', B'\right)\right] &\cong \langle\&\rangle\left[\mathsf{Hom}_{\mathcal{C}_\oplus}\left(\iota_\oplus(-), \iota_\oplus(A') \multimap \iota_\oplus(B')\right)\right] \\
&\cong \mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(-), \iota^\oplus_\&(\iota_\oplus(A') \multimap \iota_\oplus(B'))\right)
\end{aligned}$$

Let us introduce the abbreviation $A' \rightrightarrows B' = \iota^\oplus_\&(\iota_\oplus(A') \multimap \iota_\oplus(B'))$. Using Proposition 4.3.4 and the dual of Remark 4.3.5 for products, we have

$$\begin{aligned}
\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(- \otimes A, B) &\cong \langle\oplus\rangle\left[\prod_{u\in U} \sum_{v\in V} \prod_{y\in Y_v} \sum_{x\in X_u} \mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(-)\,,\; A_x \rightrightarrows B_y\right)\right] \\
&\cong \langle\oplus\rangle\left[\prod_{u\in U} \sum_{v\in V} \prod_{y\in Y_v} \mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(-)\,,\; \bigoplus_{x\in X_u} A_x \rightrightarrows B_y\right)\right] \\
&\cong \cdots \\
&\cong \langle\oplus\rangle\left[\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}\left(\iota'_\oplus(-)\,,\; \overbrace{\mathop{\&}_{u\in U} \bigoplus_{v\in V} \mathop{\&}_{y\in Y_v} \bigoplus_{x\in X_u} A_x \rightrightarrows B_y}^{\text{call this } A\multimap B}\right)\right]
\end{aligned}$$

Note that we get an expression for $A \multimap B$ that mirrors the definition of $\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(-,-)$!

To conclude, it suffices to show that the last functor in the above computation is naturally isomorphic to $\mathsf{Hom}_{\mathcal{C}_{\oplus\&}}(-, A \multimap B)$. The isomorphism can seen as an instance of the universal property of the coproduct, cf. Remark 4.3.5 which unfortunately only states naturality in the "wrong" argument for our purposes. The remaining naturality condition can be verified routinely; we leave it to the reader. □

4.5.2. **Summary of equivalences between $\mathfrak{C}$-SSTs for completions of $\mathfrak{SR}$.** As a finishing touch on the last three sections on free completions, we compile here different characterizations of regular functions.

$$\text{single-state } \mathfrak{SR}_{\oplus\&}\text{-SSTs} \overset{\text{Lemma 4.3.12}}{=} \mathfrak{SR}_{\oplus\&}\text{-SSTs} \overset{\text{Lemma 4.3.11}}{=} \text{sdm-}\mathfrak{SR}_{\&}\text{-SSTs} \overset{\text{Theorem 4.4.9}}{=} \text{sdm-}\mathfrak{SR}\text{-SSTs}$$

$$\text{sdm-}\mathfrak{SR}\text{-SSTs} \underset{\text{Corollary 4.3.9}}{=} \mathfrak{SR}_{\oplus}\text{-SSTs} \underset{\text{Lemma 4.3.12}}{=} \mathfrak{SR}\text{-SSTs} \underset{\text{Fact 4.2.14}}{=} \text{regular functions}$$

4.6. Half of a universal property for $\mathcal{SR}$

Before covering the applications of Theorem 1.2.4 / Corollary 4.5.4 to the $\lambda\ell^{\oplus\&}$-calculus in the next chapter, let us return to categorical automata theory for its own sake.

In this section, we fix a finite alphabet Γ and study the category $\mathcal{SR}(\Gamma)$ further. Observe that $\{\bullet\} \in \mathsf{Obj}(\mathcal{SR}(\Gamma))$, representing a single register, can be equipped with the structure of an internal monoid $(\{\bullet\}, \mu_\bullet, \eta_\bullet)$ by setting

$$\eta_\bullet(\bullet) = \varepsilon \qquad \text{and} \qquad \mu_\bullet(\bullet) = \mathsf{in}_2(\mathtt{l})\mathsf{in}_2(\mathtt{r}) \quad \text{where } \mathtt{l} = \mathsf{in}_1(\bullet) \text{ and } \mathtt{r} = \mathsf{in}_2(\bullet)$$

so that $\mu_\bullet \in [\{\bullet\} \to_{\mathcal{SR}(\Gamma)} \{\bullet\} \otimes \{\bullet\}]$ has the codomain $\Gamma + (\{\bullet\} + \{\bullet\}) = \Gamma + \{\mathtt{l}, \mathtt{r}\}$ when considered as a map between sets. This internal monoid is the key to giving an inductive characterization of $\mathcal{SR}$. Given a string $w = w_1 \dots w_n \in \Gamma^*$, let us write $\widehat{w} \in [\varnothing \to_{\mathcal{SR}(\Gamma)} \{\bullet\}]$ for the register transition defined by the map $\widehat{w} : \bullet \mapsto \mathsf{in}_1(w_1) \dots \mathsf{in}_1(w_n) \in (\Gamma + \varnothing)^*$.

Theorem 4.6.1. *Let $(\mathcal{C}, \otimes, \mathbf{I})$ be an* affine *symmetric monoidal category.*

For any internal monoid (M, μ, η) of $\mathcal{C}$ and any family $(m_c)_{c\in\Gamma} \in \mathsf{Hom}_{\mathcal{C}}(\mathbf{I}, M)$ of morphisms, there exists a strong monoidal functor *$F : \mathcal{SR}(\Gamma) \to \mathcal{C}$ such that:*

- *$F(\varnothing) = \mathbf{I}$, $F(\{\bullet\}) = M$ and $F(\widehat{c}) = m_c$ for every $c \in \Gamma$;*
- *$F(\mu_\bullet) = \mu$ and $F(\eta_\bullet) = \eta$ with the above definitions (implying that $F(\{\mathtt{l}, \mathtt{r}\}) = M \otimes M$);*
- *the isomorphisms $\mathbf{I} \to F(\varnothing)$ and $F(\{\bullet\}) \otimes F(\{\bullet\}) \to F(\{\bullet\} + \{\bullet\})$ that are part of the strong monoidal structure for F are equal to $\mathrm{id}_{\mathbf{I}}$ and $\mathrm{id}_{M\otimes M}$ respectively.*

Note that since F is a monoidal functor, it transports the monoid object $(\{\bullet\}, \mu_\bullet, \eta_\bullet)$ to a structure of internal monoid over $F(\{\bullet\}) = M$ in a canonical way [Mel09, Section 6.2]. A fact that encapsulates the idea of the second item above – but which, strictly speaking, also depends on the third one – is that the result of this transport is precisely (M, μ, η).

Handwaving argument. Although the intuition of the proof is simple, its execution involves a significant amount of bureaucracy; in particular, it manipulates various canonical isomorphisms given by Mac Lane's coherence theorem for symmetric monoidal categories (cf. §4.6.1). For this reason, we will only illustrate the idea here in the concrete case of the cartesian category of sets; *the full proof can be found in Section 4.6.2.*

For $(\mathcal{C}, \otimes, \mathbf{I}) = (\mathsf{Set}, \times, \{*\})$, we can reformulate the data given in the statement as a monoid M with a family of elements $(m_c)_{c\in\Gamma} \in M^\Gamma$, that can be identified with functions $m_c : \{*\} \to M$ for $c \in \Gamma$. The functor that we build then maps an object R – a finite set of registers – to the set M^R. The action on morphisms is best illustrated through an

example: $(t : z \mapsto axby) \in [\{x,y\} \to_{\mathcal{SR}(\{a,b\})} \{z\}]$ (where $\mathsf{in}_1/\mathsf{in}_2$ are omitted) becomes the map $(u_x, u_y) \in M^{\{x,y\}} \mapsto m_a u_x m_b u_y \in M \cong M^{\{z\}}$. Note that when we apply this construction to $M = \Gamma^*$ with $m_c = c$, we recover the functor $(-)^\dagger : \mathcal{SR}(\Gamma) \to \mathsf{Set}$ from Proposition 4.2.12. □

Remark 4.6.2. Informally speaking, we think of $\mathcal{SR}(\Gamma)$ as *the affine symmetric monoidal category freely generated by an internalization of the free monoid* Γ^*. To truly express this, one would need to add to Theorem 4.6.1 the uniqueness of F up to natural isomorphism. This would imply, among other things, that the morphisms of $\mathcal{SR}$ are inductively given by

- the identities id
- the compositions
- the structural morphisms associated to the tensor product
- the unique morphism $\{\bullet\} \to \varnothing$
- canonical morphisms $\widehat{c} : \top \to \{\bullet\}$ for every individual letter $c \in \Gamma$
- a canonical morphism $\eta : \top \to \{\bullet\}$ corresponding to the empty word
- a multiplication morphism $\mu : \{\bullet\} \otimes \{\bullet\} \to \{\bullet\}$ corresponding to string concatenation.

However, we do not prove this inductive presentation, nor the uniqueness property, since they are not necessary for our purposes.

Since the monoid object structure of internal homsets (Proposition 4.1.9) has a somewhat explicit description, Theorem 4.6.1 admits a specialized and simplified formulation for affine symmetric monoidal closed categories. For our applications (in particular the general preservation theorem of Section 4.7), it will be useful to give a version that also applies in the quasi-affine case.

Corollary 4.6.3. *Let $(\mathcal{C}, \otimes, \mathbf{I})$ be a* quasi-affine *symmetric monoidal* closed *category and A be an object in $\mathcal{C}$. For any family $(f_c)_{c\in\Gamma} \in \mathsf{Hom}_{\mathcal{C}}(A,A)^\Gamma$ of endomorphisms, there exists a strong monoidal functor $F : \mathcal{SR}(\Gamma) \to \mathcal{C}$ such that*

$$F(\varnothing) = \mathbf{I} \qquad F(\{\bullet\}) = (A \multimap A) \,\&\, \mathbf{I} \qquad \forall w \in \Gamma^*,\ F(\widehat{w}) = \left\langle \Lambda'\left(f_{w[1]} \circ \cdots \circ f_{w[n]}\right), \mathrm{id}_{\mathbf{I}} \right\rangle$$

where we use the notations $\widehat{(-)}$ from Theorem 4.6.1 and $\Lambda'(-)$ from Proposition 4.1.9.

Proof. Proposition 4.1.9 gives a canonical internal monoid structure to $A \multimap A$, which by Proposition 4.1.10 can be lifted to a monoid object $((A \multimap A) \,\&\, \mathbf{I}, \mu, \eta)$. For $f \in \mathsf{Hom}_{\mathcal{C}}(A,A)$, let $\Lambda''(f) = \langle \Lambda'(f), \mathrm{id}_{\mathbf{I}} \rangle : \mathbf{I} \to (A \multimap A) \,\&\, \mathbf{I}$.

We apply Theorem 4.6.1 to the slice category $\mathcal{C}/\mathbf{I}$ – which satisfies the affineness assumption – with the internal monoid $\pi_2 : (A \multimap A) \,\&\, \mathbf{I} \to \mathbf{I}$ (cf. Proposition 4.1.10 again) and the family $(\Lambda''(f_c))_{c\in\Gamma}$ (each $\Lambda''(f)$ is a morphism in the slice category from its unit $\mathrm{id}_{\mathbf{I}}$ to this π_2 since $\pi_2 \circ \Lambda''(f) = \mathrm{id}_{\mathbf{I}}$). We compose the resulting functor $\mathcal{SR} \to \mathcal{C}/\mathbf{I}$ with the forgetful functor $\mathsf{dom} : \mathcal{C}/\mathbf{I} \to \mathcal{C}$ to get $F : \mathcal{SR} \to \mathcal{C}$ such that $F(\varnothing) = \mathbf{I}$ and $F(\{\bullet\}) = (A \multimap A) \,\&\, \mathbf{I}$. As a composition of strong monoidal functors, F is also strong monoidal. This takes care of all but one of the corollary's conclusions.

For the remaining one, we need to do some preliminary work. First, let us recall two commuting diagrams below. The left one comes from the naturality of the family of (iso)morphisms $m_{R,S} : F(R) \otimes F(S) \to F(R+S)$ that make F a (strong) monoidal functor, while the right one is among the coherence conditions in Definition 4.1.2.

$$\begin{array}{ccc} \mathbf{I} \otimes \mathbf{I} & \xrightarrow{m_{\varnothing,\varnothing}} & F(\varnothing + \varnothing) \\ {\scriptstyle F(\widehat{u})\otimes F(\widehat{v})}\downarrow & & \downarrow{\scriptstyle F(\widehat{u}\otimes\widehat{v})} \\ F(\{\bullet\}) \otimes F(\{\bullet\}) & \xrightarrow{m_{\{\bullet\},\{\bullet\}}} & F(\{\bullet\} + \{\bullet\}) \end{array} \qquad \begin{array}{ccc} \mathbf{I} \otimes F(\varnothing) & \xrightarrow{\lambda_{F(\varnothing)}} & F(\varnothing) \\ {\scriptstyle m_0\otimes \mathrm{id}_{F(\varnothing)}}\downarrow & & \uparrow{\scriptstyle F(\lambda_\varnothing)} \\ F(\varnothing) \otimes F(\varnothing) & \xrightarrow{m_{\varnothing,\varnothing}} & F(\varnothing + \varnothing) \end{array}$$

Here, $m_0 : \mathbf{I} \to F(\varnothing)$ is also part of the strong monoidal structure of F and $u, v \in \Gamma^*$. The construction of Theorem 4.6.1 gives us $m_0 = \mathrm{id}_{\mathbf{I}}$ and $m_{\{\bullet\},\{\bullet\}} = \mathrm{id}_{F(\{\bullet\}+\{\bullet\})}$; furthermore, we have $\varnothing + \varnothing = \varnothing$ and $\lambda_\varnothing = \mathrm{id}_\varnothing$ in $\mathcal{SR}$. Thus, in the end, we can combine the two equalities expressed by the above diagrams and simplify them to get

$$\forall u, v \in \Gamma^*, \quad F(\widehat{u}) \otimes F(\widehat{v}) = F(\widehat{u} \otimes \widehat{v}) \circ \lambda_{\mathbf{I}}$$

Theorem 4.6.1 also guarantees that $F(\mu_\bullet) = \mu$, $F(\eta_\bullet) = \eta$ and $F(\widehat{c}) = \Lambda''(f_c)$ for all $c \in \Gamma$. At the same time, by combining Propositions 4.1.9 and 4.1.10, one can derive

$$\eta = \Lambda''(\mathrm{id}_A) \qquad \mu \circ (\Lambda''(f) \otimes \Lambda''(g)) \circ \lambda_{\mathbf{I}} = \Lambda''(f \circ g) \quad \text{for } f, g \in \mathsf{Hom}_{\mathcal{C}}(A, A)$$

We use all the above in a proof by induction of the desired conclusion.

- The base case is $F(\widehat{\varepsilon}) = F(\eta_\bullet) = \eta = \Lambda''(\mathrm{id}_A)$ (indeed, $\widehat{\varepsilon} = \eta_\bullet = (\bullet \mapsto \varepsilon)$ by definition).
- For the inductive case, write any word of length $n+1$ in Γ^* as wc for $w \in \Gamma^*$ with $|w| = n$ and $c \in \Gamma$, and suppose that by induction hypothesis $F(\widehat{w}) = \Lambda''(w)$. A direct computation of register transitions suffices to check that $\widehat{wc} = \mu_\bullet \circ (\widehat{w} \otimes \widehat{c})$. Then

$$\begin{aligned} F(\widehat{wc}) &= F(\mu_\bullet) \circ F(\widehat{w} \otimes \widehat{c}) = \mu \circ (F(\widehat{w}) \otimes F(\widehat{c})) \circ \lambda_{\mathbf{I}} \\ &= \mu \circ \left(\Lambda''\left(f_{w[n]} \circ \cdots \circ f_{w[1]}\right) \otimes \Lambda''(f_c)\right) \circ \lambda_{\mathbf{I}} = \Lambda''\left(f_{w[1]} \circ \cdots \circ f_{w[n]} \circ f_c\right) \end{aligned}$$ □

4.6.1. **Reminders on coherence for symmetric monoidal categories.** We now recall an important tool for the proof of Theorem 4.6.1. Let us fix a an symmetric monoidal category $(\mathcal{C}, \otimes, \mathbf{I})$ with an internal monoid (M, μ, η). We call the functors built from the monoidal structure on $\mathcal{C}$ "tensorial functors" since the monoidal product $\otimes$ is sometimes called a tensor product. They are more precisely defined as follows.

Definition 4.6.4. The *tensorial expressions* over a finite indexing set I are freely inductively generated as follows:

- $\mathbf{I}$ is a tensorial expression over $\varnothing$;
- i is a tensorial expression over $\{i\}$;
- if e, e' are tensorial expressions over I and I' respectively, with $I \cap I' = \varnothing$, then $(e \otimes e')$ is a tensorial expression over $I \cup I'$.

In other words, a tensorial expression over I is a binary tree whose leaves are labeled either by $\mathbf{I}$ or by $i \in I$, such that each element i appears exactly once.

A tensorial expression e over I induces a functor $F_e : \mathcal{C}^I \to \mathcal{C}$ in an obvious way. The functors thus obtained are called *tensorial functors*. A tensorial functor $F_e : \mathcal{C}^I \to \mathcal{C}$ is *ordered* when I is endowed with the total order that corresponds to the infix order on the I-labeled leaves of the expression e.

Of course, the basic intuition is that over a given totally ordered indexing set, two ordered tensorial functors express "the same thing" up to inessential bracketing, while two unordered tensorial functors should be morally the same "up to permutation". This is expressed formally as a natural isomorphism between functors, but *Mac Lane's coherence theorem* gives us something stronger: the natural isomorphisms can be chosen *canonically*.

Theorem 4.6.5 (Coherence for SMCs [Mac98, §XI.1 (Theorem 1)])**.** *There exists a map that sends each triple (I, e, e'), where e, e' are tensorial expressions over the finite set I, to a natural isomorphism from the tensorial functor F_e to $F_{e'}$, which we call a* canonical isomorphism*, such that:*

- *identities, associators, unitors and symmetries are canonical isomorphisms;*
- *composing the image of (I, e, e') by this map with the image of (I, e', e'') yields the image of (I, e, e'') – in other words, canonical isomorphisms are closed under composition;*
- *canonical isomorphisms are also closed under monoidal product.*

If $F, G : \mathcal{C}^I \to \mathcal{C}$ are *ordered* tensorial functors for the same total order on I, the construction of the canonical isomorphism between F and G works in any monoidal category, not necessarily symmetric. Indeed, general monoidal categories enjoy a coherence theorem of their own [Mac98, §VII.2], which involves its own canonical isomorphisms; thanks to the uniqueness clauses in the coherence theorems with and without symmetry, one can check that the two notions of canonical isomorphism coincide for ordered tensorial functors. The ordered case is important for us because even though our monoidal category is symmetric, the internal monoid (M, μ, η) that we are given need not be *commutative*. What the axioms for monoid objects *do* state, however, is a suitable form of associativity, a consequence of which is that n-ary products are somehow "independent of bracketing". Formally speaking:

Definition 4.6.6. To any tensorial expression e, we inductively associate a $\mathcal{C}$-morphism $\mu^{(e)} : F_e((M)_{i\in I}) \to M$ (meant to represent "e-fold monoid multiplication"):

$$\mu^{(\mathbf{I})} = \eta \qquad \mu^{(i)} = \mathrm{id} \text{ for } I = \{i\} \qquad \mu^{(e\otimes e')} = \mu \circ \left(\mu^{(e)} \otimes \mu^{(e')}\right)$$

Theorem 4.6.7 (General associativity law [Mac98, §VII.3])**.** *Let I be a totally ordered finite set, $F_e, F_{e'} : \mathcal{C}^I \to \mathcal{C}$ be two* ordered *tensorial functors and $\Xi : F_e \Rightarrow F_{e'}$ be their canonical natural isomorphism. Then $\mu^{(e)} = \mu^{(e')} \circ \Xi_{\vec{M}}$ where $\vec{M} = (M)_{i\in I}$.*

We find it convenient to work directly with tensorial functors in the rest of this section, leaving the tensorial expressions that define them implicit. Therefore, we write $\mu^{(F)}$ instead of $\mu^{(e)}$ when $F = F_e$; strictly speaking, two expressions could define the same functor for accidental reasons, but for our purposes, the right choice of e can always be inferred from the context. Similarly, we shall speak of *the* canonical isomorphism between two tensorial functors $\mathcal{C}^I \to \mathcal{C}$. In accordance with Section 4.1.1, we write

$$\bigotimes_{i=1}^{n} Y_i = (\dots(Y_1 \otimes Y_2) \otimes \dots) \otimes Y_n \qquad M^{\otimes n} = \bigotimes_{i=1}^{n} M$$

and this will be used to define ordered tensorial functors. Furthermore, for each finite set I, we fix an arbitrary choice of tensorial functor $\bigotimes_{i\in I}(-) : \mathcal{C}^I \to \mathcal{C}$, and denote by $\bigotimes_{i\in I} Y_i$ the image of $(Y_i)_{i\in I}$ by this functor.

4.6.2. **Proof of Theorem 4.6.1.** After these general preliminaries, let us focus on the specific study of the symmetric monoidal category $\mathcal{SR}(\Gamma)$ for a fixed finite alphabet Γ.

Definition 4.6.8. For $t \in [R \to_{\mathcal{SR}} R']$ – recall that this implies $t : R' \to (\Gamma + R)^*$, let $\partial(t) \subseteq R$ be the set of register variables that do not occur in any $t(r')$ for $r' \in R'$.

Definition 4.6.9. Given $w \in (\Gamma + R)^*$, we write

$$\bigotimes_{r\leftarrow\!\!\rho w} Y_r = \bigotimes_{i=1}^{|w|} \begin{cases} \mathbf{I} & \text{when } w[i] \in \mathsf{in}_1(\Gamma) \\ Y_r & \text{when } w[i] = \mathsf{in}_2(r) \text{ for } r \in R \end{cases} \qquad \text{and} \qquad M^{\leftarrow\!\!\rho w} = \bigotimes_{r\leftarrow\!\!\rho w} M$$

Lemma 4.6.10. *For $t \in [R \to_{\mathcal{SR}} R']$, we have a canonical isomorphism*

$$\bigotimes_{r\in R} Y_r \quad \cong \quad \bigotimes_{r\in\partial(t)} Y_r \otimes \bigotimes_{r'\in R'} \bigotimes_{r\leftarrow\!\!\rho t(r')} Y_r$$

Proof. To apply Mac Lane's coherence theorem, we just have to check that the right-hand side defines a tensorial functor with indexing set R. This amounts to the equality

$$R = \partial(t) \cup \bigcup_{r'\in R'} \{r \mid \exists i.\, t(r')[i] = \mathsf{in}_2(r)\}$$

where, to ensure that no index in R is repeated, the union must be *disjoint* and the letters of $t(r')$ that are in $\mathsf{in}_2(R)$ must all be distinct. Those conditions are consequences of copylessness, while the equality itself is essentially the definition of $\partial(t)$. □

We are now in a position to build the functor promised in Theorem 4.6.1. Following the statement of this theorem, we fix a family of morphisms $(m_c) \in \mathsf{Hom}_{\mathcal{C}}(\mathbf{I}, M)^{\Gamma}$, and assume that $(\mathcal{C}, \otimes, \mathbf{I})$ is affine. Thus, we may write $\langle\rangle_A : A \to \mathbf{I}$ for the terminal morphism from A, omiting the subscript A when it can be inferred from the context.

Definition 4.6.11. We define a map on objects $F : R \in \mathsf{Obj}(\mathcal{SR}) \mapsto M^{\otimes R} \in \mathsf{Obj}(\mathcal{C})$.

As for morphisms, given a register transition $t \in [R \to_{\mathcal{SR}} R']$, we set $F(t)$ to be

$$F(R) = M^{\otimes R} \xrightarrow{\sim} M^{\otimes \partial(t)} \otimes \bigotimes_{r' \in R'} \bigotimes_{i=1}^{|t(r')|} X_{r',i} \xrightarrow{\langle\rangle \otimes \widetilde{F}(t)} \mathbf{I} \otimes M^{\otimes R'} \xrightarrow{\sim} M^{\otimes R'} = F'(R)$$

where the left arrow instantiates the canonical isomorphism of Lemma 4.6.10, with

$$X_{r',i} = \begin{cases} \mathbf{I} & \text{when } t(r')[i] \in \mathsf{in}_1(\Gamma) \\ M & \text{otherwise, i.e. } t(r')[i] \in \mathsf{in}_2(R) \end{cases} \qquad \text{so that} \qquad \bigotimes_{i=1}^{|t(r')|} X_{r',i} = M^{\leftarrow \rho t(r')}$$

and in the middle arrow, $\langle\rangle : M^{\otimes \partial(t)} \to \mathbf{I}$ is the terminal morphism and

$$\widetilde{F}(t) = \bigotimes_{r' \in R'} \left(\widetilde{F}(t)_{r'} : \bigotimes_{i=1}^{|t(r')|} X_{r',i} \xrightarrow{\otimes_i f_{r',i}} \bigotimes_{i=1}^{|t(r')|} M = M^{\otimes |t(r')|} \xrightarrow{\mu^{(|t(r')|)}} M \right)$$

where for $i \in \{1, \dots, |t(r')|\}$, we pick

$$f_{r',i} = \begin{cases} m_c & \text{when } t(r')[i] = \mathsf{in}_1(c) \text{ for } c \in \Gamma \\ \mathrm{id}_M & \text{otherwise, i.e. } t(r')[i] \in \mathsf{in}_2(R) \end{cases}$$

(recall that $m_c : \mathbf{I} \to M$ is the prescribed functorial image, by the F that we are defining, of the register transition $\widehat{c} \in [\varnothing \to_{\mathcal{SR}} \{\bullet\}]$).

The tedious part in proving Theorem 4.6.1 is checking that the above definition works. This fills the several pages remaining in this section.

Proposition 4.6.12. *The operation F introduced in Definition 4.6.11 is a functor.*

Proof. Let $t \in [R \to_{\mathcal{SR}} R']$ and $t' \in [R' \to_{\mathcal{SR}} R'']$; we want to reason on $F(t') \circ F(t)$ to show that it is equal to $F(t' \circ t)$. Beware: we write $t' \circ t$ for composition of (copyless) register transitions in the category $\mathcal{SR}$, and will employ the notation $t(r')$ for set-theoretic application $(t : R' \to (\Gamma + R)^*)$, but we do *not* have $(t' \circ t)(r') = t'(t(r'))$ – indeed, the two sides of the equality are not even well-defined for $r' \in R'$! Since $t' \circ t$ is in $[R \to_{\mathcal{SR}} R'']$, it is a set-theoretic map $R'' \to (\Gamma + R)^*$.

To prove this, we first reduce our goal to Lemma 4.6.13, and then prove that lemma. The reduction is given by the commutative diagram of Figure 4.6.1 (with N to be defined later). Indeed, while the morphism on the top is $F(t') \circ F(t)$, the one defined by the three other sides of the outermost square is

$$M^{\otimes R} \xrightarrow{\sim} M^{\otimes \partial(t' \circ t)} \otimes \bigotimes_{r'' \in R''} M^{\leftarrow \rho(t' \circ t)(r'')} \xrightarrow{\langle\rangle \otimes \widetilde{F}(t' \circ t)} \mathbf{I} \otimes M^{\otimes R''} \xrightarrow{\sim} M^{\otimes R''}$$

thanks to the closure of canonical isomorphisms of tensorial functors under composition. Since these canonical isomorphisms are also unique, we can equate that with the expression for $F(t' \circ t)$ given in Definition 4.6.11, hence $F(t') \circ F(t) = F(t' \circ t)$.

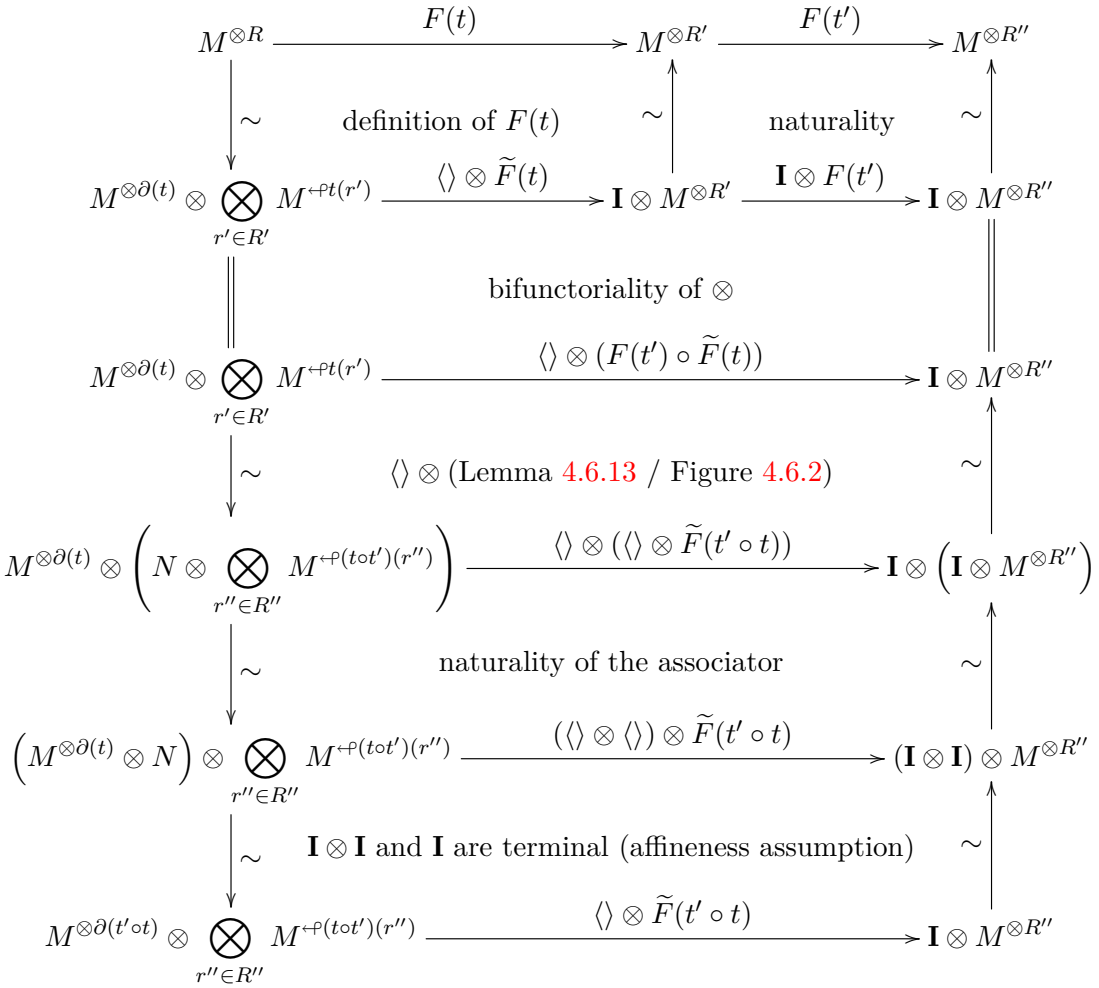


FIGURE 4.6.1. The commutativity of the outer square of this diagram establishes Proposition 4.6.12. The text in the inner squares explains why they commute; the proof text for Proposition 4.6.12 defines N and gives further justifications (in particular for the existence of the canonical isomorphisms denoted by $\xrightarrow{\sim}$).

The main justifications that are missing from Figure 4.6.1 are the commutation of one square treated in Lemma 4.6.13, and the existence of the various canonical isomorphisms involved. The ones in the right column are just unitors, so let us focus on the left column.

- The top left isomorphism is an instance of Lemma 4.6.10.
- The next one (skipping the equality) is provided by Lemma 4.6.13.
- Then the penultimate one is just the inverse of an associator (see §4.1.2), of the form

$$\alpha^{-1}_{A,B,C} \;:\; A \otimes (B \otimes C) \;\to\; (A \otimes B) \otimes C$$

- Finally, the last one reduces to $M^{\otimes\partial(t)} \otimes N \cong M^{\otimes\partial(t'\circ t)}$.

To prove the latter, we must first clarify it by defining N consistently with Lemma 4.6.13:

$$N = \bigotimes_{r'\in\partial(t')} M^{\leftarrow\rho t(r')}$$

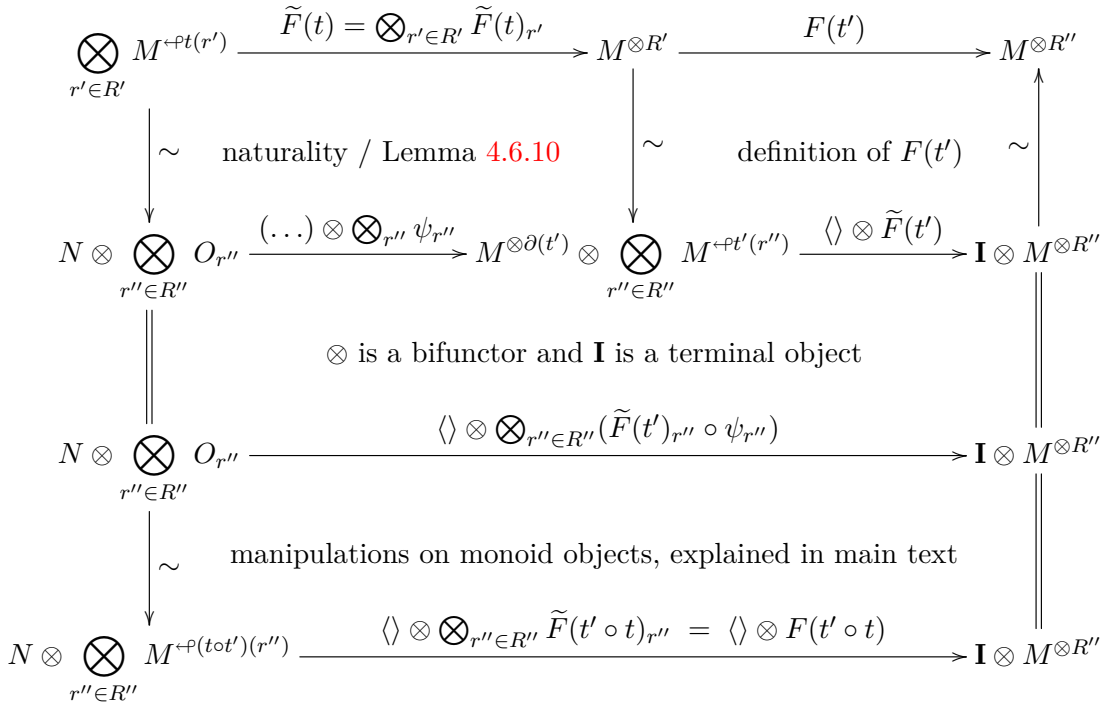


FIGURE 4.6.2. Lemma 4.6.13 defines N, $O_{r''}$, and $\psi_{r''}$, and proves that this diagram commutes.

We then see that the canonical isomorphism that we are looking for is the instantiation to the constant family $(M)_{r \in \partial(t' \circ t)}$ of the following natural isomorphism in Y_r for $r \in \partial(t' \circ t)$:

$$\bigotimes_{r \in \partial(t)} Y_r \otimes \bigotimes_{r' \in \partial(t')} \bigotimes_{r \hookleftarrow t(r')} Y_r \quad \cong \quad \bigotimes_{r \in \partial(t' \circ t)} Y_r$$

This is a consequence of an elementary combinatorial fact:

$$\partial(t) \cup \bigcup_{r' \in \partial(t')} \{r \mid \exists i \in \{1, \dots, |t(r')|\}.\ t(r')[i] = \mathsf{in}_2(r)\} \quad = \quad \partial(t' \circ t)$$

Informally speaking, this means that a register is thrown away by $t' \circ t$ if and only if is either thrown away by t, or used by t to compute a value that is then discarded by t'. Furthermore, thanks to copylessness, the union is *disjoint* and for each r in the union over $r' \in \partial(t')$, there is a *single* (r', i) such that $t(r')[i] = \mathsf{in}_2(r)$; those properties are necessary to get the natural isomorphism. We leave a formal proof of this combinatorial identity to the reader.

This being done, all that remains to conclude our proof is to show Lemma 4.6.13 below. □

Lemma 4.6.13. *The diagram of Figure 4.6.2 commutes, with the following definitions:*

$$N = \bigotimes_{r' \in \partial(t')} M^{\hookleftarrow t(r')} \qquad O_{r''} = \bigotimes_{r' \hookleftarrow t'(r'')} M^{\hookleftarrow t(r')} \qquad \psi_{r''} = \bigotimes_{r' \hookleftarrow t'(r'')} \widetilde{F}(t)_{r'}$$

PROOF. In addition to the information already given in Figure 4.6.2, there are two things to justify about this figure: the top-left naturality square, and the commutativity

of the bottom square. Let us tackle the former. Recall that Lemma 4.6.10 gives us the canonical isomorphism

$$\bigotimes_{r'\in R'} Y_{r'} \quad\cong\quad \bigotimes_{r'\in\partial(t')} Y_{r'} \otimes \bigotimes_{r''\in R''} \bigotimes_{r'\leftarrow\!\rho t'(r'')} Y_{r'}$$

which is natural in $Y_{r'}$ for $r'\in R'$. By instantiating with $Y_{r'} = M^{\leftarrow\!\rho t(r')}$, we get

$$\bigotimes_{r'\in R'} M^{\leftarrow\!\rho t(r')} \quad\cong\quad \bigotimes_{r'\in\partial(t')} M^{\leftarrow\!\rho t(r')} \otimes \bigotimes_{r''\in R''} \bigotimes_{r'\leftarrow\!\rho t'(r'')} M^{\leftarrow\!\rho t(r')} \quad=\quad N\otimes \bigotimes_{r''\in R''} O_{r''}$$

by definition of N and $O_{r''}$. Hence the vertical canonical isomorphism at the top left of Figure 4.6.2; as for the top middle, it is the instantiation of the same natural isomorphism with $Y_{r'} = M$, as already observed in Definition 4.6.11. To make the top-left square a naturality square, we should then have

$$(\dots)\otimes \bigotimes_{r''\in R''} \psi_{r''} \quad=\quad \bigotimes_{r'\in\partial(t')} \widetilde{F}(t)_{r'} \otimes \bigotimes_{r''\in R''} \bigotimes_{r'\leftarrow\!\rho t'(r'')} \widetilde{F}(t)_{r'}$$

which is indeed the case with our definition of $\psi_{r''}$.

Our next and final task is the commutativity of the bottom square of Figure 4.6.2. Thanks to the bifunctoriality of $\otimes$, it reduces to a simpler commutative diagram:

$$\begin{array}{ccc} O_{r''} = \bigotimes_{r'\leftarrow\!\rho t'(r'')} M^{\leftarrow\!\rho t(r')} & \xrightarrow{\widetilde{F}(t')_{r''}\circ\psi_{r''}} & M \qquad \text{for } r''\in R'' \\ \sim\downarrow & \nearrow_{\widetilde{F}(t'\circ t)_{r''}} & \\ M^{\leftarrow\!\rho(t\circ t')(r'')} & & \end{array}$$

Let $r''\in R''$. To show that this indeed commutes, we start by writing out $\widetilde{F}(t')_{r''}\circ\psi_{r''}$ as

$$\bigotimes_{r'\leftarrow\!\rho t'(r'')} M^{\leftarrow\!\rho t(r')} \xrightarrow{\bigotimes_{r'}\widetilde{F}(t)_{r'}} M^{\leftarrow\!\rho t'(r'')} = \bigotimes_{j=1}^{|t'(r'')|} X'_{r'',j} \xrightarrow{\bigotimes_j f'_{r'',j}} M^{\otimes|t'(r'')|} \xrightarrow{\mu^{(|t'(r'')|)}} M$$

where, analogously to the $X_{r',i}$ and $f_{r',i}$ involved in the definition of $\widetilde{F}(t)_{r'}$, we take

$$(X'_{r'',j},\ (f'_{r'',j} : X'_{r'',j}\to M)) = \begin{cases} (\mathbf{I},\quad m_c) & \text{when } t'(r'')[j] = \mathsf{in}_1(c) \text{ for } c\in\Gamma \\ (M,\ \mathrm{id}_M) & \text{otherwise, i.e. } t'(r'')[j]\in\mathsf{in}_2(R') \end{cases}$$

The leftmost arrow in the above sequence is how we defined $\psi_{r''}$ in the lemma statement, while the composition of the two others equals $\widetilde{F}(t')_{r''}$ by definition.

To manipulate this, let us introduce a new notation:

$$\bigotimes_{w}^{\texttt{foldMap}} f \quad=\quad \bigotimes_{i=1}^{|w|} f(w_i) \qquad \text{for } w = w_1\dots w_n \text{ and } f \text{ a function}$$

Note that in general, this may lead to monoidal products with repeated factors: there is no a priori guarantee that this defines a tensorial functor. The function f will often be expressed as a copairing using the following notation:

$$f : \begin{cases} \mathsf{in}_1(y) \mapsto f_1(y) \\ \mathsf{in}_2(z) \mapsto f_2(z) \end{cases} \quad\Longleftrightarrow\quad f : x\in Y+Z \mapsto \begin{cases} f_1(y) & \text{when } x = \mathsf{in}_1(y) \text{ for } y\in Y \\ f_2(z) & \text{when } x = \mathsf{in}_2(z) \text{ for } z\in Z \end{cases}$$

In the case that we are interested in right now, we have by functoriality of $\otimes$:

$$\widetilde{F}(t')_{r''} \circ \psi_{r''} \;=\; \mu^{(|t'(r'')|)} \circ \bigotimes_{t'(r'')}^{\texttt{foldMap}} \begin{cases} \mathsf{in}_1(c) \;\mapsto m_c \\ \mathsf{in}_2(r') \mapsto \widetilde{F}(t)_{r'} = \mu^{(|t(r')|)} \circ \bigotimes_{t(r')}^{\texttt{foldMap}} \begin{cases} \mathsf{in}_1(a) \mapsto m_a \\ \mathsf{in}_2(r) \mapsto \mathrm{id}_M \end{cases} \end{cases}$$

To simplify this, we introduce the following tensorial functor:

$$L_{r''}((Y_p)_{p\in P_{r''}}) \quad = \quad \bigotimes_{j=1}^{|t'(r'')|} \begin{cases} Y_{j,1} & \text{when } t'(r'')[j] \in \mathsf{in}_1(\Gamma) \\ \bigotimes_{i=1}^{|t(r')|} Y_{j,i} & \text{when } t'(r'')[j] = \mathsf{in}_2(r') \end{cases}$$

$$\text{where} \qquad P_{r''} \quad = \quad \sum_{j=1}^{|t'(r'')|} \begin{cases} \{1\} & \text{when } t'(r'')[j] \in \mathsf{in}_1(\Gamma) \\ \{1,\dots,|t(r')|\} & \text{when } t'(r'')[j] = \mathsf{in}_2(r') \end{cases}$$

(using the dependent sum operation, cf. §2.1.1, so that $P_{r''} \subset \mathbb{N}^2$). There is a unique total order on $P_{r''}$ that makes $L_{r''}$ into an *ordered* tensorial functor: the *lexicographical order* inherited from $\mathbb{N}^2$. Thus, we may meaningfully speak of $\mu^{(L_{r''})} : L_{r''}((M)_{p\in P_{r''}}) \to M$, the $L_{r''}$-ary monoid multiplication, whose inductive definition leads to:

$$\mu^{(L_{r''})} \quad = \quad \mu^{(|t'(r'')|)} \circ \bigotimes_{t'(r'')}^{\texttt{foldMap}} \begin{cases} \mathsf{in}_1(c) \;\mapsto \mu^{(1)} = \mathrm{id}_M \\ \mathsf{in}_2(r') \mapsto \mu^{(|t(r')|)} \end{cases}$$

Now that this is defined, we can state the following equation, that directly follows by functoriality from the previous expressions of $\widetilde{F}(t')_{r''} \circ \psi_{r''}$ and of $\mu^{(L_{r''})}$:

$$\widetilde{F}(t')_{r''} \circ \psi_{r''} \quad = \quad \mu^{(L_{r''})} \circ \bigotimes_{t'(r'')}^{\texttt{foldMap}} \begin{cases} \mathsf{in}_1(c) \;\mapsto \quad m_c \\ \mathsf{in}_2(r') \mapsto \bigotimes_{t(r')}^{\texttt{foldMap}} \begin{cases} \mathsf{in}_1(a) \mapsto m_a \\ \mathsf{in}_2(r) \mapsto \mathrm{id}_M \end{cases} \end{cases}$$

For the next step, recall that by Definition 4.2.11, $(t' \circ t)(r'') = t^\star(t'(r''))$ (which is *not* $t'(t(r''))$, as previously emphasized, since this '$\circ$' is composition in $\mathcal{SR}(\Gamma)$), where $t^\star$ is the morphism of free monoids such that $t^\dagger(\mathsf{in}_1(c)) = \mathsf{in}_1(c)$ and $t^\dagger(\mathsf{in}_2(r')) = t(r')$. Thus, the j-th term of the dependent sum above is equal to $\{1,\dots,|t^\dagger(t'(r'')[j])|\}$, and from the morphism property

$$(t' \circ t)(r'') = t^\star(t'(r'')) = t^\star(t'(r'')[1]) \cdot \ldots \cdot t^\star(t'(r'')[|t'(r'')|])$$

we obtain an bijection $\xi_{r''} : P_{r''} \xrightarrow{\sim} \{1,\dots,(t' \circ t)(r'')\}$ such that for all $(j,i) \in P_{r''}$,

$$(t' \circ t)(r'')[\xi(j,i)] = t^\star(t'(r'')[j])[i] = \begin{cases} \mathsf{in}_1(c) & \text{when } t'(r'')[j] = \mathsf{in}_1(c) \\ t(r')[i] & \text{when } t'(r'')[j] = \mathsf{in}_2(r') \end{cases}$$

Thanks to this, we have

$$\widetilde{F}(t')_{r''} \circ \psi_{r''} \;=\; \mu^{(L_{r''})} \circ L_{r''} \left(\left(\begin{cases} m_c & \text{when } (t' \circ t)(r'')[\xi(p)] = \mathsf{in}_1(c) \text{ for } c \in \Gamma \\ \mathrm{id}_M & \text{otherwise, i.e. } (t' \circ t)(r'')[\xi(p)] \in \mathsf{in}_2(R) \end{cases} \right)_{p\in P_{r''}} \right)$$

It is not hard to see that the bijection ξ is *monotone* (taking the lexicographical order on $P_{r''}$ as we did earlier). Therefore, the natural isomorphism

$$\Xi_{(Y_p)_{p\in P_{r''}}} : L\left((Y_p)_{p\in P_{r''}}\right) \xrightarrow{\sim} \bigotimes_{k=1}^{|(t'\circ t)(r'')|} Y_{\xi^{-1}(k)}$$

induced by ξ is a *canonical isomorphism of ordered tensorial functors* (i.e. it only uses associators and unitors, not symmetries). Thus, we may apply the general associativity law for internal monoids (Theorem 4.6.7): $\mu^{(L_{r''})} = \mu^{(|(t'\circ t)(r'')|)} \circ \Xi$ (we omit the subscript of Ξ for convenience). By naturality of Ξ, we then have

$$\widetilde{F}(t')_{r''} \circ \psi_{r''} \quad = \quad \mu^{(|(t'\circ t)(r'')|)} \circ \left(\bigotimes_{(t'\circ t)(r'')}^{\texttt{foldMap}} \begin{cases} \mathsf{in}_1(c) & \mapsto m_c \\ \mathsf{in}_2(r) & \mapsto \mathrm{id}_M \end{cases} \right) \circ \Xi$$

By definition, this is equal to $\widetilde{F}(t' \circ t) \circ \Xi$, which is what we needed to conclude the proof. □

Proposition 4.6.14. *The functor F is strong monoidal.*

Proof. From the definition, we immediately get

$$F(R + R') = M^{\otimes(R+R')} \cong M^{\otimes R} \otimes M^{\otimes R'} = F(R) \otimes F(R')$$

as an instance of a canonical isomorphism

$$\bigotimes_{x \in R+R'} Y_x \quad \cong \quad \bigotimes_{r \in R} Y_{\mathsf{in}_1(r)} \otimes \bigotimes_{r' \in R'} Y_{\mathsf{in}_2(r')}$$

which is natural in Y_x for $x \in R + R'$. One can then verify that this family of isomorphisms $F(R + R') \cong F(R) \otimes F(R')$ is natural in R and R'. □

To finish proving Theorem 4.6.1, it suffices to carry out some short explicit computations to check that this functor F satisfies the claimed equalities. We leave this to the reader.

4.7. On closure under precomposition by regular functions

Our next categorical automata result is our first application of monoidal closure:

Theorem 4.7.1. *Let $\mathfrak{C}$ be a string streaming setting with output set X. Suppose that the underlying category $\mathcal{C}$ is* symmetric monoidal closed *and* quasi-affine*. Furthermore, let us assume that $\mathbb{T}_{\mathfrak{C}}$ is equal to the monoidal unit $\mathbf{I}$.*

Then for any $f : \Gamma^* \to X$ computed by some $\mathfrak{C}$-SST, and any regular $g : \Sigma^* \to \Gamma^*$, the function $f \circ g : \Sigma^* \to X$ is computed by some (stateful) $\mathfrak{C}$-SST. In other words, the class of functions defined by $\mathfrak{C}$-SSTs is closed under precomposition by regular functions.

Before proving the above theorem, let us check that it entails known preservation and composition properties.

Corollary 4.7.2. *Let $L \subseteq \Gamma^*$ be a regular language and $g : \Sigma^* \to \Gamma^*$ be a regular function. Then the language $g^{-1}(L) \subseteq \Sigma^*$ is regular.*

Proof. That L is regular is equivalent to its indicator function $\chi_L : \Gamma^* \to \{0, 1\}$ being computed by some single-state $\mathfrak{Finset}_2$-SST, see Example 4.2.5. The underlying category of finite sets is *cartesian closed*, and the monoidal structure given by a cartesian product is automatically symmetric and affine. According to Theorem 4.7.1, $\chi_L \circ g$ can therefore be computed by some $\mathfrak{Finset}_2$-SST. Observing that the category of finite sets has coproducts, and applying Lemma 4.3.12, we even have a *single-state* $\mathfrak{Finset}_2$-SST for $\chi_L \circ g$. Finally, the latter is none other than the indicator function of $g^{-1}(L)$. □

Corollary 4.7.3. *Let $f : \Gamma^* \to \Delta^*$ and $g : \Sigma^* \to \Gamma^*$ be regular functions. Then $f \circ g$ is also a regular function.*

Proof. This is just the application of Theorem 4.7.1 to $\mathfrak{SR}_{\oplus\&}$-SSTs. The functions that they can compute are exactly the regular functions (cf. §4.5.2). By Theorem 1.2.4 / Corollary 4.5.4, the underlying category $\mathcal{SR}_{\oplus\&}$ is symmetric monoidal closed. Finally, $\mathcal{SR}_{\oplus\&}$ is quasi-affine since it has all cartesian products by construction. □

Remark 4.7.4. As we shall see in Proposition 5.1.16, the closure under composition of regular functions also follows from our $\lambda\ell^{\oplus\&}$-calculus characterization (Theorem 1.2.3). However, note that the latter relies on some non-trivial syntactic analysis of $\lambda\ell^{\oplus\&}$-terms, as discussed at the start of Chapter 5. The argument we give in this section circumvents that difficulty. That said, it still shares some (non-syntactic) ingredients with our proof of Theorem 1.2.3, namely:

- the monoidal closure and quasi-affineness of $\mathcal{SR}_{\oplus\&}$;
- the fact that $\mathfrak{SR}_{\oplus\&}$-SSTs are no more expressive than $\mathfrak{SR}$-SSTs.

These results still require substantial developments – indeed, this composition property is quite non-trivial as mentioned in the introduction – but bypass the need of mentioning the $\lambda\ell^{\oplus\&}$-calculus. Beyond this simplification, the main advantage of our approach here is that we get a more general theorem, that applies to many streaming settings; in particular, the final output does not have to be a string.

We now come to the proof of this generalized preservation theorem, which uses the universal property from the previous section (more specifically, its Corollary 4.6.3).

Proof of Theorem 4.7.1. We give below a proof assuming that f is defined by some *single-state* $\mathfrak{C}$-SST (but beware: the $\mathfrak{C}$-SST computing $f \circ g$ will still be stateful!). The general case can be applied by considering the streaming setting $\mathfrak{C}_{\oplus\mathrm{const}}$-SST, as was briefly mentioned in Remark 4.3.13, and using the fact that single-state $\mathfrak{C}_{\oplus\mathrm{const}}$-SSTs, stateful $\mathfrak{C}_{\oplus\mathrm{const}}$-SSTs and stateful $\mathfrak{C}$-SSTs are equally expressive. We leave it to the reader to check that the symmetric monoidal structure and the cartesian products in $\mathcal{C}$ can be lifted to $\mathcal{C}_{\oplus\mathrm{const}}$, making this generalization possible.

Therefore, we may assume without loss of generality that f is computed by a single-state $\mathfrak{C}$-SST $T_f = (\{\bullet\}, A_f, \delta_f, i_f, o_f)$ where A_f is an object of $\mathcal{C}$ and

$$\delta_f : \Gamma \to \mathsf{Hom}_{\mathcal{C}}(A_f, A_f) \qquad i_f \in \mathsf{Hom}_{\mathcal{C}}(\top\!\!\top, A_f) \qquad o_f \in \mathsf{Hom}_{\mathcal{C}}(A_f, \perp\!\!\!\perp)$$

Let $T_g = (Q, q_0, R_g, \delta_g, i_g, o_g)$ be an usual copyless SST (i.e., a $\mathcal{SR}$-SST) computing the regular function g, where Q and R_g are finite sets and

$$q_0 \in Q \qquad \delta_g : \Sigma \times Q \to Q \times [R_g \to_{\mathcal{SR}} R_g]$$
$$i_g \in [\varnothing \to_{\mathcal{SR}} R_g] \qquad o_g : Q \to [R_g \to_{\mathcal{SR}} \{\bullet\}]$$

We will write $A = A_f$ and $R = R_g$ for short.

We want to build from this data a $\mathfrak{C}$-SST T defining $f \circ g$. Since $\mathcal{C}$ is quasi-affine and symmetric monoidal closed, we can apply Corollary 4.6.3 to the object $A \multimap A$ and to a family of morphisms $(\widetilde{\delta_f}(c))_{c\in\Gamma} \in \mathsf{Hom}_{\mathcal{C}}(A \multimap A,\ A \multimap A)^{\Gamma}$ that will be defined later. This gives us a functor $F_{\delta_f} : \mathcal{SR} \to \mathcal{C}$, enjoying various properties that will be progressively recalled, which is at the heart of our construction.

The set of states of our new $\mathfrak{C}$-SST T is Q, with initial state q_0, and its memory object is $F_{\delta_f}(R)$. The initialization morphism is defined as $i = F_{\delta_f}(i_g) \in \mathsf{Hom}_{\mathcal{C}}\left(\mathbf{I}, F_{\delta_f}(R)\right)$ – we use the assumption $\top\!\!\top = \mathbf{I}$, and the fact that $F_{\delta_f}(\varnothing) = \mathbf{I}$ (by Corollary 4.6.3). The transition function is

$$\begin{array}{rccccc} \delta : & \Sigma \times Q & \longrightarrow & Q & \times & \mathsf{Hom}_{\mathcal{C}}\left(F_{\delta_f}(R), F_{\delta_f}(R)\right) \\ & c\ ,\ q & \mapsto & \pi_1(\delta_g(c,q))\,, & & F_{\delta_f}(\pi_2(\delta_g(c,q))) \end{array}$$

Finally, using $j_f : F_{\delta_f}(\{\bullet\}) \to A$ to be defined later, we take as our new output function

$$o : q \in Q \mapsto o_f \circ j_f \circ F_{\delta_f}(o_g(q)) \in \mathsf{Hom}_{\mathcal{C}}\left(F_{\delta_f}(R), \perp\!\!\!\perp\right)$$

Let us now sketch the verification that this defines the intended function $f \circ g : \Sigma^* \to X$. In the process, we will fill the missing definitions to make everything work out.

Let $w = w_1 \dots w_n \in \Sigma^*$ be an input string. The sequence $q_0, \dots, q_n \in Q$ of states visited by both T_g and T when fed this input is obtained by $q_{i+1} = \pi_1(\delta_g(w_i, q_i))$ from the initial q_0. By definition of the output of T_g, we have:

$$\widehat{g(w)} \quad = \quad o_g(q_n) \circ \pi_2(\delta_g(w_n, q_{n-1})) \circ \dots \circ \pi_2(\delta_g(w_1, q_0)) \circ i_g$$

where $\widehat{g(w)} \in [\varnothing \to_{\mathcal{SR}(\Gamma)} \{\bullet\}]$ corresponds to $g(w) \in \Gamma^*$ (cf. Theorem 4.6.1) and the '$\circ$' denotes a composition of register transitions (i.e. of $\mathcal{SR}$-morphisms). Similarly, the output $T(w)$ of the $\mathfrak{C}$-SST T that we built on the input w is defined as

$$T(w) \quad = \quad (\!| o(q_n) \circ F_{\delta_f}(\pi_2(\delta_g(w_n, q_{n-1}))) \circ \dots \circ F_{\delta_f}(\pi_2(\delta_g(w_1, q_0))) \circ i |\!)$$

which, by unfolding the definitions of o and i, applying the functoriality of F_{δ_f} and comparing with the previous equality, one can simplify into

$$T(w) \quad = \quad (\!| o_f \circ j_f \circ F_{\delta_f}\left(\widehat{g(w)}\right) |\!)$$

It is now time to define $j_f \in \mathsf{Hom}_{\mathcal{C}}\left(F_{\delta_f}(\{\bullet\}), A\right)$. To do so, let us first introduce

$$\mathsf{appto}(\varphi) \ : \ B \multimap C \xrightarrow{\sim} (B \multimap C) \otimes \mathbf{I} \xrightarrow{\mathrm{id} \otimes \varphi} (B \multimap C) \otimes B \xrightarrow{\mathrm{ev}} C$$

where the last arrow is the evaluation map $\mathrm{ev}_{B,C}$, for any $B, C \in \mathsf{Obj}(\mathcal{C})$ and $\varphi : \mathbf{I} \to B$. An useful property, whose verification we leave to the reader, is

$$\mathsf{appto}(\varphi) \circ \Lambda'(\psi) \quad = \quad \psi \circ \varphi \qquad \text{for any } \psi : B \to C$$

where $\Lambda' : \mathsf{Hom}_{\mathcal{C}}(B, C) \xrightarrow{\sim} \mathsf{Hom}_{\mathcal{C}}(\mathbf{I},\ B \multimap C)$ is defined in Proposition 4.1.9. We then take

$$j_f \ : \ F_{\delta_f}(\{\bullet\}) \xrightarrow{\pi_1} (A \multimap A) \multimap (A \multimap A) \xrightarrow{\mathsf{appto}(\Lambda'(\mathrm{id}_A))} A \multimap A \xrightarrow{\mathsf{appto}(i_f)} A$$

where π_1 is the left projection from $F_{\delta_f}(\{\bullet\}) = ((A \multimap A) \multimap (A \multimap A)) \,\&\, \mathbf{I}$ (this equality is guaranteed by Corollary 4.6.3) and i_f is the initialization morphism of T_f. Using the equation

$$F_{\delta_f}\left(\widehat{g(w)}\right) \quad = \quad \left\langle \Lambda'\left(\widetilde{\delta_f}(g(w)_1) \circ \dots \circ \widetilde{\delta_f}(g(w)_m)\right),\ \mathrm{id}_{\mathbf{I}} \right\rangle \quad \text{where } m = |g(w)|$$

coming from Corollary 4.6.3, we then have

$$j_f \circ F_{\delta_f}\left(\widehat{g(w)}\right) \quad = \quad \mathsf{appto}(i_f) \circ \widetilde{\delta_f}(g(w)_1) \circ \dots \circ \widetilde{\delta_f}(g(w)_m) \circ \Lambda'(\mathrm{id}_A)$$

Next, we define $\widetilde{\delta_f}(c) = \Lambda(\mathrm{ev}_{A,A} \circ (\mathrm{id}_{A \multimap A} \otimes \delta_f(c))) \in \mathsf{Hom}_{\mathcal{C}}(A \multimap A,\ A \multimap A)$ for $c \in \Gamma$. In other words, $\widetilde{\delta_f}(c)$ is the curryfication of

$$(A \multimap A) \otimes A \xrightarrow{\mathrm{id} \otimes \delta_f(c)} (A \multimap A) \otimes A \xrightarrow{\mathrm{ev}} A$$

One can then check that $\widetilde{\delta_f}(c) \circ \Lambda'(\psi) = \Lambda'(\psi \circ (\delta_f(c)))$ for any $\psi : A \to A$. Putting everything together, we finally have

$$\begin{aligned} T(w) &= (\!| o_f \circ \mathsf{appto}(i_f) \circ \widetilde{\delta_f}(g(w)_1) \circ \dots \circ \widetilde{\delta_f}(g(w)_m) \circ \Lambda'(\mathrm{id}_A) |\!) \\ &= (\!| o_f \circ \mathsf{appto}(i_f) \circ \Lambda'(\delta_f(g(w)_m) \circ \dots \circ \delta_f(g(w)_1)) |\!) \\ &= (\!| o_f \circ \delta_f(g(w)_m) \circ \dots \circ \delta_f(g(w)_1) \circ i_f |\!) \end{aligned}$$

and this final expression is precisely the definition of the output of T_f on $g(w)$. Since T_f computes f, we end up with $T(w) = f(g(w))$, as we wanted. □

To conclude this section, let us note that an analogous result for precomposition by regular tree functions can be shown by leveraging the results of Chapter 6; we leave it as an exercise. An important subtlety: since the presence of the additive conjunction is important to compute regular tree functions (as we stressed in the introduction), one must consider tree streaming settings whose underlying categories *have finite cartesian products* (which entails quasi-affineness).

4.8. Uniformization through monoidal closure

Let us wrap up this chapter by recalling below the categorical uniformization theorem that we mentioned in Section 4.4 and providing its proof. (Recall that according to Lemma 4.4.8, the conclusion amounts to saying that non-deterministic sdm-$\mathfrak{C}$-SSTs are uniformizable by partial sdm-$\mathfrak{D}$-SSTs.)

Theorem 4.8.1. *Let $\mathfrak{C}$ and $\mathfrak{D}$ be streaming settings such that there is a morphism of streaming settings $\mathfrak{C} \to \mathfrak{D}$, whose underlying functor is $F : \mathcal{C} \to \mathcal{D}$. Assume further that $\mathcal{D}$ carries a* symmetric monoidal affine *structure and has* internal homsets *$F(C) \multimap F(C')$ for every pair of objects $C, C' \in \mathsf{Obj}(\mathcal{C})$.*

Then, sdm-$\mathfrak{C}_{\&}$-SSTs are subsumed by sdm-$\mathfrak{D}$-SSTs.

As promised at the end of Section 4.4, our technical development until now allows us to derive the equivalence between sdm-$\mathfrak{SR}_{\&}$-SSTs and sdm-$\mathfrak{SR}$-SSTs from the above result.

Proof of Theorem 4.4.9. We instantiate Theorem 4.8.1 with $\mathfrak{C} = \mathfrak{SR}$, $\mathfrak{D} = \mathfrak{SR}_{\oplus}$, and the functor $\iota_{\oplus} : \mathcal{SR} \to \mathcal{SR}_{\oplus}$. We have already seen that $\mathcal{SR}_{\oplus}$ is a symmetric monoidal affine category. The assumption on internal homsets is exactly Lemma 4.3.14. The theorem then tells us that sdm-$\mathfrak{SR}_{\&}$-SSTs are subsumed by sdm-$\mathfrak{SR}_{\oplus}$-SSTs, and the latter are no more expressive than sdm-$\mathfrak{SR}$-SSTs by Corollary 4.3.9. □

Not coincidentally, the idea for our proof of Theorem 4.8.1 is extracted from a direct determinization proof for copyless SSTs. Our main contribution is in recognizing that the latter implicitly involves internal homsets as one of its key components. (While the determinization argument in [AD11] goes through MSO, the direct proof might be part of the folklore, see e.g. [BC18, Problem 139 (p. 226)].)

We do not prove the statement in excruciating details, but provide key formal definitions so that a reader familiar with a modicum of automata theory and category theory should be able to reconstitute a fully formal argument with ease. Let us stress once again that all of the combinatorics may be regarded as adaptation of known arguments.

Our approach relies on a notion of *transformation forest*, a name that we borrow from [BC18, Chapter 13] for an extremely similar concept[5]. This gadget is also reminiscent of trees used in determinization procedures like the Muller-Schupp construction for automata over ω-words [MS95] (another exposition can be found in [BC18, Chapter 1]), and of the sharing techniques used in the original paper on SSTs [AČ10, §5.2]. In determinization procedures, this constitutes an elaboration of powerset constructions recalling not only

[5]There are two formal differences between our notions, which are not very big but worth mentioning for readers of [BC18]. First, edges of a transformation forest are intended to be associated with (elements of) a monoid, while ours should be associated with ("elements of") internal homsets $F(C_u) \multimap F(C_v)$. Were we trying to prove $\mathfrak{D}$-uniformization for $\mathfrak{C}$-SSTs, we would have necessarily $C_u = C_v$ and the aforementioned object would have a monoid structure internal to $\mathcal{D}$, so this distinction is more an artefact of our settings rather than an essential one. Second, what [BC18] calls transformation forests refers to a class of forest with "no junk", such as dangling leaves not referring to an intended output or spurious internal nodes, while we allow those in an initial definition; we add the adjective "normalized" for those containing "no junk" as we shall see later, so this is merely a terminological detail.

reachable states, but also crucial information on *how* those states are reached. Here, the vertices v of such forests will be labelled by objects C_v of $\mathcal{C}$ and each edge (u, v) will be correspond to a "register containing a value of type $F(C_u) \multimap F(C_v)$".

We decompose this proof sketch in three parts: first, we introduce transformation forests, their semantic interpretation as families of maps in $\mathcal{D}$; we explain how they may be composed and that maps in $\mathcal{C}_{\&}$ may be regarded as depth-1 transformation forests. Then, we explain how to reduce the size of transformation forests in a sound way (this is the crucial part ensuring that the resulting sdm-$\mathfrak{D}$-SSTs will have finitely many states). Finally, we explain how to put all of this together to uniformize sdm-$\mathfrak{C}$-SSTs.

4.8.1. **Transformation forests and their semantics.** A transformation forest is defined as a tuple $\mathcal{F} = (V, E, O, (C_v)_{v \in V})$ where

- V is a non-empty finite set of vertices
- $E \subseteq V^2$ is a set of directed edges, pointing from parents to children
- O is a non-empty subset of V which we cal the set of *output nodes*
- every C_v is an object of $\mathcal{C}$

When a transformation forest $\mathcal{F}$ is fixed, we call $I_{\mathcal{F}}$ its set of roots (which we may sometimes *input nodes*; we drop the subscript when there is no ambiguity). Given a transformation forest $\mathcal{F} = (V, E, (v_o)_{o \in O}, (C_v)_{v \in V})$, we assign the following object of $\mathcal{D}$:

$$\mathrm{Ty}(\mathcal{F}) \quad = \quad \bigotimes_{(u,v) \in E} F(C_u) \multimap F(C_v)$$

An example of a transformation forest $\mathcal{F}$ and a computation of its type $\mathrm{Ty}(\mathcal{F})$ is pictured in Figure 4.8.1. To guide the intuition, one may note that there is an embedding[6] of a set of suitable labellings of the forest $\mathcal{F}$ into $\mathsf{Hom}_{\mathcal{D}}(\top, \mathrm{Ty}(\mathcal{F}))$.

$$\prod_{(u,v) \in E} \mathsf{Hom}_{\mathcal{D}}(F(C_u), F(C_v)) \quad \longrightarrow \quad \mathsf{Hom}_{\mathcal{D}}(\top, \mathrm{Ty}(\mathcal{F}))$$

We will now call abusively the input of $\mathcal{F}$ the object $A = \&_{i \in I} C_i$ and the output $B = \&_{o \in O} C_o$; we write $\mathcal{F} : A \to B$ in the sequel. The justification for this notation is that there is a family of maps

$$[\![\mathcal{F}]\!] \ \in \ \prod_{o \in O} \sum_{i \in I} \mathsf{Hom}_{\mathcal{D}}(\mathrm{Ty}(\mathcal{F}), F(C_i) \multimap F(C_o))$$

obtained by internalizing the composition of morphisms along branches of $\mathcal{F}$, which we call the *semantics of* $\mathcal{F}$.

We also note that this allows to interpret arbitrary $\mathcal{C}_{\&}$-morphisms: such a morphism $f : A \to B$ can be mapped to a pair $(\mathrm{Grph}(f), \widehat{f})$ consisting of

- a depth-1 transformation forest $\mathrm{Grph}(f) : \&_{i \in I} C_i \to \&_{o \in O} C_o$ (cf. Figure 4.8.2)
- a morphism $\widehat{f} \in \mathsf{Hom}_{\mathcal{D}}(\top, \mathrm{Ty}(\mathcal{F}))$

so that, if $f = (i_o, \alpha_o)_{o \in O}$, we have $[\![\mathrm{Grph}(f)]\!]_o \circ \widehat{f} = (i_o, \Lambda(\alpha_o \circ \rho^{-1}))$.

Given two forests $\mathcal{F} : A \to B$ and $\mathcal{F}' : B \to C$, there is a composition $\mathcal{F}' \circ \mathcal{F}$ obtained by gluing the input nodes of $\mathcal{F}'$ along the output nodes of $\mathcal{F}$. A crucial point is that the semantics of the composite $[\![-]\!]$ may be computed as follows

$$[\![\mathcal{F}' \circ \mathcal{F}]\!](o) \ = \ \big([\![\mathcal{F}]\!]_1([\![\mathcal{F}']\!]_1(o')), [\![\mathcal{F}']\!]_2(o') \circ [\![\mathcal{F}]\!]_2([\![\mathcal{F}']\!]_1)\big)$$

[6] This becomes an isomorphism when $\mathsf{Hom}_{\mathcal{D}}(\top, A \otimes B) \cong \mathsf{Hom}_{\mathcal{D}}(\top, A) \times \mathsf{Hom}_{\mathcal{D}}(\top, B)$; this is the case for our intended application $\mathcal{D} = \mathcal{SR}_{\oplus}$.

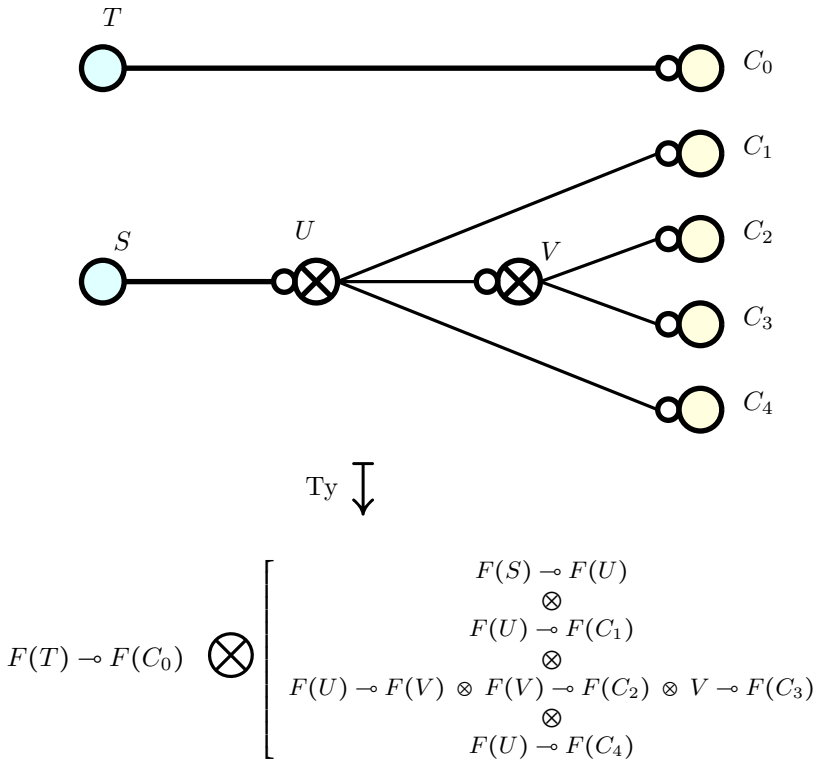


FIGURE 4.8.1. A transformation forest $\mathcal{F} : T \,\&\, S \to \mathop{\&}_{i=0}^{4} C_i$.

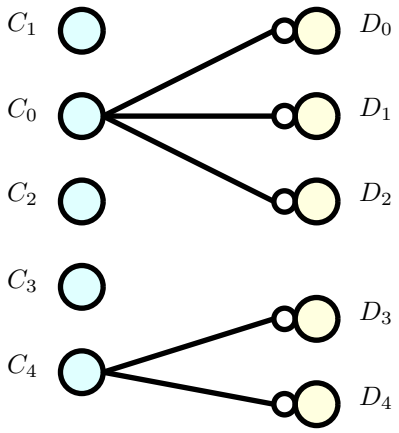


FIGURE 4.8.2. A depth-1 transformation forest.

4.8.2. **Reducing transformation forests.** We now introduce two elementary transformations $\mathcal{F} \mapsto \mathcal{F}'$ over transformation forests, together with associated morphisms $\mathrm{Ty}(\mathcal{F}) \to \mathrm{Ty}(\mathcal{F}')$:

- **Pruning**: if $v \in V_{\mathcal{F}}$ is a leaf which is not an output node in the forest $\mathcal{F} : A \to B$, call $\mathrm{prune}(\mathcal{F}, v) : A \to B$ the forest obtained by removing v and adjacent edges from $\mathcal{F}$.

  This induces a canonical map

$$\mathit{prune}(v) : \mathrm{Ty}(\mathcal{F}) \to \mathrm{Ty}(\mathrm{prune}(\mathcal{F}, v))$$

  by using the weakening maps on the components corresponding to the deleted edges.
- **Contraction**: If there is a vertex v with a unique child c and a (unique) parent p in the forest $\mathcal{F} : A \to B$, call $\mathrm{contract}(\mathcal{F}, v) : A \to B$ the forest obtained by replacing the edges (p, v) and (v, c) with a single edge (p, c) and removing v.

  There is a canonical map

$$\mathit{contract}(v) : \mathrm{Ty}(\mathcal{F}) \to \mathrm{Ty}(\mathrm{contract}(\mathcal{F}, v))$$

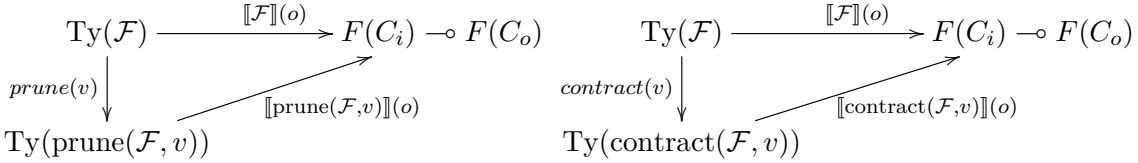
  induced by the internal composition map

$$\mathsf{Hom}_{\mathcal{D}}\left([F(C_p) \multimap F(C_v)] \otimes [F(C_v) \multimap F(C_c)], F(C_p) \multimap F(C_c)\right)$$

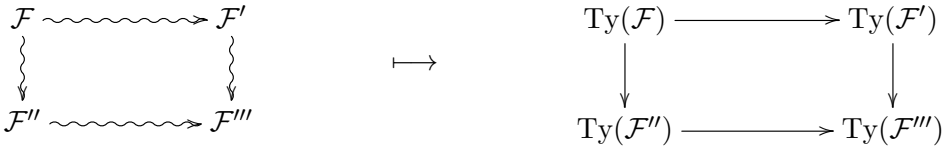
The auxiliary maps *prune* and *contract* operations are compatible with the semantic map $[\![\cdot]\!]$ in the sense that for every $o \in O$ and suitable vertices u, v of $\mathcal{F}$, we have

$$i = \pi_1([\![\mathcal{F}]\!](o)) = \pi_1([\![\mathrm{prune}(\mathcal{F}, u)]\!](o)) = \pi_1([\![\mathrm{contract}(\mathcal{F}, v)]\!](o))$$

and the following diagrams commute in $\mathcal{D}$



Consider the rewrite system $\rightsquigarrow$ over forests $\mathcal{F} : A \to B$ induced by those two operations, and call $\rightsquigarrow^{=}$ its reflexive closure and $\rightsquigarrow^{*}$ the transitive closure of $\rightsquigarrow^{=}$. It can be shown that the reflexive closure $\rightsquigarrow^{=}$ satisfies the *diamond lemma*, i.e., that for every $\mathcal{F}, \mathcal{F}'$ and $\mathcal{F}''$ such that $\mathcal{F} \rightsquigarrow^{=} \mathcal{F}'$ and $\mathcal{F} \rightsquigarrow^{=} \mathcal{F}''$, there exists $\mathcal{F}'''$ such that $\mathcal{F}' \rightsquigarrow^{=} \mathcal{F}'''$ and $\mathcal{F}'' \rightsquigarrow^{=} \mathcal{F}'''$. This ensures that $\rightsquigarrow$ is confluent. Furthermore, given a rewrite $\mathcal{F} \rightsquigarrow^{*} \mathcal{F}'$, there is a map $\mathrm{Ty}(\mathcal{F}) \to \mathrm{Ty}(\mathcal{F}')$ obtained by composing maps $\mathit{prune}(v)$ and $\mathit{contract}(v)$ (and identities for trivial rewrites $\mathcal{F} \rightsquigarrow^{*} \mathcal{F}$). It can be shown that this map does *not* depend on the rewrite path. This is done first by arguing that if we have a rewrite square for $\rightsquigarrow^{=}$, the associated diagram in $\mathcal{D}$ is commutative, and then proceed by induction over the rewrite paths using the diamond lemma.



Defining the size of a forest as its number of vertices, it is clear that $\mathrm{prune}(\mathcal{F}, u)$ and $\mathrm{contract}(\mathcal{F}, v)$ have size strictly less than $\mathcal{F}$, so the rewrite system is also terminating. With confluence, it means that for every forest $\mathcal{F} : A \to B$, there is a unique forest $\mathrm{reduce}(\mathcal{F}) : A \to B$ such that $\mathcal{F} \rightsquigarrow^{*} \mathrm{reduce}(\mathcal{F})$ and there is no $\mathcal{F}'$ such that $\mathrm{reduce}(\mathcal{F}) \rightsquigarrow \mathcal{F}'$. We call $\mathrm{reduce}(\mathcal{F})$ the *normal form* of $\mathcal{F}$; a forest $\mathcal{F}$ is called *normal* if $\mathrm{reduce}(\mathcal{F}) = \mathcal{F}$. By the discussion above, there are canonical maps $\mathit{reduce}_{\mathcal{F}} : \mathrm{Ty}(\mathcal{F}) \to \mathrm{Ty}(\mathrm{reduce}(\mathcal{F}))$ coming from rewrites.

The last important thing to note is that, if the input $A = \&_{i \in I} C_i$ and output $B = \&_{o \in O} C_o$ are fixed, up to isomorphism, there are finitely many normal forests $\mathcal{F} : A \to B$ as

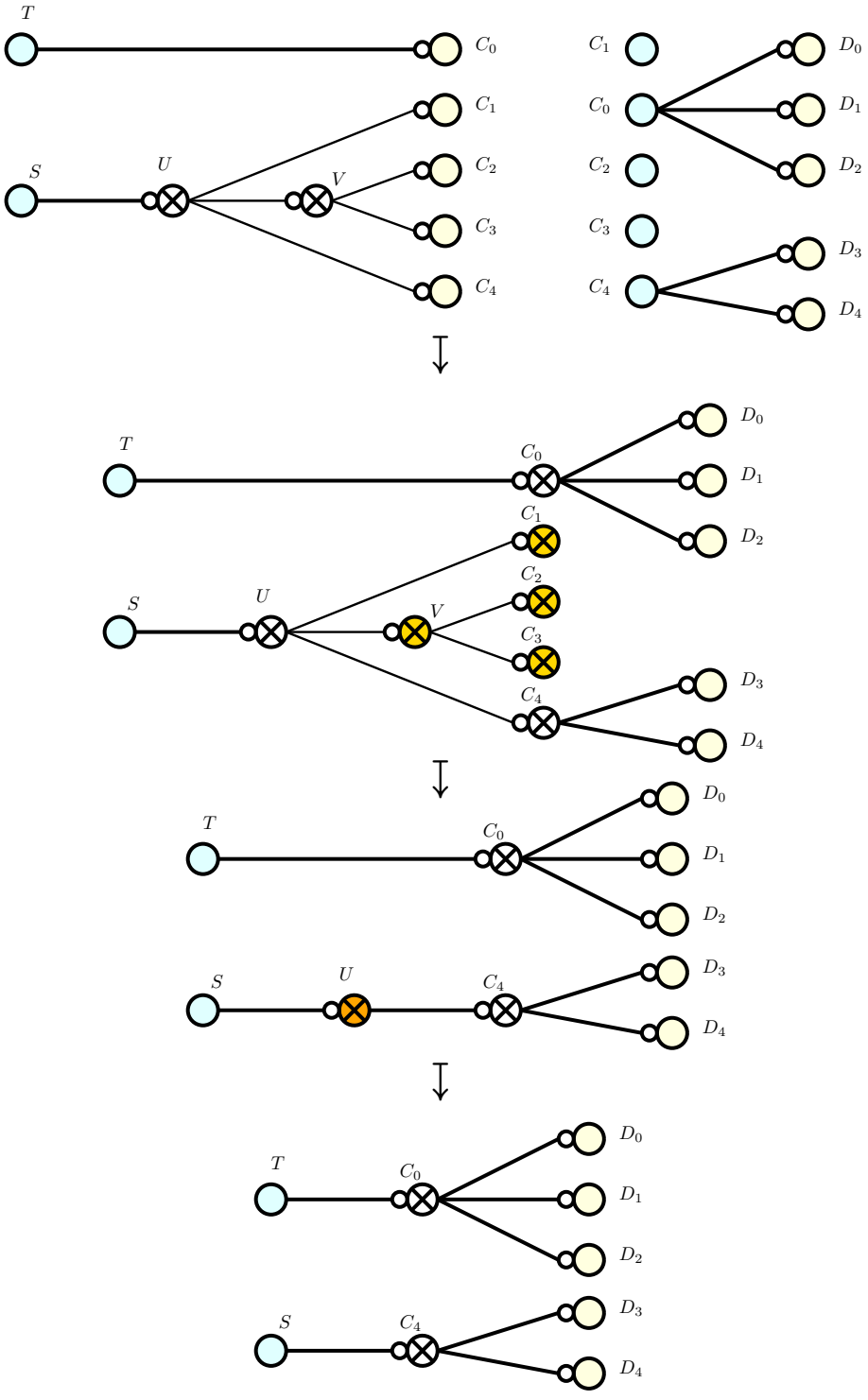


FIGURE 4.8.3. An example of composing normal transformation forests, where first the usual composition and then two big steps of reduction (that are respectively pruning and contracting) are carried out in succession.

their size is bounded by $2(|I|+|O|)$. We write $\mathrm{NF}(A,B)$ for a finite set of representative for all normal forests $A \to B$, and given $\mathcal{F} \in \mathrm{NF}(B,C)$ and $\mathcal{F}' \in \mathrm{NF}(A,B)$, we write $\mathcal{F} \circ_{\mathcal{N}} \mathcal{F}'$ for the unique forest in $\mathrm{NF}(A,C)$ which is isomorphic to $\mathrm{reduce}(\mathcal{F} \circ \mathcal{F}')$; an example of the full computation of a $\mathcal{F} \circ_{\mathcal{N}} \mathcal{F}'$ is given in Figure 4.8.3. Similarly, we assume that $\mathrm{Grph}(f) \in \mathrm{NF}(A,B)$ for every morphism $f \in \mathsf{Hom}_{\mathcal{C}_\&}(A,B)$.

4.8.3. **Putting everything together.** Let $\mathcal{T} = \left(Q, q_0, (A_q)_{q\in Q}, \delta, i, o\right)$ be a sdm-$\mathfrak{C}_\&$-SST with input Σ^*. We define the sdm-$\mathfrak{D}$-SST

$$\mathcal{T}' \;=\; \left(\sum_{q\in Q} \mathrm{NF}(A_{q_0}, A_q), (q_0, \mathrm{Grph}(\mathrm{id})), (\mathrm{Ty}(\mathcal{F}))_{q,\mathcal{F}}, \delta', i'o'\right)$$

where

$$\delta' \;:\; \Sigma \to \prod_{q,\mathcal{F}} \sum_{r,\mathcal{F}'} \mathsf{Hom}_{\mathcal{D}}\left(\mathrm{Ty}(\mathcal{F}), \mathrm{Ty}(\mathcal{F}')\right)$$

$$i' \in \mathsf{Hom}_{\mathcal{D}}(\top, \mathrm{Grph}(\mathrm{id})) \qquad\qquad o' \in \prod_{q,\mathcal{F}} \mathsf{Hom}_{\mathcal{D}}(\mathrm{Ty}(\mathcal{F}), \perp\!\!\!\perp)$$

are given as follows, assuming that $A_q = \mathop{\&}_{x\in X_q} C_{q,x}$

- Notice that $\mathrm{Ty}(\mathrm{Grph}(\mathrm{id})) = \bigotimes_{x\in X_{q_0}} F(C_{q_0,x}) \multimap F(C_{q_0,x})$; we simply take the constant map corresponding to the X_{q_0}-fold tensor of $\Lambda(\mathrm{id}_{C_{q_0,x}})$ for i'.
- Fix $a \in \Sigma$ and recall that $\delta(a)$ is a family of pairs
$$(\delta^Q(a)_q, \delta^{\mathcal{C}_\&}(a)_q)_{q\in Q} \;\in\; \prod_{q\in Q} \sum_{r\in Q} \mathsf{Hom}_{\mathcal{C}_\&}(A_q, A_r)$$
δ' is then defined by the equation
$$\delta'(a)_{q,\mathcal{F}} \;=\; \left(\left(\delta^Q(a)_q, \delta^{\mathrm{NF}}(a)_{q,\mathcal{F}}\right), \delta^{\mathcal{D}}(a)_{q,\mathcal{F}}\right)$$
where we set $\delta^{\mathrm{NF}}(a)_{q,\mathcal{F}} = \mathrm{Grph}(\delta^{\mathcal{C}_\&}(a)_q) \circ_{\mathcal{N}} \mathcal{F}$ and $\delta^{\mathcal{D}}(a)_{q,\mathcal{F}}$ is obtained as in Figure 4.8.4
- Finally, for $q, \mathcal{F}$ ranging over states of $\mathcal{T}'$, we define $o'_{q,\mathcal{F}}$. Recall that $\mathcal{F}$ determines a canonical family
$$[\![\mathcal{F}]\!] \in \prod_{x\in X_q} \sum_{x_0\in X_{q_0}} \mathsf{Hom}_{\mathcal{D}}(\mathrm{Ty}(\mathcal{F}), F(C_{q_0,x_0}) \multimap F(C_{q,x}))$$
First, as $\perp\!\!\!\perp^{\mathfrak{C}_\&} = \iota_\&(\perp\!\!\!\perp^{\mathfrak{C}})$, note that there is a unique $x \in X_q$ such that $o_q = (x, o^{\mathcal{C}})_*$ for some $o^{\mathcal{C}} \in \mathsf{Hom}_{\mathcal{C}}\left(C_{q,x}, \perp\!\!\!\perp^{\mathfrak{C}}\right)$. Writing the pair $[\![\mathcal{F}]\!]_x$ as (x_0, f), $o'_{q,\mathcal{F}}$ is depicted in Figure 4.8.4.

We omit the proof that $[\![\mathcal{T}]\!] = [\![\mathcal{T}']\!]$ by induction over the length of an input word. While spelling it out may be a bit notation-heavy, there is no particular difficulty considering the remarks above linking $[\![-]\!]$, the composition of transformation forests and their normal forms.

$\delta^{\mathcal{D}}(a)_{q,\mathcal{F}}$	$o'_{q,\mathcal{F}}$
$\mathrm{Ty}(\mathcal{F})$ $\downarrow \sim$ $\top \otimes \mathrm{Ty}(\mathcal{F})$ $\downarrow \widehat{\delta^{\mathcal{C}_{\&}}(a)_q} \otimes \mathrm{id}$ $\mathrm{Ty}(\mathrm{Grph}(\delta^{\mathcal{C}_{\&}}(a)_q)) \otimes \mathrm{Ty}(\mathcal{F})$ $\downarrow \sim$ $\mathrm{Ty}(\mathrm{Grph}(\delta^{\mathcal{C}_{\&}}(a)_q) \circ \mathcal{F})$ $\downarrow reduce$ $\mathrm{Ty}(\mathrm{reduce}(\mathrm{Grph}(\delta^{\mathcal{C}_{\&}}(a)_q) \circ \mathcal{F}))$ $\Vert$ $\mathrm{Ty}(\delta^{\mathrm{NF}}(a)_{q,\mathcal{F}})$	$\mathrm{Ty}(\mathcal{F})$ $\downarrow f$ $F(C_{q_0,x_0}) \multimap F(C_{q,x})$ $\downarrow \sim$ $(F(C_{q_0,x_0}) \multimap F(C_{q,x})) \otimes \top\!\!\top$ $\downarrow$ $(F(C_{q_0,x_0}) \multimap F(C_{q,x})) \otimes F(\top\!\!\top)$ $\downarrow \mathrm{id} \otimes F(i_{x_0})$ $(F(C_{q_0,x_0}) \multimap F(C_{q,x})) \otimes F(C_{q_0,x_0})$ $\downarrow \mathrm{ev}$ $F(C_{q,x})$ $\downarrow F(o^{\mathcal{C}})$ $F(\perp\!\!\!\perp)$ $\downarrow$ $\perp\!\!\!\perp$

FIGURE 4.8.4. Definition of $\delta^{\mathcal{D}}(a)_{q,\mathcal{F}}$ and $o'_{q,\mathcal{F}}$.

CHAPTER 5

String transductions in a linear λ-calculus

Now that we are equipped with the categorical tools developed in the previous chapter, we can prove the parts of Theorem 1.2.3 pertaining to string functions, that is, the first two rows. That is the goal of this chapter. Some of our developments will also apply more generally to ranked trees, as a preparation for the next chapter where the final row of Theorem 1.2.3 (concerning regular tree functions) will be shown.

In Section 5.1, we introduce the equational theory and typing rules of the $\lambda\ell^{\oplus\&}$-calculus, and explain how it can be used to define functions on strings and trees thanks to Church encodings, with a few examples. Section 5.2 uses a syntactic lemma to reframe our implicit characterization of regular functions as the equivalence between $\mathfrak{SR}$-SSTs and single-state $\mathfrak{L}$-SSTs – where $\mathfrak{L}$ is a streaming setting made of purely linear $\lambda\ell^{\oplus\&}$-terms – and then proves this categorical reformulation as an immediate application of Theorem 1.2.4 / Corollary 4.5.4. As for the part of Theorem 1.2.3 about comparison-free polyregular functions, it is tackled with a more ad-hoc argument in Section 5.3.

Both sections 5.2 and 5.3 depend on an analysis of the syntax of the $\lambda\ell^{\oplus\&}$-calculus to prepare the groundwork for their semantic evaluation arguments. The required syntactic properties are encapsulated in concise lemmas whose proofs are deferred to the final section of this chapter (§5.4). We expend considerable effort on those proofs – most of it spent on routine yet cumbersome bureaucracy, and some on actual technical subtleties – but the obstacles are unrelated to the various conceptual points concerning automata and semantics that we wished to stress. As we already summarized those points in the introduction (§1.2), let us say a few words now about syntax. *(This discussion can be safely skipped.)*

The idea is that, since the string/tree function defined by a $\lambda\ell^{\oplus\&}$-term depends only on its $\beta\eta$-equivalence class, we will pick a representative of this class whose shape is restricted (and thus can be exhaustively analyzed) depending on its type, called a *normal form*. Much of Section 5.4 is devoted to *normalization* results, i.e. "every $\lambda\ell^{\oplus\&}$-term is $\beta\eta$-equivalent to a normal form". We prove this first for an "old-school" definition of normal and neutral terms, and then for a more refined notion of *focused* normal forms.

For our purposes, the naive notion of β-normal form is inadequate because of the *positive* connectives $\otimes/\oplus$. (A term is β-normal if it does not admit any β-reduction, and the β-reduction relation is obtained by orienting some equations from Figure 5.1.2.) Our better notions of normal form can be reached by combining β-reductions with applications of well-chosen η-conversions (called *extrusions*) whose role to unlock new β-redexes. Similar issues arise in the literature concerned with deciding $\beta\eta$-equivalence in λ-calculi with positive connectives, e.g. [Lin07; Alt+01]; we refer to [Sch16] for a comprehensive and pedagogical overview of this subject, including the application of focusing to canonical normal forms for simply typed λ-terms with sums. In our case, we are interested in normal forms only because they lend themselves to (lengthy) case analyses to understand the closed inhabitants of the types $\mathtt{Tree}_{\mathbf{\Gamma}}[\tau] \multimap \mathtt{Tree}_{\mathbf{\Sigma}}$ and $\mathtt{Str}_{\Gamma}[\tau] \to \mathtt{Str}_{\Sigma}$. Our normal forms are *not* meant to serve as canonical representatives of $\beta\eta$-equivalence classes, nor do they provide a decision procedure for equivalence.

All these complications could have been avoided by restricting to the fragment of the $\lambda\ell^{\oplus\&}$-calculus containing only the negative connectives $\to$ / $\multimap$ /& and the corresponding type constructors, for which β-normal forms would have been sufficient – in fact, we might choose to do so in an upcoming journal version to avoid frightening readers. (Additionally, normalization for this negative fragment is immediate by a reduction-preserving translation to the simply typed λ-calculus with surjective pairing.) However, it was tempting to include those positive connectives in our work since our semantic tools can handle them without difficulty (our category $\mathcal{SR}_{\oplus\&}$ from the previous chapter can interpret $\otimes/\oplus$). The choice made in this thesis is also an opportunity to demonstrate that the existing tools for working with the syntactic metatheory of typed λ-calculi are perfectly adequate for the job.

5.1. The $\lambda\ell^{\oplus\&}$-calculus, Church encodings, and definability of functions

5.1.1. **Types & terms.** We consider a linear λ-calculus which we dub the *$\lambda\ell^{\oplus\&}$-calculus*, based (via the proofs-as-programs correspondence) on propositional intuitionistic linear logic with both multiplicative and additive connectives (IMALL) together with a base type $\mathbb{o}$. The grammar of types is as follows:

$$\tau, \sigma ::= \mathbb{o} \mid \tau \multimap \sigma \mid \tau \otimes \sigma \mid \mathbf{I} \mid \tau \to \sigma \mid \tau \,\&\, \sigma \mid \tau \oplus \sigma \mid \top \mid 0$$

A typing context Ψ is a finite set of declarations $x_1 : \tau_i, \ldots, x_k : \tau_k$ where the x_i are pairwise distinct variables (which constitute the set of free variables of Ψ) and the τ_i are types. Typed $\lambda\ell^{\oplus\&}$-terms are given in Figure 5.1.1 along with the inductive definition of the typing judgment $\Psi;\ \Delta \vdash t : \tau$, where Ψ and Δ are contexts (with disjoint sets of free variables), τ is a type and t is a term. In such a judgment, Ψ is called the *non-linear* context and Δ the *linear* context; the basic idea is that variables in Ψ may be "used" arbitrarily many times, while those in Δ must be "used" *exactly once* (for a notion of usage such that the additive pair $\langle x, x\rangle$ uses x once, whereas the multiplicative pair $x \otimes x$ uses x twice).

Remark 5.1.1. This is formally more restrictive than an *affineness condition*, i.e. than requiring variables in Δ to be used *at most once*. In practice, $\lambda\ell^{\oplus\&}$ is not less expressive than its affine variant, which would be obtained by adjoining the following *weakening rule*:

$$\frac{\Psi;\ \Delta \vdash t : \tau}{\Psi;\ \Delta,\ \Delta' \vdash t : \tau}$$

The basic idea is that the affineness can be encoded at the level of types by using the linear type $\tau \,\&\, \mathbf{I}$ instead of the affine type τ (as argued for instance in [Gir95, §1.2.1]).

The simply typed λ-calculus admits an embedding into $\lambda\ell^{\oplus\&}$. Conversely, there is a mapping from $\lambda\ell^{\oplus\&}$ to the simply typed λ-calculus with products and sums by "forgetting linearity" (and replacing the tensorial product eliminator $\texttt{let}\ x \otimes y = t\ \texttt{in}\ u$ by the variant based on projections $u[\pi_1(t)/x, \pi_2(t)/y]$).

As usual, we identify $\lambda\ell^{\oplus\&}$-terms up to renaming of bound variables (α-equivalence) and admit the standard definition of the capture-avoiding substitution. The computational meaning of terms is specified by their $\beta\eta$-equivalence (denoted by $=_{\beta\eta}$), generated by the equations in Figure 5.1.2 and congruence. Note that those equations are implicitly typed and that typing is invariant under $\beta\eta$-equivalence.

We now isolate an important class of types and terms for the sequel.

Definition 5.1.2. Recall from the introduction (Definition 1.2.2) that we call a type *purely linear* if it does not have any occurrence of the '$\to$' connective. A $\lambda\ell^{\oplus\&}$-term t is also said to be *purely linear* if the typing judgment $\Psi;\ \Delta \vdash t : \tau$ admits a derivation where all the types that occur are purely linear.

$$\frac{}{\Psi;\ x:\tau \vdash x:\tau} \qquad \frac{}{\Psi, x:\tau;\ \cdot \vdash x:\tau}$$

$$\frac{\Psi;\ \Delta,\ x:\tau \vdash t:\sigma}{\Psi;\ \Delta \vdash \lambda x.t:\tau \multimap \sigma} \qquad \frac{\Psi;\ \Delta \vdash t:\tau \multimap \sigma \qquad \Psi;\ \Delta'' \vdash u:\tau}{\Psi;\ \Delta,\ \Delta' \vdash t\ u:\sigma}$$

$$\frac{\Psi,\ x:\tau;\ \Delta \vdash t:\sigma}{\Psi;\ \Delta \vdash \lambda^{!}x.t:\tau \to \sigma} \qquad \frac{\Psi;\ \Delta \vdash t:\tau \to \sigma \qquad \Psi;\ \cdot \vdash u:\tau}{\Psi;\ \Delta \vdash t\ u:\sigma}$$

$$\frac{\Psi;\ \Delta \vdash t:\tau \qquad \Psi;\ \Delta' \vdash u:\sigma}{\Psi;\ \Delta,\ \Delta' \vdash t \otimes u:\tau \otimes \sigma} \qquad \frac{\Psi;\ \Delta' \vdash u:\tau \otimes \sigma \qquad \Psi;\ \Delta, x:\tau, y:\sigma \vdash t:\kappa}{\Psi;\ \Delta,\ \Delta' \vdash \mathtt{let}\ x \otimes y = u\ \mathtt{in}\ t:\kappa}$$

$$\frac{}{\Psi;\ \cdot \vdash ():\mathbf{I}} \qquad \frac{\Psi;\ \Delta \vdash t:\mathbf{I} \qquad \Psi;\ \Delta' \vdash u:\tau}{\Psi;\ \Delta,\ \Delta' \vdash \mathtt{let}\ () = t\ \mathtt{in}\ u:\tau}$$

$$\frac{\Psi;\ \Delta \vdash t:\tau \qquad \Psi;\ \Delta \vdash u:\sigma}{\Psi;\ \Delta \vdash \langle t,u\rangle:\tau\ \&\ \sigma} \qquad \frac{\Psi;\ \Delta \vdash t:\tau\ \&\ \sigma}{\Psi;\ \Delta \vdash \pi_1(t):\tau} \qquad \frac{\Psi;\ \Delta \vdash t:\tau\ \&\ \sigma}{\Psi;\ \Delta \vdash \pi_2(t):\sigma}$$

$$\frac{\Psi;\ \Delta \vdash t:\tau}{\Psi;\ \Delta \vdash \mathsf{in}_1(t):\tau \oplus \sigma} \qquad \frac{\Psi;\ \Delta \vdash t:\sigma}{\Psi;\ \Delta \vdash \mathsf{in}_2(t):\tau \oplus \sigma}$$

$$\frac{\Psi;\ \Delta, x:\tau \vdash u:\kappa \qquad \Psi;\ \Delta, x:\tau \vdash v:\kappa \qquad \Psi;\ \Delta' \vdash t:\tau \oplus \sigma}{\Psi;\ \Delta, \Delta' \vdash \mathsf{case}(t, x.u, x.v):\kappa}$$

$$\frac{}{\Psi;\ \Delta \vdash \langle\rangle:\top} \qquad \frac{\Psi;\ \Delta \vdash t:0}{\Psi;\ \Delta,\ \Delta' \vdash \mathsf{abort}(t):\tau}$$

FIGURE 5.1.1. Typing rules of $\lambda\ell^{\oplus\&}$.

β:	$(\lambda x.t)\ u\ =_\beta\ t[u/x]$	$(\lambda^{!}x.t)\ u\ =_\beta\ t[u/x]$
	$\mathsf{case}(\mathsf{in}_1(t), x.u, x.v)\ =_\beta\ u[t/x]$	$\pi_1(\langle t,u\rangle)\ =_\beta\ t$
	$\mathsf{case}(\mathsf{in}_2(t), x.u, x.v)\ =_\beta\ v[t/x]$	$\pi_2(\langle t,u\rangle)\ =_\beta\ u$
	$\mathtt{let}\ x \otimes y = t \otimes u\ \mathtt{in}\ v\ =_\beta\ v[t/x][u/y]$	$\mathtt{let}\ () = ()\ \mathtt{in}\ t\ =_\beta\ t$
η:	$\lambda x.t\ x\ =_\eta\ t$	$\lambda^{!}x.t\ x\ =_\eta\ t$
	$\mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u[x \otimes y/z]\ =_\eta\ u[t/z]$	$\langle \pi_1(t), \pi_2(t)\rangle\ =_\eta\ t$
	$\mathtt{let}\ x \otimes y = t\ \mathtt{in}\ v[u/z]\ =_\eta\ v[\mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u/z]$	$x\ =_\eta\ \langle\rangle$
	$\mathtt{let}\ () = t\ \mathtt{in}\ u[()/z]\ =_\eta\ u[t/z]$	
	$\mathtt{let}\ () = t\ \mathtt{in}\ v[u/z]\ =_\eta\ v[\mathtt{let}\ () = t\ \mathtt{in}\ u/z]$	
	$\mathsf{case}(t, x.u[\mathsf{in}_1(x)/z], y.u[\mathsf{in}_2(y)/z])\ =_\eta\ u[t/z]$	$\mathsf{abort}(t)\ =_\eta\ u$

FIGURE 5.1.2. Equations for $\lambda\ell^{\oplus\&}$-terms (relating terms that have matching types).

Remark 5.1.3. For any derivable typing judgment $\Psi;\ \Delta \vdash t : \tau$, if the types occurring in Ψ and Δ, as well as τ, are purely linear, then t is $\beta\eta$-equivalent to some purely linear $\lambda\ell^{\oplus\&}$-term. (This is a consequence of normalization.)

Finally, we say as usual that a $\lambda\ell^{\oplus\&}$-term is *closed* when it does not contain any free variables, that is, when it is typed in the empty context.

5.1.2. **Church encodings of strings and trees.** In order to discuss string-to-string (and tree-to-tree) functions in the $\lambda\ell^{\oplus\&}$-calculus, we need to discuss how they are encoded. Recall from Section 1.1.6 that *Church encodings* provide a canonical way to encode algebraic data types in the simply typed λ-calculus. For instance, for natural numbers and strings over $\{a, b\}$, writing $\underline{w}$ for the Church encoding of w, we have

$$\mathtt{Nat}^! \ = \ (\mathbb{o} \to \mathbb{o}) \to \mathbb{o} \to \mathbb{o} \qquad\qquad \mathtt{Str}^!_{\{a,b\}} \ = \ (\mathbb{o} \to \mathbb{o}) \to (\mathbb{o} \to \mathbb{o}) \to \mathbb{o} \to \mathbb{o}$$

$$\underline{3} = \lambda^! s.\lambda^! z.s\ (s\ (s\ z)) \qquad\qquad \underline{aab} = \lambda^! a.\lambda^! b.\lambda^! \varepsilon.a\ (a\ (b\ \varepsilon))$$

Conversely, one may show that any closed simply typed λ-term of type $\mathtt{Nat}^!$ (resp. $\mathtt{Str}^!_{\{a,b\}}$) is $\beta\eta$-equivalent to the Church encoding of some number (resp. string). In the rest of this dissertation, we will use a less common, but more precise $\lambda\ell^{\oplus\&}$-type for Church encodings of strings of trees, first introduced in [Gir87, §5.3.3].

Definition 5.1.4. Let Σ be an alphabet. As already announced in §1.2.2, we define $\mathtt{Str}_\Sigma$ as $(\mathbb{o} \multimap \mathbb{o}) \to \ldots \to (\mathbb{o} \multimap \mathbb{o}) \to \mathbb{o} \to \mathbb{o}$ where there are $|\Sigma|$ occurrences of $\mathbb{o} \multimap \mathbb{o}$. Note in particular that thanks to the isomorphism[1] $(A \,\&\, B) \to C \ \cong \ A \to B \to C$ (non-linear currying), we have

$$\mathtt{Str}_\Sigma \ \cong \ \left(\mathop{\&}_{a\in\Sigma} (\mathbb{o} \multimap \mathbb{o}) \right) \to \mathbb{o} \to \mathbb{o}$$

It can be checked that $\mathtt{Str}_\Sigma$ has the same closed inhabitants (up to $\beta\eta$-equivalence) as the usual $\mathtt{Str}^!_\Sigma$ presented above, but one should keep in mind that this choice is not entirely innocuous. It is in large part motivated by our main results, which might no longer hold when taking $\mathtt{Str}^!_\Sigma$ instead of $\mathtt{Str}_\Sigma$.

This situation generalizes to trees. For instance, the Church encoding of the tree depicted in Figure 2.1.1 is $\lambda^! a.\lambda^! b.\lambda^! c.\ a\ (a\ c\ (b\ c))\ c \ : \ (\mathbb{o} \to \mathbb{o} \to \mathbb{o}) \to (\mathbb{o} \to \mathbb{o}) \to \mathbb{o} \to \mathbb{o}$.

Definition 5.1.5. Given a ranked alphabet $\mathbf{\Sigma} = (\Sigma, \mathrm{ar})$, the $\lambda\ell^{\oplus\&}$ type $\mathtt{Tree}_{\mathbf{\Sigma}}$ is defined as

$$\mathtt{Tree}_{\mathbf{\Sigma}} \ = \ (\mathbb{o} \multimap \ldots \multimap \mathbb{o}) \to \ldots \to (\mathbb{o} \multimap \ldots \multimap \mathbb{o}) \to \mathbb{o}$$

where there are $|\Sigma|$ top-level arguments, and, within the component corresponding to the letter $a \in \Sigma$, there are $|\mathrm{ar}(a)|$. In other words, we have the isomorphism

$$\mathtt{Tree}_{\mathbf{\Sigma}} \ \cong \ \left(\mathop{\&}_{a\in\Sigma} (\mathbb{o}^{\otimes \mathrm{ar}(a)} \multimap \mathbb{o}) \right) \to \mathbb{o}$$

Remark 5.1.6. The isomorphism of Remark 2.1.1 translates to an equality $\mathtt{Str}_\Sigma = \mathtt{Tree}_{\overline{\Sigma}}$.

The fundamental property of Church encodings is that they induce not just a map from trees in $\mathbf{Tree}(\mathbf{\Sigma})$ to $\lambda\ell^{\oplus\&}$-terms of type $\mathtt{Tree}_{\mathbf{\Sigma}}$ in the empty context, but a *bijective* correspondence when the latter are considered up to $\beta\eta$-equivalence:

Proposition 5.1.7. *Fix a ranked alphabet $\mathbf{\Sigma}$. The map $t \mapsto \underline{t}$ taking trees $t \in \mathbf{Tree}(\mathbf{\Sigma})$ to their Church encodings modulo $=_{\beta\eta}$ is a bijection.*

[1] We keep this notion informal, but suffices to say that this is intended to be definable internally to $\lambda\ell^{\oplus\&}$.

$$
\begin{array}{l}
\delta \;=\; \lambda a\; z.\; \mathsf{let}\; (b, z') \;= z\; \mathsf{in} \\
\qquad\qquad \mathsf{let}\; (x, y) \;= z'\; \mathsf{in} \\
\qquad\qquad \mathsf{if}\; b\; \mathsf{then} \\
\qquad\qquad\quad (\mathtt{tt}, \langle \lambda u.\; \pi_1(x)\; (a\; u), \mathtt{let}\; () = \pi_2(y)\; \mathtt{in}\; \pi_2(x)\rangle, y) \\
\qquad\qquad \mathsf{else} \\
\qquad\qquad\quad (\mathtt{ff}, \langle \lambda u.\; a\; (\pi_1(x)\; u), \mathtt{let}\; () = \pi_2(y)\; \mathtt{in}\; \pi_2(x)\rangle, y)
\end{array}
$$

$$
\begin{array}{l}
\delta_{\|} \;=\; \lambda z.\; \mathsf{let}\; (b, z') \;= z\; \mathsf{in} \\
\qquad\qquad \mathsf{let}\; (x, y) \;= z'\; \mathsf{in} \\
\qquad\qquad \mathsf{if}\; b\; \mathsf{then} \\
\qquad\qquad\quad (\mathtt{ff}, \langle \lambda u.u, \mathtt{let}\; () = \pi_2(y)\; \mathtt{in}\; \pi_2(x)\rangle, \langle \lambda v.\pi_1(y)\; (\pi_1(x)\; u), \mathtt{let}\; () = \pi_1(x)\; \mathtt{in}\; \pi_2(y)\rangle) \\
\qquad\qquad \mathsf{else} \\
\qquad\qquad\quad (\mathtt{tt}, \langle \lambda u.u, \mathtt{let}\; () = \pi_2(y)\; \mathtt{in}\; \pi_2(x)\rangle, \langle \lambda v.\pi_1(x)\; (\pi_1(y)\; u), \mathtt{let}\; () = \pi_1(x)\; \mathtt{in}\; \pi_2(y)\rangle)
\end{array}
$$

$$
\begin{array}{l}
o \;=\; \lambda\varepsilon\; z.\; \mathsf{let}\; (b, z') \;= z\; (\mathtt{tt}, \langle \lambda u.u, ()\rangle, \langle \lambda u.u\rangle)\; \mathsf{in} \\
\qquad\qquad \mathsf{let}\; (x, y) \;= z'\; \mathsf{in} \\
\qquad\qquad \mathsf{let}\; () \;= \pi_2(x)\; \mathsf{in} \\
\qquad\qquad \mathsf{if}\; b\; \mathsf{then} \\
\qquad\qquad\quad \pi_1(y)\; \varepsilon \\
\qquad\qquad \mathsf{else} \\
\qquad\qquad\quad \mathtt{let}\; () = \pi_2(y)\; \mathtt{in}\; \varepsilon
\end{array}
$$

FIGURE 5.1.3. Auxiliary $\lambda\ell^{\oplus\&}$-terms for Example 5.1.12, where $\mathtt{tt} = \mathsf{in}_1(())$, $\mathtt{ff} = \mathsf{in}_2(())$, $\mathsf{if}\; t\; \mathsf{then}\; u\; \mathsf{else}\; v = \mathsf{case}(t, x.\mathtt{let}\; () = x\; \mathtt{in}\; u, y.\mathtt{let}\; () = y\; \mathtt{in}\; v)$.

The surjectivity part of this statement is a consequence of normalization. For this reason, we shall prove this in Section 5.4.3 after our normalization result.

5.1.3. **Two ways to define functions over Church encodings.** We are now ready to give our notions of computation for our string (and tree) functions. First, we need an operation of type substitution in $\lambda\ell^{\oplus\&}$, which allow to substitute an arbitrary type κ for $\mathbb{o}$.

$$\mathbb{o}[\kappa] = \kappa \qquad (\tau \multimap \sigma)[\kappa] = \tau[\kappa] \multimap \sigma[\kappa] \qquad \ldots$$

(This already made an appearance in Chapter 1, in particular §1.1.6.)

Remark 5.1.8. Type substitution extends in the obvious way to typing contexts as well, and even to *typing derivations*, so that

$$\Psi;\; \Delta \vdash t : \tau \qquad \Rightarrow \qquad \Psi[\kappa];\; \Delta[\kappa] \vdash t : \tau[\kappa]$$

In particular, it means that every $t : \mathtt{Tree}_{\mathbf{\Sigma}}$ is also of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]$ for any type κ.

This remark ensures that the following notion of definable tree functions (strings being a special case) in the $\lambda\ell^{\oplus\&}$-calculus makes sense.

Definition 5.1.9. A function $f : \mathbf{Tree}(\mathbf{\Sigma}) \to \mathbf{Tree}(\mathbf{\Gamma})$ is called $\lambda\ell^{\oplus\&}$-*definable* when there exists a *purely linear* type κ together with a closed $\lambda\ell^{\oplus\&}$-term

$$\mathtt{f} \quad : \quad \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$$

such that f and $\mathtt{f}$ coincide up to Church encoding; i.e., for every tree $t \in \mathbf{Tree}(\mathbf{\Sigma})$

$$\underline{f(t)} \;=_{\beta\eta}\; \mathtt{f}\; \underline{t}$$

In particular, a string function $\Sigma^* \to \Gamma^*$ is $\lambda\ell^{\oplus\&}$-definable when the corresponding unary tree function $\mathbf{Tree}(\overline{\Sigma}) \to \mathbf{Tree}(\overline{\Gamma})$ (cf. Remark 2.1.1) is. Note that by Remark 5.1.6,

$$\mathtt{Tree}_{\overline{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\overline{\Gamma}} \quad = \quad \mathtt{Str}_{\Sigma}[\kappa] \multimap \mathtt{Str}_{\Gamma}$$

Remark 5.1.10. Proposition 5.1.7 has an important consequence: *every closed $\lambda\ell^{\oplus\&}$-term of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$ defines some function $\mathbf{Tree}(\mathbf{\Sigma}) \to \mathbf{Tree}(\mathbf{\Gamma})$.*

We will give another notion of definable string function in the $\lambda\ell^{\oplus\&}$-calculus soon, but first, let us give some concrete examples of $\lambda\ell^{\oplus\&}$-definable functions. They should illustrate a claim from Theorem 1.2.3 (that we will establish later): over both strings and trees, $\lambda\ell^{\oplus\&}$-definable functions coincide with regular functions.

Example 5.1.11. The `reverse` function $\Sigma^* \to \Sigma^*$ is $\lambda\ell^{\oplus\&}$-definable. (It was used previously for instance to define the "iterated reverse" in Example 2.3.7.) Supposing that we have $\Sigma = \{a_1, \dots, a_k\}$, one $\lambda\ell^{\oplus\&}$-term that implements it is

$$\lambda s.\lambda^! a_1. \dots \lambda^! a_k.\lambda^! \varepsilon.\ s\ (\lambda x.\lambda z.x\ (a_1\ z)) \dots (\lambda x.\ (a_k\ z))\ (\lambda x.x)\ \varepsilon\ :\ \mathtt{Str}_{\Sigma}[\mathbb{o} \multimap \mathbb{o}] \multimap \mathtt{Str}_{\Sigma}$$

Example 5.1.12. The copyless SST of Figure 2.3.1 is computed by a $\lambda\ell^{\oplus\&}$-term of type $\mathtt{Str}_{\Sigma \sqcup \{\|\}}\,[\tau] \multimap \mathtt{Str}_{\Sigma}$ with $\tau = \mathtt{Bool} \otimes ((\mathbb{o} \multimap \mathbb{o}) \,\&\, \mathbf{I}) \otimes ((\mathbb{o} \multimap \mathbb{o}) \,\&\, \mathbf{I})$. Intuitively, `Bool` corresponds to the current state of the SST while each component $(\mathbb{o} \multimap \mathbb{o}) \,\&\, \mathbf{I}$ corresponds to a register. Supposing that we have $\Sigma = \{a_1, \dots, a_k\}$, and that the letter $\|$ corresponds to the first constructor in the input string, the $\lambda\ell^{\oplus\&}$-definition is given by

$$\lambda s.\lambda^! a_1. \dots \lambda^! a_k.\lambda^! \varepsilon.\ o\ (s\ \delta_{\|}\ (\delta\ a_1)\ \dots\ (\delta\ a_k))$$

where the terms $\delta : (\mathbb{o} \multimap \mathbb{o}) \multimap \tau \multimap \tau$, $\delta_{\|} : \mathbb{o} \multimap \tau \multimap \tau$ and $o : \mathbb{o} \multimap (\tau \multimap \tau) \multimap \mathbb{o}$ are defined in Figure 5.1.3.

Example 5.1.13. Consider the ranked alphabet $\mathbf{\Sigma} = \{a : 2, b : 2, c : \varnothing\}$ (where $2 = \{\triangleleft, \triangleright\}$) and the alphabet $\Gamma = \{a, b, c\}$. The conditional swap of Example 2.6.1 is $\lambda\ell^{\oplus\&}$-definable as a term of type $\mathtt{Tree}_{\mathbf{\Sigma}}[(\mathbb{o} \multimap \mathbb{o}) \,\&\, (\mathbb{o} \multimap \mathbb{o})] \to \mathtt{Str}_{\Gamma}$ reminiscent of the BRTT given in Example 2.6.3. Observe the use of an *additive conjunction* '&' (that is not of the form $(- \,\&\, \mathbf{I})$ meant to make data discardable), reflecting the fact that this BRTT is single-use-restricted but not copyless. To wit, setting $\tau = (\mathbb{o} \multimap \mathbb{o}) \,\&\, (\mathbb{o} \multimap \mathbb{o})$ and assuming free variables $a, b : \mathbb{o} \multimap \mathbb{o}$, define the auxiliary terms

$$\begin{array}{rcll} \delta_a & = & \lambda l.\lambda r.\ \langle \pi_1(l) \circ a \circ \pi_1(r), \pi_1(r) \circ a \circ \pi_1(l) \rangle & : \tau \multimap \tau \multimap \mathbb{o} \multimap \mathbb{o} \\ \delta_b & = & \lambda l.\lambda r.\ (\lambda x.\ \langle x, x \rangle)\ (\pi_1(l) \circ b \circ \pi_1(r)) & : \tau \multimap \tau \multimap \mathbb{o} \multimap \mathbb{o} \end{array}$$

where $f \circ g$ stands for the composition $\lambda z.\ f\ (g\ z)$. The conditional swap is then coded as

$$\lambda t.\lambda^! a.\lambda^! b.\lambda^! c.\lambda^! \varepsilon.\ \pi_2\ (t\ \delta_a\ \delta_b\ (\lambda x.\ c\ x))\ \varepsilon$$

Remark 5.1.14. Once again, our set-up, summarized in Definition 5.1.9, is biased toward characterizing regular functions; there are many non-equivalent alternatives which also make perfect sense. For instance, changing the following would be reasonable:

- allow κ to be be arbitrary (i.e. to contain '$\to$') or with some restrictions;
- change the type of Church encodings (recall the distinction $\mathtt{Str}^!_{\Sigma}/\mathtt{Str}_{\Sigma}$).

These alternatives share many good structural properties with our choice. Giving more automata-theoretic characterizations for those and comparing them lies beyond the scope of this dissertation, but would be interesting. (Let us mention that the pure linearity constraint on the type κ actually has a clear operational meaning: as mentioned in the introduction, it corresponds to single-use-restricted assignments.)

Another parameter that can be tweaked, and for which we *do* investigate the effects of the change, is the use of the linear arrow $\multimap$ at the top level of the type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$. Morally, this corresponds to the fact that a streaming string transducer or bottom-up tree transducer *traverses its input in a single pass*. We shall consider replacing $\multimap$ by the non-linear function arrow $\to$ in the case of strings:

Definition 5.1.15. A function $f : \Sigma^* \to \Gamma^*$ is called $(\lambda\ell^{\oplus\&}, \to)$-*definable* when there exists a *purely linear* type κ together with a closed $\lambda\ell^{\oplus\&}$-term

$$\mathtt{f} \quad : \quad \mathtt{Str}_{\Sigma}[\kappa] \to \mathtt{Str}_{\Gamma}$$

such that for every string $s \in \Sigma^*$,

$$\underline{f(s)} \ =_{\beta\eta} \ \mathtt{f}\ \underline{s}$$

According to Theorem 1.2.3, this will provide our characterization of comparison-free polyregular functions. But even without knowing their relationship to automata theory yet, we can already deduce the following property of the above-defined function classes:

Proposition 5.1.16. *The class of $\lambda\ell^{\oplus\&}$-definable is closed under composition, and so is the class of $(\lambda\ell^{\oplus\&}, \to)$-definable functions.*

Proof. We prove the case of $\lambda\ell^{\oplus\&}$-definitions; for $(\lambda\ell^{\oplus\&}, \to)$-definitions, the same argument where $\multimap$ is replaced by $\to$ everywhere applies.

Let $t_1 : \mathtt{Tree}_{\mathbf{\Gamma}}[\kappa_1] \multimap \mathtt{Tree}_{\mathbf{\Pi}}$ and $t_2 : \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa_2] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$ where κ_1 and κ_2 are both purely linear. Using Remark 5.1.8, we then have $t_2 : (\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa_2] \multimap \mathtt{Tree}_{\mathbf{\Gamma}})[\kappa_1]$ and the result of this type substitution is equal to $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa_2[\kappa_1]] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}[\kappa_1]$. Therefore, $u = \lambda x.\ t_1\ (t_2\ x)$ is a well-typed $\lambda\ell^{\oplus\&}$-term, with type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa_2[\kappa_1]] \multimap \mathtt{Tree}_{\mathbf{\Pi}}$. The function computed by u is $\lambda\ell^{\oplus\&}$-definable by definition since $\kappa_2[\kappa_1]$ is purely linear, and it is the composition of those computed by the $\lambda\ell^{\oplus\&}$-terms t_1 and t_2. □

5.2. Regularity equals $\lambda\ell^{\oplus\&}$-definability

Now we prove the first row of Theorem 1.2.3.

5.2.1. **The syntactic category $\mathcal{L}$ of purely linear $\lambda\ell^{\oplus\&}$-terms.** First, we relate our notion of generalized SSTs from Chapter 4 to the $\lambda\ell^{\oplus\&}$-calculus. If $\Gamma = \{b_1, \dots, b_n\}$ is an alphabet, call $\widetilde{\Gamma}$ the non-linear typing context

$$\widetilde{\Gamma} = (b_1 : \mathbb{0} \multimap \mathbb{0},\ \dots,\ b_n : \mathbb{0} \multimap \mathbb{0},\ \varepsilon : \mathbb{0})$$

Definition 5.2.1. We call $\mathcal{L}(\widetilde{\Gamma})$ (or just $\mathcal{L}$ when Γ is clear from the context) the category

- whose objects are purely linear types;
- whose morphisms from τ to σ are $\lambda\ell^{\oplus\&}$-terms t such that $\widetilde{\Gamma};\ \cdot \vdash t : \tau \multimap \sigma$, considered *up to $\beta\eta$-equivalence*;
- whose identity is given by $\lambda x.x$ and composition of f and g by $\lambda x.\ f\ (g\ x)$.

Remark 5.2.2. In the definition of morphisms, represented by $\lambda\ell^{\oplus\&}$-terms, we only impose the pure linearity restriction on the types of the terms. Because $\lambda\ell^{\oplus\&}$ is normalizing (Theorem 5.4.1), we could have further assumed the $\lambda\ell^{\oplus\&}$-terms to be normal and thus, to only contain subterms whose types are also purely linear (cf. Remark 5.1.3). Therefore, it makes sense to say that $\mathcal{L}(\widetilde{\Gamma})$ is about the *purely linear fragment of* $\lambda\ell^{\oplus\&}$ augmented with (inert) constants from $\widetilde{\Gamma}$.

Similarly, for the equivalence relation we use to actually define the homsets, we use the full $\beta\eta$-equivalence, which could, on the face of it, require to go oustide of the purely linear fragment of $\lambda\ell^{\oplus\&}$ to establish certain equalities (for instance, consider the rather artificial

derivation $\lambda x.x =_{\beta\eta} \lambda x.(\lambda^! y.y)\ x =_{\beta\eta} \lambda x.x$). This is also unnecessary as it can be checked that the normalization argument relies on a reduction relation $\to_{\beta\varepsilon}$ which is confluent up to commutative conversions $\approx_c$ (Theorem 5.4.4); since both $\to_{\beta\varepsilon}$ and $\approx_c$ preserve the purely linear fragment, this is enough to conclude that we could ignore non-purely linear terms when defining $=_{\beta\eta}$ for the purely linear fragment. (This is a kind of *conservativity* property.)

$\mathcal{L}$ is a monoidal closed category with products and coproducts, which captures the expressiveness of purely linear $\lambda\ell^{\oplus\&}$-terms enriched with constants for the "empty word" and prepending letters of Γ to the left of a "word" when regarding the type $\mathbb{o}$ being regarded as the type of such words. This leads to the expected notion of streaming setting (cf. §4.2).

Definition 5.2.3. $\mathfrak{L}$ is the streaming setting $(\mathcal{L}, \mathbf{I}, \mathbb{o}, (\!|-|\!)_{\mathfrak{L}})$ with output Γ^* such that $(\!|t|\!)_{\mathfrak{L}} = w$ if and only if $\lambda^! b_1 \ldots \lambda^! b_n.\ \lambda^! \varepsilon.\ t\ ()$ is $\beta\eta$-equivalent to the Church encoding of w (this defines a total function because of Proposition 5.1.7).

Our interest in $\mathfrak{L}$ lies in the following lemma. (Recall that a $\mathfrak{C}$-SST is said to be *single-state* if its set of states is a singleton.)

Lemma 5.2.4. *A function $\Sigma^* \to \Gamma^*$ is computable by a single-state $\mathfrak{L}$-SSTs if and only if it is $\lambda\ell^{\oplus\&}$-definable in the sense of Definition 5.1.9.*

To prove one direction of this equivalence, we need a technical lemma on $\lambda\ell^{\oplus\&}$-terms defining string functions. In order to state the lemma in the more general case of tree functions, so that it may be reused in Chapter 6, we extend the notation $\widetilde{\Gamma}$ above as follows: given a ranked alphabet $\mathbf{\Sigma} = \{a_1 : A_1,\ \ldots,\ a_n : A_n\}$ (recall the notation from §2.1), let

$$\widetilde{\mathbf{\Sigma}} = (a_1 : \mathbb{o} \multimap \ldots \multimap \mathbb{o},\ \ldots,\ a_n : \mathbb{o} \multimap \ldots \multimap \mathbb{o})$$

where the type of a_i has $|A_i|$ arguments (thus, it contains $|A_i| + 1$ occurrences of $\mathbb{o}$).

Lemma 5.2.5. *Let $\mathbf{\Sigma} = \{a_1 : A_1, \ldots, a_n : A_n\}$ and $\mathbf{\Gamma} = \{b_1 : B_1, \ldots, b_k : B_k\}$ be ranked alphabets such that there is some $A_i = \varnothing$ (i.e., $\mathbf{Tree}(\mathbf{\Sigma}) \neq \varnothing$). Up to $\beta\eta$-equivalence, every term of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$ is of the shape*

$$\lambda s.\lambda^! b_1 \ldots \lambda^! b_k.\ o\ (s\ d_1\ \ldots\ d_n)$$

such that o and the d_i are purely linear $\lambda\ell^{\oplus\&}$-terms with no occurrence of s. In particular:

$$\widetilde{\mathbf{\Gamma}}; \cdot \vdash o : \kappa \multimap \mathbb{o} \qquad\qquad \widetilde{\mathbf{\Gamma}}; \cdot \vdash d_i : \kappa \multimap \ldots \multimap \kappa$$

(with the type of d_i having $|B_i|$ arguments).

The proof, by case analysis of the $\lambda\ell^{\oplus\&}$-terms *in normal form* of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$, will be provided in Section 5.4.4. For now, let us use this to establish the equivalence between $\lambda\ell^{\oplus\&}$-definability and $\mathfrak{L}$-SSTs.

Proof of Lemma 5.2.4. Before beginning the proof, it should be noted that SSTs process strings from left to right while Church encodings work rather from right to left. This is not a big issue in the presence of higher-order functions.

$\big(\mathfrak{L}\text{-SST} \subseteq \lambda\ell^{\oplus\&}\big)$ Given a $\mathfrak{L}$-SST $\mathcal{T} = (\{*\}, *, \tau, \delta, i, o)$, δ may be regarded as family of $\lambda\ell^{\oplus\&}$ terms $(t_a)_{a\in\Sigma}$ (with free variables in $\widetilde{\Gamma}$). Suppose that $\Sigma = \{a_1, \ldots, a_k\}$ and recall that Example 5.1.11 provides a $\lambda\ell^{\oplus\&}$-term $\mathtt{rev} : \mathtt{Str}_\Sigma[\mathbb{o} \multimap \mathbb{o}] \multimap \mathtt{Str}_\Sigma$ implementing the reversal of its input string. $[\![\mathcal{T}]\!]$ is implemented by the following $\lambda\ell^{\oplus\&}$-term:

$$\lambda s.\ \lambda^! b_1.\ \ldots\ \lambda^! b_n.\ \lambda^! \varepsilon.\ o\ (\mathtt{rev}\ s\ t_{a_1} \ldots t_{a_k}\ (i\ ())) \quad : \quad \mathtt{Str}_\Sigma[\tau \multimap \tau] \multimap \mathtt{Str}_\Gamma$$

$\big(\lambda\ell^{\oplus\&} \subseteq \mathfrak{L}\text{-SST}\big)$ Any $\lambda\ell^{\oplus\&}$-term of type $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{Str}_\Gamma$ is $\beta\eta$-equivalent to

$$\lambda s.\ \lambda^! b_1.\ \ldots\ \lambda^! b_n.\ \lambda^! \varepsilon.\ t\ (s\ u_1\ \ldots\ u_k\ v)$$

by Lemma 5.2.5, where t, v and the u_i are some terms typable in $\widetilde{\Gamma}$. The $\mathfrak{L}$-SST

$$\mathcal{T} = (\{*\},\, *,\, \tau \multimap \tau,\, \delta,\, \lambda x.\, x,\, \lambda f.\, o\ (f\ i)) \quad \text{where} \quad \delta(a_i, *) = (*,\, \lambda g.\lambda x.\, a_i\ (g\ x))$$

computes the same string function. □

Lemma 5.2.4 therefore enables us to reframe the part of Theorem 1.2.3 about regular functions as a statement comparing the expressiveness of single-state $\mathfrak{L}$-SSTs and stateful $\mathfrak{SR}$-SSTs (indeed, the latter correspond to usual copyless SSTs which compute regular function, cf. Section 4.2.1). This motivates our abstract development focused on comparing the expressiveness of various $\mathfrak{C}$-SSTs. In order to build morphisms from $\mathfrak{L}$ to other streaming settings, we shall make use of the following fundamental property.

Lemma 5.2.6. *Let $\mathfrak{C}$ be a streaming setting $(\mathcal{C}, \top, \bot\!\!\bot, (\!|-|\!))$ with output Γ^* whose underlying category $\mathcal{C}$ is symmetric monoidal closed with $\top = \mathbf{I}$ and has finite products and coproducts. Let $(f_b)_{b\in\Gamma} \in \mathsf{Hom}_{\mathcal{C}}(\bot\!\!\bot, \bot\!\!\bot)^{\Gamma}$ and $e \in \mathsf{Hom}_{\mathcal{C}}(\top, \bot\!\!\bot)$ be such that $(\!|e|\!) = \varepsilon$ and, for every $g \in \mathsf{Hom}_{\mathcal{C}}(\top, \bot\!\!\bot)$, we have $(\!|f_b \circ g|\!) = b(\!|g|\!)$ (that is, f_b acts by concatenating the single-letter word b on the left).*

Then there is a canonical morphism $[\![-]\!] : \mathfrak{L} \to \mathfrak{C}$ of streaming settings such that $[\![\mathbb{o}]\!] = \bot\!\!\bot$. Moreover the underlying functor is strong monoidal for $\oplus$, $\&$, $\otimes$ and preserves $\multimap$.

Without spelling out the details, this lemma essentially states that $\mathcal{L}$ is *initial* among symmetric monoidal closed categories with products and coproducts (thought the uniqueness part of initiality has not been included in our statement). We do not offer a proof of this lemma, which we consider folklore. The interested reader may find a detailed treatment in [Bie94, Chapter 4] for the case of $\mathcal{L}(\varnothing)$ (i.e., where the $\lambda\ell^{\oplus\&}$-terms have no free variables). Let us note that, because of the specific way we defined $\mathcal{L}$, the first step in this proof should invoke the conservativity of the congruence $=_{\beta\eta}$ over the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus (Remark 5.2.2). This is because the notion of symmetric monoidal closed category does not require the existence of an exponential modality '!' (that would allow encoding $\tau \to \sigma$ as $!\tau \multimap \sigma$), and thus, all the equations in the initial symmetic monoidal closed category with products and coproducts should only satisfy those equations mentioning those constructs.

5.2.2. **$\mathfrak{L}$-SSTs compute regular functions.** We can now give the proof of the first implicit characterization stated in Theorem 1.2.3. Thanks to Lemma 5.2.4 this amounts to the equivalence between single-state $\mathfrak{L}$-SST and regular functions.

The technically more difficult part is soundness: every single-state $\mathfrak{L}$-SST computes a regular function. But almost all the work has been done for this already. According to §4.5.2, single-state $\mathfrak{SR}_{\oplus\&}$-SSTs compute regular functions. Thanks to Lemma 4.2.8, this means we just have to build a morphism of streaming settings $\mathfrak{L} \to \mathfrak{SR}_{\oplus\&}$. This is done by combining the initiality of the syntactic category (as stated for our specific purposes in Lemma 5.2.6) with the fact that $\mathcal{SR}_{\oplus\&}$ is monoidal closed (Corollary 4.5.4).

For the converse inclusion, we use the characterization of regular functions by stateful $\mathfrak{SR}$-SSTs. Since $\mathcal{L}$ has coproducts (given by the $\oplus$ type constructor), $\mathfrak{L}$-SSTs are equivalent to single-state $\mathfrak{L}$-SSTs. We then conclude using the following result.

Lemma 5.2.7. *There is a morphism of streaming settings $\mathfrak{SR} \to \mathfrak{L}$.*

We build this morphism using the generic construction of Corollary 4.6.3 (which has already been applied in Section 4.7.). This is not strictly necessary; see Lemma 6.7.2 for a sketch of a "manual" definition of such a morphism in the case of trees. In fact, to prove that all $\lambda\ell^{\oplus\&}$-definable string functions are regular, the level of abstraction that we are working with is largely overkill: it would suffice to encode copyless SSTs as $\lambda\ell^{\oplus\&}$-terms, a programming exercise that is not particularly difficult.

PROOF OF LEMMA 5.2.7. We first give the construction of the underlying functor. First, note that $\mathcal{L}$ has all cartesian products and internal homsets, given by the syntactic connectives on types with the same notations. Thus, it is symmetric monoidal closed and quasi-affine; we can therefore apply Corollary 4.6.3 to the base type $\mathbb{o}$, regarded as an object of $\mathcal{L}$, and to the family of endomorphisms $(\widetilde{\Gamma};\ \cdot \vdash c : \mathbb{o} \multimap \mathbb{o})_{c\in\Gamma} \in \mathsf{Hom}_{\mathcal{L}}(\mathbb{o},\mathbb{o})^{\Gamma}$. (Indeed, by definition, every letter $c \in \Gamma$ serves as a variable c given the type $\mathbb{o} \multimap \mathbb{o}$ in the typing context $\widetilde{\Gamma}$.) This gives us $F : \mathcal{SR} \to \mathcal{L}$ such that $F(\varnothing) = \mathbf{I}$, $F(\{\bullet\}) = (\mathbb{o} \multimap \mathbb{o}) \,\&\, \mathbf{I}$ and

$$\forall w \in \Gamma^*, \quad F(\widehat{w}) \ =_{\beta\eta}\ \lambda z.\, \mathtt{let}\ () = z\ \mathtt{in}\ \langle (\lambda x.\, w_1\,(w_2\,\ldots(w_n\,x)\ldots)),\ ()\rangle$$

Since $\mathbb{T}_{\mathfrak{GR}} = \varnothing$ and $\mathbb{T}_{\mathfrak{L}} = \mathbf{I}$, we can simply take $i = \mathrm{id}_{\mathbf{I}} : \mathbb{T}_{\mathfrak{L}} \to F(\mathbb{T}_{\mathfrak{GR}})$ as part of our morphism of streaming settings. The map $o : F(\perp\!\!\!\perp_{\mathfrak{GR}}) \to \perp\!\!\!\perp_{\mathfrak{L}}$ is more interesting. Since $\perp\!\!\!\perp_{\mathfrak{GR}} = \{\bullet\}$ and $\perp\!\!\!\perp_{\mathfrak{L}} = \mathbb{o}$, we can see that o must be a $\lambda\ell^{\oplus\&}$-term of type $((\mathbb{o} \multimap \mathbb{o}) \,\&\, \mathbf{I}) \multimap \mathbb{o}$ in the context $\widetilde{\Gamma}$. The choice that works is $o = \lambda p.\, \pi_1(p)\ \varepsilon$ (recalling that $(\varepsilon : \mathbb{o}) \in \widetilde{\Gamma}$ stands for the empty string). Proving that $(\!|o \circ F(\widehat{w}) \circ i|\!)_{\mathfrak{L}} = (\!|\widehat{w}|\!)_{\mathfrak{GR}}$ for any $w \in \Gamma^*$ is merely a matter of performing $\beta\eta$-conversions starting from the above equation on $F(\widehat{w})$; we leave this to the reader. Since every $f \in \mathsf{Hom}_{\mathcal{SR}}(\mathbb{T}, \perp\!\!\!\perp)$ is of the form $f = \widehat{w}$ for some $w \in \Gamma^*$, this suffices to show that (F, i, o) fits the definition of a morphism of streaming settings. □

5.3. COMPARISON-FREE POLYREGULARITY EQUALS $(\lambda\ell^{\oplus\&}, \to)$-DEFINABILITY

Having just proved the first result on strings claimed in Theorem 1.2.3, we now move on to the second one.

5.3.1. **Extensional completeness.** The easier direction, i.e., the fact that every comparison-free polyregular function is $(\lambda\ell^{\oplus\&}, \to)$-definable (Definition 5.1.15), is established via the definition of the former as the smallest class of string-to-string functions containing all regular functions and closed under composition by substitution (Definition 3.1.2):

- As a consequence of what we just saw in the previous section, every regular function is $(\lambda\ell^{\oplus\&}, \to)$-definable. Indeed, the function computed by $t : \mathtt{Str}_\Sigma[\tau] \multimap \mathtt{Str}_\Pi$ is also implemented by $\lambda^!x.\, t\ x : \mathtt{Str}_\Sigma[\tau] \to \mathtt{Str}_\Pi$.
- The class of $(\lambda\ell^{\oplus\&}, \to)$-definable functions is closed under CbS, as the lemma below states.

Lemma 5.3.1. *Let $f : \Sigma^* \to I^*$ be computed by $t : \mathtt{Str}_\Sigma[\tau] \to \mathtt{Str}_I$ and $g_i : \Sigma^* \to \Pi^*$ be defined by $u_i : \mathtt{Str}_\Sigma[\sigma_i] \to \mathtt{Str}_\Pi$ for each $i \in I$. Assume that τ and every σ_i are purely linear types. Then $\mathrm{CbS}(f, (g_i)_{i\in I})$ is computed by some $\lambda\ell^{\oplus\&}$-term of type $\mathtt{Str}_\Sigma[\kappa] \to \mathtt{Str}_\Pi$ where κ is purely linear. In other words,*

PROOF. Let the type κ and the following $\lambda\ell^{\oplus\&}$-terms be given by Lemma 5.3.2 below:

$$\mathtt{cast}_0 : \mathtt{Str}_\Sigma[\kappa] \multimap \mathtt{Str}_\Sigma[\tau] \qquad \mathtt{cast}_i : \mathtt{Str}_\Sigma[\kappa] \multimap \mathtt{Str}_\Sigma[\sigma_i[\mathbb{o} \multimap \mathbb{o}]] \quad \text{for } i \in I$$

By Remark 5.1.8, we have $u_i : \mathtt{Str}_\Sigma[\sigma_i[\mathbb{o} \multimap \mathbb{o}]] \to \mathtt{Str}_\Pi[\mathbb{o} \multimap \mathbb{o}]$ for $i \in I$. This allows us to define the following $\lambda\ell^{\oplus\&}$-term that implements $\mathrm{CbS}(f, (g_i)_{i\in I})$:

$$\lambda^!s.\lambda^!h_1 \ldots \lambda^!h_{|\Pi|}.\lambda^!x.\ t\ (\mathtt{cast}_0\ s)\ \hat{u}_1 \ldots\ \hat{u}_{|I|}\ x \quad : \quad \mathtt{Str}_\Sigma[\kappa] \to \mathtt{Str}_\Pi$$

(to make things clear, $h_1, \ldots, h_{|\Pi|} : \mathbb{o} \multimap \mathbb{o}$ and $x : \mathbb{o}$) where, for each $i \in I$, we take

$$\hat{u}_i = u_i\ (\mathtt{cast}_i\ s)\ (\lambda f.\lambda y.\ h_1\ (f\ y))\ \ldots\ (\lambda f.\lambda y.\ h_{|\Pi|}\ (f\ y))\ (\lambda y.\ y)$$

(thus, $s : \mathtt{Str}[\kappa]$, $h_1 : \mathbb{o} \multimap \mathbb{o}$, $\ldots$, $h_{|\Pi|} : \mathbb{o} \multimap \mathbb{o}$; $\cdot \vdash \hat{u}_i : \mathbb{o} \multimap \mathbb{o}$). □

Lemma 5.3.2. *Let Σ be a finite alphabet, J be a finite set of indices, and $(\tau_j)_{j\in J}$ be a family of purely linear types. Then there exist a purely linear type κ and closed $\lambda\ell^{\oplus\&}$-terms $\mathtt{cast}_j : \mathtt{Str}_\Sigma[\kappa] \multimap \mathtt{Str}_\Sigma[\tau_j]$ such that, for every $w \in \Sigma^*$, we have $\mathtt{cast}_j\ \underline{w} =_{\beta\eta} \underline{w}$.*

PROOF. For the sake of simplicity, let us demonstrate the idea for $J = \{1, 2\}$. Take

$$\kappa = ((\tau_1 \multimap \tau_1) \,\&\, \mathbf{I}) \otimes ((\tau_2 \multimap \tau_2) \,\&\, \mathbf{I})$$

and define $\mathtt{cast}_1$ as the $\lambda\ell^{\oplus\&}$-term

$$\lambda^! s.\lambda^! h_1 \ldots \lambda^! h_{|\Sigma|}.\lambda^! x.\ \mathtt{let}\ f \otimes g = s\ u_1\ \ldots\ u_{|\Sigma|}\ (v \otimes v)\ \mathtt{in}\ \mathtt{let}\ () = \pi_2\ g\ \mathtt{in}\ (\pi_1\ f)\ x$$

where $v = \langle (\lambda y.\ y), () \rangle$ and $u_i = \lambda z.\ \mathtt{let}\ f \otimes g = z\ \mathtt{in}\ \langle (\lambda y.\ h_i\ ((\pi_1\ f)\ y)), () \rangle \otimes g$. □

5.3.2. **Outline of a semantic evaluation argument.** We would now like to prove the converse: every $(\lambda\ell^{\oplus\&}, \to)$-definable function is comparison-free polyregular. Again, this will be done using a combination of a syntactic analysis of normal forms for $\lambda\ell^{\oplus\&}$-terms and semantic evaluation in the monoidal closed category $\mathcal{SR}_{\oplus\&}$ (cf. Section 4.5). We start here by presenting the general setup and how the pieces fit together to reach the desired conclusion; the semantic lemmas will be proved in the next subsections, while the syntactic aspects will be properly treated only towards the end of the chapter (§5.4.6).

Let Σ be a finite alphabet. Recall that any $A \in \mathsf{Obj}(\mathcal{SR}(\Sigma)_{\oplus\&})$ has the form

$$A \;=\; \bigoplus_{q \in Q(A)} \;\mathop{\&}_{x \in X(A)_q} R(A)_{q,x}$$

where $\oplus$ and $\&$ do not merely denote (co)products determined only up to isomorphism: the objects of $\mathcal{SR}(\Sigma)_{\oplus\&}$ are *defined* as formal coproducts of products of finite register sets. Therefore, A is uniquely identified by the data $(Q(A), (X(A)_q)_{q \in Q}, (R(A)_{q,x})_{q \in Q, x \in X(A)_q})$, and we shall use the notations $Q(A)$, $X(A)$ and $R(A)$ for the components of this triple in the rest of this section.

Since $\mathcal{SR}(\Sigma)_{\oplus\&}$ is isomorphic to a coproduct completion, it admits a canonical coproduct-preserving forgetful functor to FinSet. We denote this functor by Q since the image of an object A in $\mathcal{SR}_{\oplus\&}$ is none other than the above-defined $Q(A)$. We also write

$$A \to_\Sigma B = \mathsf{Hom}_{\mathcal{SR}_{\oplus\&}(\Sigma)}(A, B) \qquad\qquad \lfloor A \rfloor_\Sigma = \mathbf{I} \to_\Sigma A$$

Proposition 5.3.3. *Using the dependent sum operation on sets (§2.1.1), we have*

$$\forall A \in \mathsf{Obj}(\mathcal{SR}_{\oplus\&}),\quad \lfloor A \rfloor_\Sigma \;\cong\; \sum_{q \in Q(A)} (\Sigma^*)^{R(A)_q} \quad \textit{where} \quad R(A)_q \;=\; \sum_{x \in X(A)_q} R(A)_{q,x}$$

PROOF. First, by definition of $(-)_{\oplus\&}$,

$$\lfloor A \rfloor_\Sigma \;\cong\; \sum_{q \in Q(A)} \prod_{x \in X(A)_q} \mathsf{Hom}_{\mathcal{SR}}(\mathbf{I}, R(A)_{q,x}) \;\cong\; \sum_{q \in Q(A)} \prod_{x \in X(A)_q} (\Sigma^*)^{R(A)_{q,x}}$$

To justify the last step, recall that in $\mathcal{SR}$, the monoidal unit is $\mathbf{I} = \varnothing$, and $\mathsf{Hom}_{\mathcal{SR}}(\mathbf{I}, R(A)_{q,x})$ consists of all the functions $R(A)_{q,x} \to (\Sigma + \varnothing)^*$ (the copylessness condition is vacuous when the source is $\varnothing$). We then conclude using the canonical bijection $S^T \times S^U \cong S^{T+U}$. □

Definition 5.3.4. Let $q \in Q(A)$, $r \in R(A)_q$ and $f : \Gamma^* \to \lfloor A \rfloor_\Sigma$. We define $L[f]_q \subseteq \Gamma^*$ and $\rho[f]_{q,r} : \Gamma^* \to \Sigma^*$ as follows. Let $w \in \Gamma^*$ and let $(q', (s_{r'})_{r' \in R(A)_{q'}})$ correspond to $f(w)$ by the bijection of the previous proposition. Then

$$w \in L[f]_q \iff q = q' \qquad\qquad \rho[f]_{q,r} = \begin{cases} s_r & \text{if } q = q' \\ \varepsilon & \text{otherwise} \end{cases}$$

We say that f is *comparison-free polyregular* when:

- $L[f]_q$ is a regular language for all $q \in Q(A)$;
- $\rho[f]_{q,r}$ is a comparison-free polyregular function for all q, r.

We will apply this semantic notion of cfp function to denotations of $\lambda\ell^{\oplus\&}$-terms. First, let $[\![-]\!] : \mathcal{L}(\Sigma) \to \mathcal{SR}_{\oplus\Sigma}(\Sigma)$ be the functor given by Lemma 5.2.6. For any $\lambda\ell^{\oplus\&}$-term t such that $\widetilde{\Sigma};\ \cdot\ \vdash t : A$ (where the typing context $\widetilde{\Sigma}$ is defined at the beginning of §5.2.1), write

$$\lambda().t = \lambda x.\ \texttt{let}\ ()\ \texttt{=}\ x\ \texttt{in}\ t \qquad \text{and} \qquad [\![t]\!]' = [\![\lambda().t]\!] \in \lfloor A \rfloor_\Sigma$$

(indeed, $\lambda().t \in \mathsf{Hom}_{\mathcal{L}}(\mathbf{I}, A)$). This is not sufficient to interpret terms of type $\mathtt{Str}_\Gamma[\tau] \to \mathtt{Str}_\Sigma$, however, so we introduce the following ad-hoc notion.

Definition 5.3.5. Suppose we are given a $\lambda\ell^{\oplus\&}$-term t with $\widetilde{\Sigma},\ s : \mathtt{Str}_\Gamma[\tau];\ \cdot \vdash t : \sigma$ where σ and τ are purely linear. We define the interpretation $\langle\!|t|\!\rangle : w \in \Gamma^* \mapsto [\![t[\underline{w}/s]]\!]' \in \lfloor [\![\sigma]\!] \rfloor_\Sigma$.

With these definitions in mind, we can state the suitable semantic generalization of our desired soundness result.

Theorem 5.3.6. *Let t be a $\lambda\ell^{\oplus\&}$-term such that $\widetilde{\Sigma},\ s : \mathtt{Str}_\Gamma[\tau];\ \cdot \vdash t : \sigma$ with σ and τ purely linear. Then the map between sets $\langle\!|t|\!\rangle : \Gamma^* \to \lfloor [\![\sigma]\!] \rfloor_\Sigma$ is comparison-free polyregular.*

Let us show that this indeed implies what we want.

Proof that $(\lambda\ell^{\oplus\&}, \to)$-definable $\implies$ comparison-free polyregular.
Let $t : \mathtt{Str}_\Gamma[\tau] \to \mathtt{Str}_\Sigma$ be a closed $\lambda\ell^{\oplus\&}$-term defining the function $f : \Gamma^* \to \Sigma^*$. According to the above theorem, the interpretation $g = \left\langle\!\middle|t\ s\ \ b_1\ \ \ldots\ \ b_{|\Sigma|}\ \varepsilon\middle|\!\right\rangle : \Gamma^* \to \lfloor \mathfrak{o} \rfloor_\Sigma$ (of t applied to the free variables in the context $\widetilde{\Sigma},\ s : \mathtt{Str}_\Gamma[\tau]$) is cfp. For any $w \in \Gamma^*$,

$$\left\langle\!\middle|t\ s\ \ b_1\ \ \ldots\ \ b_{|\Sigma|}\ \varepsilon\middle|\!\right\rangle(w) = [\![t\ \underline{w}\ \ b_1\ \ \ldots\ \ b_{|\Sigma|}\ \varepsilon]\!]' = [\![\underline{f(w)}\ \ b_1\ \ \ldots\ \ b_{|\Sigma|}\ \varepsilon]\!]'$$

By Definition 5.2.3 and using the η-conversion rules, $(\!|\underline{f(w)}\ \ b_1\ \ \ldots\ \ b_{|\Sigma|}\ \varepsilon|\!)_{\mathfrak{L}} = f(w)$. The fact that $[\![-]\!]$ is a morphism of streaming settings (Lemma 5.2.6), combined with the observation that the bijection $(\!|-|\!)_{\mathfrak{GR}_{\oplus\&}} : \lfloor \mathfrak{o} \rfloor_\Sigma \xrightarrow{\sim} \Sigma^*$ is an instance of Proposition 5.3.3, leads to $f(w) = \rho[g]_{q,r}(w)$ where q, r are such that $\{q\} = Q([\![\mathfrak{o}]\!])$ and $\{r\} = R([\![\mathfrak{o}]\!])_q$ (note that by Lemma 5.2.6, $[\![\mathfrak{o}]\!] = \perp\!\!\!\perp_{\mathfrak{GR}_{\oplus\&}} = \iota_{\oplus\&}(\{\bullet\})$). Since g is comparison-free polyregular, so is $f = \rho[g]_{q,r}$ by definition. □

We now intend to prove Theorem 5.3.6. The heart of the argument is contained in the following lemma involving a semantic counterpart to composition by substitution (§3.1).

Lemma 5.3.7. *Let $f_c : \Gamma^* \to \lfloor A \multimap A \rfloor_\Sigma$ for $c \in \Gamma$ and $g : \Gamma^* \to \lfloor A \rfloor_\Sigma$ be comparison-free polyregular. Then the map $h : \Gamma^* \to \lfloor A \rfloor_\Sigma$ defined below is also cfp:*

$$h \quad : \quad w = w[1] \ldots w[n] \quad \longmapsto \quad \Lambda^-(f_{w[n]}(w)) \circ \cdots \circ \Lambda^-(f_{w[1]}(w)) \circ (g(w))$$

where, for $\varphi \in \lfloor A \multimap A \rfloor_\Sigma$, we write $\Lambda^-(\varphi)$ for the corresponding morphism $A \to_\Sigma A$ (to be precise, Λ^- is the inverse of the bijection Λ' defined in Proposition 4.1.9).

A couple of further properties stating that comparison-free polyregular functions play well with the categorical structure of $\mathcal{SR}(\Sigma)_{\oplus\&}$ are also helpful. The first one allows to combine two functions $f : \Gamma^* \to \lfloor A \rfloor_\Sigma$ and $g : \Gamma^* \to \lfloor B \rfloor_\Sigma$ into a single function $\Gamma^* \to \lfloor A \otimes B \rfloor_\Sigma$. The second one pertains to the action of $\mathcal{SR}(\Sigma)_{\oplus\&}$ on $\lfloor A \rfloor_\Sigma$.

Lemma 5.3.8. *If $f : \Gamma^* \to \lfloor A \rfloor_\Sigma$ and $g : \Gamma^* \to \lfloor B \rfloor_\Sigma$ are comparison-free polyregular, so are the following:*

- $w \mapsto (f(w) \otimes g(w)) \circ \lambda_{\mathbf{I}}^{-1} \in \lfloor A \otimes B \rfloor_\Sigma$ *(where $\lambda_{\mathbf{I}}$ is the unitor $\mathbf{I} \otimes \mathbf{I} \to_\Sigma \mathbf{I}$)*
- $w \mapsto h \circ f(w) \in \lfloor C \rfloor_\Sigma$ *for any $h : A \to_\Sigma C$.*

Finally, our last tool for proving Theorem 5.3.6 is a syntactic lemma whose proof is deferred to Section 5.4.6.

Lemma 5.3.9. *Let τ be a purely linear type. The set of $\beta\eta$-equivalence classes of $\lambda\ell^{\oplus\&}$-terms t such that $\widetilde{\Sigma},\ s : \mathtt{Str}_\Gamma[\tau];\ \cdot \vdash t : \sigma$ for some purely linear σ can be exhaustively generated by the following inductive rules:*

- *base case: any term t such that $\widetilde{\Sigma};\ \cdot \vdash t : \sigma$;*
- *inductive case: $o\ \langle s\ d_1\ \dots\ d_{|\Sigma|}\ d_\varepsilon, ()\rangle$ for any $d_\varepsilon, d_1, \dots, d_{|\sigma|}, o$ in this set, with respective types $\sigma,\ \sigma \multimap \sigma,\ \dots,\ \sigma \multimap \sigma,\ (\sigma\ \&\ \mathbf{I}) \multimap \sigma'$ (thus, the new term has type σ').*

Proof of Theorem 5.3.6. Using the above lemma, we proceed by structural induction. In the base case, we have a term t such that the function $\langle\!|t|\!\rangle$ is constant, and therefore comparison-free polyregular. In the inductive case, we have

$$\widetilde{\Sigma},\ s : \mathtt{Str}_\Gamma[\tau];\ \cdot \vdash t = o\ \langle s\ d_1\ \dots\ d_{|\Sigma|}\ d_\varepsilon, ()\rangle : \sigma'$$

and the induction hypothesis is that $\langle\!|d_\varepsilon|\!\rangle, \langle\!|d_1|\!\rangle, \dots, \langle\!|d_{|\sigma|}|\!\rangle, \langle\!|o|\!\rangle$ are cfp. For $w \in \Gamma^*$,

$$\begin{aligned}
\left\langle\!\left|s\ d_1\ \dots\ d_{|\Sigma|}\ d_\varepsilon\right|\!\right\rangle(w) &= [\![\underline{w}\ d_1[\underline{w}/s]\ \dots\ d_{|\Sigma|}[\underline{w}/s]\ d_\varepsilon[\underline{w}/s]]\!]' \\
&= [\![\lambda().\ \underline{w}\ d_1[\underline{w}/s]\ \dots\ d_{|\Sigma|}[\underline{w}/s]\ d_\varepsilon[\underline{w}/s]]\!] \\
&= [\![d_{w[1]}[\underline{w}/s] \circ \dots \circ d_{w[|w|]}[\underline{w}/s] \circ (\lambda().\ d_\varepsilon[\underline{w}/s])]\!] \\
&= [\![d_{w[1]}[\underline{w}/s]]\!] \circ \dots \circ [\![d_{w[|w|]}[\underline{w}/s]]\!] \circ [\![\lambda().\ d_\varepsilon[\underline{w}/s]]\!] \\
&= \Lambda^-([\![d_{w[1]}[\underline{w}/s]]\!]') \circ \dots \circ \Lambda^-([\![d_{w[|w|]}[\underline{w}/s]]\!]') \circ [\![d_\varepsilon[\underline{w}/s]]\!]' \\
&= \Lambda^-(\left\langle\!\left|d_{w[1]}\right|\!\right\rangle(w)) \circ \dots \circ \Lambda^-(\left\langle\!\left|d_{w[|w|]}\right|\!\right\rangle(w)) \circ \langle\!|d_\varepsilon|\!\rangle(w)
\end{aligned}$$

The penultimate step is the preservation of the monoidal closed structure by the functor $[\![-]\!]$; indeed, the canonical bijection $\mathsf{Hom}_{\mathcal{L}}(\sigma,\sigma) \xrightarrow{\sim} \mathsf{Hom}_{\mathcal{L}}(\mathbf{I}, \sigma \multimap \sigma)$ sends t to $\lambda().t$, just as in the definition of $[\![-]\!]'$ from $[\![-]\!]$. By Lemma 5.3.7, $\left\langle\!\left|s\ d_1\ \dots\ d_{|\Sigma|}\ d_\varepsilon\right|\!\right\rangle(w)$ is therefore comparison-free polyregular. Now, it can be checked that for $w \in \Gamma^*$,

$$\left\langle\!\left|o\ \langle s\ d_1\ \dots\ d_{|\Sigma|}\ d_\varepsilon, ()\rangle\right|\!\right\rangle(w) \ =\ \mathrm{ev} \circ \left(\langle\!|o|\!\rangle(w) \otimes \left(p \circ \left\langle\!\left|s\ d_1\ \dots\ d_{|\Sigma|}\ d_\varepsilon\right|\!\right\rangle(w)\right)\right) \circ \lambda_{\mathbf{I}}^{-1}$$

where ev comes from the universal property of $([\![\tau]\!]\ \&\ \mathbf{I}) \multimap [\![\sigma]\!]$ and $p : [\![\tau]\!] \to_\Sigma [\![\tau]\!]\ \&\ \mathbf{I}$ is the canonical morphism pairing its input with $()$. We may thus conclude the argument by applying Lemma 5.3.8 three times. □

5.3.3. **Proof of Lemma 5.3.7.** Let us fix $q \in Q(A)$ and $r \in R(A)_q$. First, we show that the language $L[h]_q$ is regular.

Proof. We have $L[h]_q = \{w \in \Gamma^* \mid Q(h(w))(\bullet) = q\}$ by definition. Unraveling the definition of h and applying the functoriality of Q, we also have

$$Q(h(w)) \ =\ Q(\Lambda^-(f_{w[n]}(w))) \circ \dots \circ Q(\Lambda^-(f_{w[1]}(w))) \circ Q(g(w))$$

from which we can deduce that

$$L[h]_q \ =\ \bigcup_{q' \in Q(A)}\ \bigcup_{\delta}\ \underbrace{\{w_1 \dots w_n \mid \delta(-,w_n) \circ \dots \circ \delta(-,w_1)(q') = q\}}_{\text{recognized by a DFA with transition function } \delta} \cap L[g]_{q'} \cap \bigcap_{c \in \Gamma} L_{c,\delta(-,c)}$$

where δ ranges over functions $Q(A) \times \Gamma \to Q(A)$ and

$$L_{c,\eta} = \{w \mid Q(\Lambda^-(f_c(w))) = \eta\}$$

Since g was assumed to be cfp, $L[g]_{q'}$ is regular for each $q' \in Q(A)$. For every $c \in \Gamma$ and $\eta : Q(A) \to Q(A)$, we also have (using $\mathrm{ev}_{A,A} : (A \multimap A) \otimes A \to_\Sigma A$ from Definition 4.1.3)

$$L_{c,\eta} \ =\ \bigcup \left\{L[f_c]_{q''} \;\middle|\; q'' \in Q(A \multimap A) \text{ such that } \eta = Q(\mathrm{ev}_{A,A})(q'', -)\right\}$$

and each $L[f_c]_{q''}$ is regular by assumption, so $L_{c,\eta}$ is also regular. Since regular languages are closed under finite boolean operations, we are done. □

This being done, we will now show the following claim, which entails the lemma statement by definition: the function $\rho[h]_{q,r} : \Gamma^* \to \Sigma^*$ can be written as a composition by substitution $\mathrm{CbS}(h'_{q,r}, (\gamma_i)_{i\in I})$ where

- $I = \sum_{q'\in Q(A)} R(A)_{q'} \sqcup \Gamma \times \left(\sum_{q''\in Q(A\multimap A)} R(A \multimap A)_{q''} \right)$;
- $h'_{q,r} : \Gamma^* \to I^*$ is a regular function;
- $\gamma_{q',r'} = \rho[g]_{q',r'}$ for $q' \in Q(A)$ and $r' \in R(A)_{q'}$;
- $\gamma_{c,q'',r''} = \rho[f_c]_{q'',r''}$ for $c \in \Gamma$, $q'' \in Q(A \multimap A)$ and $r'' \in R(A \multimap A)_{q''}$.

Proof. We will show that for any $q_0 \in Q(A)$ and $(q_c)_{c\in\Gamma} \in Q(A \multimap A)^\Gamma$, we can define using such a composition by substitution a function that coincides with $\rho[h]_{q,r}$ on the regular language $L \subseteq \Gamma^*$ defined as

$$L = L[g]_{q_0} \cap \bigcap_{c\in\Gamma} L[f_c]_{q_c}$$

From this, one can derive the desired conclusion concerning $\rho[h]_{q,r}$ using the closure of regular functions under regular conditionals (Proposition 2.3.11): combine the functions obtained for every combinations of q_0 and $(q_c)_c$, leveraging the fact that

$$\mathrm{CbS}\left(\left(w \mapsto \begin{cases}\alpha(w) & \text{if } w \in L'\\ \beta(w) & \text{otherwise}\end{cases}\right), (\gamma_i)_{i\in I}\right) = \left(w \mapsto \begin{cases}\mathrm{CbS}(\alpha, (\gamma_i)_{i\in I})(w) & \text{if } w \in L'\\ \mathrm{CbS}(\beta, (\gamma_i)_{i\in I})(w) & \text{otherwise}\end{cases}\right)$$

We now fix q_0 and $(q_c)_{c\in\Gamma}$. For $w \in \Gamma^*$, let $F_w : \mathcal{SR}(I)_{\oplus\&} \to \mathcal{SR}(\Sigma)_{\oplus\&}$ be the letter substitution functor induced by $i \mapsto \gamma_i(w)$ (this functor is the identity on objects). Let $\zeta \in \lfloor A \rfloor_I$ correspond to

$$(q_0, ((q_0, r'))_{r'\in R(A)_{q'}}) \in \sum_{q'\in Q(A)} (I^*)^{R(A)_{q'}}$$

We claim that if $w \in L[g]_{q_0}$, then $F_w(\zeta) = g(w)$. This is because $F_w(\zeta)$ corresponds to

$$(q_0, (\gamma_{q_0,r'}(w))_{r'\in R(A)_{q'}}) = (q_0, (\rho[g]_{q_0,r'}(w))_{r'\in R(A)_{q'}}) \in \sum_{q'\in Q(A)} (\Sigma^*)^{R(A)_{q'}}$$

Similarly, for each $c \in \Gamma$, there exists $\xi_c \in \lfloor A \multimap A \rfloor_I$ *that depends on* q_c *but not* w such that for any $w \in L[f]_{q_c}$, we have $F_w(\xi_c) = f_c(w)$. The functor F_w also preserves the monoidal closed structure on the nose, so we have $F_w(\Lambda^-(\xi_c)) = \Lambda^-(f_c(w))$.

By definition of L, all those conditions on w are implied by $w \in L$. By functoriality,

$$\forall w = w[1]\dots w[n] \in L, \quad F_w(\Lambda^-(\xi_{w[n]}) \circ \dots \circ \Lambda^-(\xi_{w[1]}) \circ \zeta) = h(w)$$

The action of F_w on the "register contents" of an element of $\lfloor A \rfloor_I$ is to replace every $i \in I$ by $\gamma_i(w)$. Comparing this with the definition of composition by substitution, we get:

$$\forall w \in L,\ \mathrm{CbS}\left(\rho[w[1]\dots w[n] \mapsto \Lambda^-(\xi_{w[n]}) \circ \dots \circ \Lambda^-(\xi_{w[1]}) \circ \zeta]_{q,r}, (\gamma_i)_{i\in I}\right)(w) = \rho[h]_{q,r}(w)$$

Call $h' : \Gamma^* \to I^*$ the first argument of the CbS in the left-hand side. It has the shape of a function computed by a *single-state* $\mathfrak{SR}_{\oplus\&}$*-SST*. As shown in Section 4.5.2, such devices only compute *regular functions*. Thus, h' is regular, which concludes the proof. □

5.3.4. **Proof of Lemma 5.3.8.**

PROOF OF THE FIRST ITEM. Since $\otimes$ is defined in $\mathcal{SR}_{\oplus\&}$ by distributivity over formal coproducts, we have $Q(A \otimes B) = Q(A) \times Q(B)$. To check that $L[f \otimes g]_{(q,q')}$ is regular, observe that it is equal to $L[f]_q \cap L[g]_{q'}$. For $r \in R(A)_q$, we have $\rho[f \otimes g]_{(q,q'),r} = \rho[f]_{q,r}$ and for $r' \in R(B)_q$, we have $\rho[f \otimes g]_{(q,q'),r'} = \rho[g]_{q',r'}$; hence, for every $r \in R(A \otimes B)$, we know that $\rho[f \otimes g]_{(q,q'),r}$ is comparison-free polyregular, so we may conclude. □

PROOF SKETCH FOR THE SECOND ITEM. Fix $q \in Q(B)$. The following equality shows that $L[h^\circ \circ f]_q$ is regular:

$$L[h^\circ \circ f]_q = \bigcup \{ L[f]_{q'} \mid q' \in Q(A) \text{ such that } Q(h)(q') = q \}$$

Next, given $r \in R(B)_q$, we wish to show that $\rho[h^\circ \circ f]_{q,r}$ is comparison-free polyregular. To do so, we rely on Proposition 3.1.8: cfp functions are closed under both concatenation and regular conditionals. The idea is as follows: for $q' \in Q(h)^{-1}(q)$, one can build a function that coincides with $\rho[h^\circ \circ f]_{q,r}$ on $L[f]_{q'}$ by concatenating a finite sequence of functions that are either constant or equal to $\rho[f]_{q',r'}$ for some $r' \in R(A)_{q'}$. (Without entering into cumbersome details, let us give a hint as to the provenance of the concatenation pattern: it comes from applying to r some register transition, i.e. $\mathcal{SR}$-morphism, that is bundled as part of the $\mathcal{SR}_{\oplus\&}$-morphism h.) A regular conditional can then be used to combine the finitely many possibilities for q' and thus conclude the proof. □

5.4. SYNTACTIC BUREAUCRACY

This section, which concludes the current chapter on string functions in the $\lambda\ell^{\oplus\&}$-calculus, proves the boring syntactic lemmas that have been delayed until now. Recall that we have already discussed the high-level considerations regarding these syntactic aspects at the beginning of the chapter.

In Section 5.4.1 we show a first normalisation result. We then apply it in the three subsequent subsections to prove our claims on the structure of Church encodings and of $\lambda\ell^{\oplus\&}$-terms of type $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{Str}_\Gamma$ (where τ is purely linear). For $\mathtt{Str}_\Sigma[\tau] \to \mathtt{Str}_\Gamma$, it is more convenient to use a refined notion of normal forms, which does not only eliminate all β-redexes, but also orders the use of eliminators (term applications $t\ u$, case, …) in a principled way according to a notion of *polarity*. In proof theory, this is called *focusing*. We establish the corresponding focused normalization theorem in Section 5.4.5 and then use it for the proof of our last syntactic lemma in Section 5.4.6.

5.4.1. **Normalization of the $\lambda\ell^{\oplus\&}$-calculus.** We are going to prove that the $\lambda\ell^{\oplus\&}$-calculus is strongly normalizing. What it means is that any $\lambda\ell^{\oplus\&}$-term t admitting a typing derivation $\Psi;\ \Delta \vdash t : A$ can be shown to be $\beta\eta$-equivalent to a *normal* term u. The notion of normal (NF) and *neutral* (NE) are defined via the typing system presented in Figure 5.4.1. The intuition is that a normal term cannot be β-reduced further, and that neutral terms substituted in normal terms produce terms that stay normal. Our proof proceeds along the lines of [RR97, Appendix A.1], using reducibility candidates. All this is completely unsurprising; our only reason for including this routine material is that we are not aware of a text treating exactly $\lambda\ell^{\oplus\&}$ (e.g., incorporating additives, a native $\otimes$ and units).

To describe said reducibility candidates, we first need to give an oriented version of $\beta\eta$-equality. The β-reduction relation $\to_\beta$ is obtained by closing the relations given in Figure 5.4.2 by congruence. We write $\to_\beta^*$ for the reflexive transitive closure relation. Much like with $=_\beta$, we assume that terms related by $\to_\beta$ have the same type in the same context.

As it is not the case that every typable term β-reduces to a normal form, we need to describe another set of reduction rules which involve $=_\eta$. Those *extrusion rules* are

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau}{\Psi;\ \Delta \vdash_{\mathrm{NF}} t : \tau}$$

$$\frac{}{\Psi;\ x : \tau \vdash_{\mathrm{NE}} x : \tau} \qquad \frac{}{\Psi, x : \tau;\ \cdot \vdash_{\mathrm{NE}} x : \tau}$$

$$\frac{\Psi;\ \Delta,\ x : \tau \vdash_{\mathrm{NF}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}} \lambda x.t : \tau \multimap \sigma} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \multimap \sigma \qquad \Psi;\ \Delta' \vdash_{\mathrm{NF}} u : \tau}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} t\ u : \sigma}$$

$$\frac{\Psi,\ x : \tau;\ \Delta \vdash_{\mathrm{NF}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}} \lambda^{!} x.t : \tau \to \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NF}} t : \tau \qquad \Psi;\ \Delta' \vdash_{\mathrm{NF}} u : \sigma}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NF}} t \otimes u : \tau \otimes \sigma} \qquad \frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}} t : \tau \otimes \sigma \qquad \Psi;\ \Delta,\ x : \tau,\ y : \sigma \vdash_{X} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{X} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \kappa}$$

$$\frac{}{\Psi;\ \cdot \vdash_{\mathrm{NF}} () : \mathbf{I}} \qquad \frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}} t : \mathbf{I} \qquad \Psi;\ \Delta \vdash_{X} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{X} \mathtt{let}\ () = t\ \mathtt{in}\ u : \kappa}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NF}} t : \tau \qquad \Psi;\ \Delta \vdash_{\mathrm{NF}} u : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}} \langle t, u \rangle : \tau \,\&\, \sigma} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \,\&\, \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NE}} \pi_1(t) : \tau} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \,\&\, \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NE}} \pi_2(t) : \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NF}} t : \tau}{\Psi;\ \Delta \vdash_{\mathrm{NF}} \mathsf{in}_1(t) : \tau \oplus \sigma} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NF}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}} \mathsf{in}_2(t) : \tau \oplus \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \sigma \oplus \tau \qquad \Psi;\ \Delta',\ x : \sigma \vdash_{X} u : \kappa \qquad \Psi;\ \Delta',\ y : \tau \vdash_{X} v : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{X} \mathsf{case}(t, x.u, y.v) : \kappa}$$

$$\frac{}{\Psi;\ \Delta \vdash_{\mathrm{NF}} \langle\rangle : \top} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : 0}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} \mathsf{abort}(t) : \tau}$$

FIGURE 5.4.1. Normal forms for $\lambda\ell^{\oplus\&}$-terms ($\vdash_{\mathrm{NF}}$ for normal forms and $\vdash_{\mathrm{NE}}$ for neutral forms and $X \in \{\mathrm{NE}, \mathrm{NF}\}$).

β-redexes	$(\lambda x.t)\ u \ \to_\beta \ t[u/x]$	$(\lambda^{!} x.t)\ u \ \to_\beta \ t[u/x]$
	$\pi_1(\langle t, u \rangle) \ \to_\beta \ t$	$\pi_2(\langle t, u \rangle) \ \to_\beta \ u$
	$\mathsf{case}(\mathsf{in}_1(t), x.u, x.v) \ \to_\beta \ u[t/x]$	$\mathsf{case}(\mathsf{in}_2(t), x.u, x.v) \ \to_\beta \ v[t/x]$
	$\mathtt{let}\ x \otimes y = t \otimes u\ \mathtt{in}\ v \ \to_\beta \ v[t/x][u/y]$	$\mathtt{let}\ () = ()\ \mathtt{in}\ t \ \to_\beta \ t$

FIGURE 5.4.2. β-redexes.

listed in Figure 5.4.3; we also write $\to_\varepsilon$ for the congruence closure of the relation described there. While the number of cases is daunting, it should be remarked that these rules are obtained mechanically by considering the nesting of an eliminator for a positive type (i.e., the $\mathtt{let} \cdot = \cdot \mathtt{in} \cdot$ constructions ($\otimes, \mathbf{I}$), case ($\oplus$) and abort (0)) within another eliminator (the aforementioned constructions plus function application ($\to, \multimap$) and projections π_1, π_2 ($\&$)). For a more careful discussion of (a subset) of these rules, we point the reader to [Sch16, Section 3.3]. We write $\to_\varepsilon^*$ for the reflexive transitive closure of $\to_\varepsilon$, $\to_{\beta\varepsilon}$ for the union of $\to_\varepsilon$ and $\to_\beta$ and $\to_\varepsilon$, and $\to_{\beta\varepsilon}^*$ for its reflexive transitive closure. With these notations, we can state the finer version of the normalization theorem for $\lambda\ell^{\oplus\&}$.

Theorem 5.4.1. *For every term term t such that $\Psi;\ \Delta \vdash t : \tau$, there exists t' such that $t \to_{\beta\varepsilon}^* t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau$.*

Before embarking on the definitions of the reducibility candidates and the proof of Theorem 5.4.1 itself, we first make a couple of observations relating $\to_{\beta\varepsilon}^*$ and $=_{\beta\eta}$, which are both proved by straightforward induction on the relations $\to_\beta$ and $\to_{\beta\varepsilon}$.

Lemma 5.4.2. *Suppose that we have terms t and t' with matching types and that $t \to_{\beta\varepsilon} t'$. Then, we have $t =_{\beta\eta} t'$. Furthermore if t is normal, so is t'. Similarly, if t is neutral, so is t'.*

Lemma 5.4.3. *If t is normal, then there is no t' such that $t \to_\beta t'$.*

Another crucial ingredient is the confluence of the reduction relation $\to_{\beta\varepsilon}$ (or Church-Rosser property). Alas, this does not hold for syntactic equality (up to α-equivalence. However, it holds up to *commuting conversions*, an equivalence relation $\approx_c$ inductively defined by the clauses in Figure 5.4.4 and closure under congruence. We merely state the confluence property that we will use.

Theorem 5.4.4. *If we have $t \to_{\beta\varepsilon}^* u$ and $t \to_{\beta\varepsilon}^* v$, then there exists u' and v' such that $u \to_{\beta\varepsilon}^* u'$, $v \to_{\beta\varepsilon}^* v'$ and $u' \approx_c v'$.*

A first observation is these are compatible with $=_{\beta\eta}$ and that neutral and normal term are preserved by commutative conversions.

Lemma 5.4.5. *If $t \approx_c t'$, then $t =_{\beta\eta} t'$.*

Lemma 5.4.6. *If t is neutral (resp. normal) and $t \approx_c t'$, then t' is also neutral (resp. normal).*

We can now turn to the definition of the reducibility candidates, where we write $t \to_{\beta\varepsilon}^* \approx_c t'$ when there is some t'' such that $t \to_{\beta\varepsilon}^* t''$ and $t'' \approx_c t'$ hold.

Definition 5.4.7. Define a judgment $\Psi;\ \Delta \models t : \tau$ by induction over the type τ as follows:

- for $\tau = \mathbb{0}, \mathbf{I}, 0$ or $\top$, we have $\Psi;\ \Delta \models t : \tau$ if and only if there is t' such that $t \to_{\beta\varepsilon}^* t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NE}} t' : \tau$
- $\Psi;\ \Delta \models t : \tau \multimap \sigma$ holds if and only if
  - there is t' such that $t \to_{\beta\varepsilon}^* t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau \multimap \sigma$
  - for every u, Δ' such that $\Psi;\ \Delta' \models u : \tau$, we have $\Psi;\ \Delta,\ \Delta' \models t\ u : \sigma$
- $\Psi;\ \Delta \models t : \tau \to \sigma$ holds if and only if
  - there is t' such that $t \to_{\beta\varepsilon}^* t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau \to \sigma$
  - for every u such that $\Psi;\ \cdot \models u : \tau$, we have $\Psi;\ \Delta \models t\ u : \sigma$
- $\Psi;\ \Delta \models t : \tau \otimes \sigma$ holds if and only if
  - there is t' such that $t \to_{\beta\varepsilon}^* t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau \otimes \sigma$

Nested $\otimes/\mathbf{I}$ eliminator

$$\begin{aligned}
(\texttt{let}\ p = t\ \texttt{in}\ u)\ v &\to_\varepsilon \texttt{let}\ p = t\ \texttt{in}\ (u\ v)\\
\pi_i(\texttt{let}\ p = t\ \texttt{in}\ u) &\to_\varepsilon \texttt{let}\ p = t\ \texttt{in}\ \pi_i(u)\\
\texttt{let}\ q = \texttt{let}\ p = t\ \texttt{in}\ u\ \texttt{in}\ v &\to_\varepsilon \texttt{let}\ p = t\ \texttt{in}\ \texttt{let}\ q = u\ \texttt{in}\ v\\
\mathsf{abort}(\texttt{let}\ p = t\ \texttt{in}\ u) &\to_\varepsilon \texttt{let}\ p = t\ \texttt{in}\ \mathsf{abort}(u)\\
\mathsf{case}(\texttt{let}\ p = t\ \texttt{in}\ u, x.v, y.w) &\to_\varepsilon \texttt{let}\ p = t\ \texttt{in}\ \mathsf{case}(u, x.v, y.w)
\end{aligned}$$

Nested 0 eliminator

$$\begin{aligned}
\mathsf{abort}(t)\ u &\to_\varepsilon \mathsf{abort}(t)\\
\pi_i(\mathsf{abort}(t)) &\to_\varepsilon \mathsf{abort}(t)\\
\texttt{let}\ p = \mathsf{abort}(t)\ \texttt{in}\ u &\to_\varepsilon \mathsf{abort}(t)\\
\mathsf{abort}(\mathsf{abort}(t)) &\to_\varepsilon \mathsf{abort}(t)\\
\mathsf{case}(\mathsf{abort}(t), x.u, y.v) &\to_\varepsilon \mathsf{abort}(t)
\end{aligned}$$

Nested $\oplus$ eliminator

$$\begin{aligned}
\mathsf{case}(t, x.u, y.v)\ w &\to_\varepsilon \mathsf{case}(t, x.u\ w, y.v\ w)\\
\pi_i(\mathsf{case}(t, x.u, y.v)) &\to_\varepsilon \mathsf{case}(t, x.\pi_i(u), y.\pi_i(v))\\
\texttt{let}\ p = \mathsf{case}(t, x.u, y.v)\ \texttt{in}\ w &\to_\varepsilon \mathsf{case}(t, x.\texttt{let}\ p = u\ \texttt{in}\ w, y.\texttt{let}\ p = v\ \texttt{in}\ w)\\
\mathsf{abort}(\mathsf{case}(t, x.u, y.v)) &\to_\varepsilon \mathsf{case}(t, x.\mathsf{abort}(u), y.\mathsf{abort}(v))\\
\mathsf{case}(\mathsf{case}(t, x.u, y.v), x'.u', y'.v') &\to_\varepsilon \mathsf{case}(t, x.\mathsf{case}(u, x'.u', y'.v'), y.\mathsf{case}(v, x'.u', y'.v'))
\end{aligned}$$

FIGURE 5.4.3. The extrusion relation $\to_\varepsilon$ ($i = 1, 2$ and p, q are patterns $()$ or $z \otimes z'$).

$$\begin{aligned}
\texttt{let}\ q = u\ \texttt{in}\ \texttt{let}\ p = t\ \texttt{in}\ v &\approx_c \texttt{let}\ p = t\ \texttt{in}\ \texttt{let}\ q = u\ \texttt{in}\ v\\
\texttt{let}\ p = t\ \texttt{in}\ \mathsf{abort}(u) &\approx_c \mathsf{abort}(\texttt{let}\ p = t\ \texttt{in}\ u)\\
\texttt{let}\ p = t\ \texttt{in}\ \mathsf{case}(u, x.v, y.w) &\approx_c \mathsf{case}(u, x.\texttt{let}\ p = t\ \texttt{in}\ v, y.\texttt{let}\ p = t\ \texttt{in}\ w)\\
\mathsf{case}(u, x.\mathsf{abort}(t), y.\mathsf{abort}(v)) &\approx_c \mathsf{abort}(\mathsf{case}(u, x.t, y.v))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{case}(t, x.\mathsf{case}(u, x'.v, y'.w), y.\mathsf{case}(u, x'.v', y'.w'))\\
&\approx_c\\
&\mathsf{case}(u, x'.\mathsf{case}(t, x.v, y.v'), y'.\mathsf{case}(t, x.w, y.w'))
\end{aligned}$$

FIGURE 5.4.4. Commutative conversions $\approx_c$ (p, q are patterns $()$ or $z \otimes z'$ and both sides are assumed to be well-scoped).

  - if there are t_1, t_2 such that and $t \to^*_{\beta\varepsilon} t_1 \otimes t_2$, then there are Δ_1 and Δ_2 such that $\Delta = \Delta_1, \Delta_2$ and

$$\Psi;\ \Delta_1 \models t_1 : \tau \qquad \text{and} \qquad \Psi;\ \Delta_2 \models t_2 : \sigma$$

- $\Psi;\ \Delta \models t : \tau \,\&\, \sigma$ holds if and only if
  - there is t' such that $t \to^*_{\beta\varepsilon} t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau \,\&\, \sigma$
  - $\Psi;\ \Delta \models \pi_1(t) : \tau$
  - $\Psi;\ \Delta \models \pi_2(t) : \sigma$
- $\Psi;\ \Delta \models t : \tau \oplus \sigma$ holds if and only if
  - there is t' such that $t \to^*_{\beta\varepsilon} t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau \oplus \sigma$
  - if there is u such that $t \to^*_{\beta\varepsilon} \approx_c \mathsf{in}_1(u)$, then $\Psi;\ \Delta \models u : \tau$
  - if there is v such that $t \to^*_{\beta\varepsilon} \approx_c \mathsf{in}_2(v)$, then $\Psi;\ \Delta \models v : \sigma$

The set of terms t such that $\Psi;\ \Delta \models t : \tau$ constitutes our set of reducibility candidates at type τ in the context $\Psi;\ \Delta$. They are defined in such a way that if $\Psi;\ \Delta \models t : \tau$, then there is t' such that $t \to^*_{\beta\varepsilon} t'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau$. We shall be able to conclude this section if we show an *adequacy lemma* stating that every typable term lies in a reducibility candidate. Before doing that, we first need a couple of stability properties: closure under anti-reduction, and the fact that every neutral term lies in a reducibility candidate.

Lemma 5.4.8. *If t is a neutral term, then t cannot be $\beta\eta$-equivalent to one of the following*

$$\lambda x.u \qquad \lambda^! x.u \qquad (u, v) \qquad u \otimes v \qquad \langle\rangle \qquad () \qquad \mathsf{in}_1(u) \qquad \mathsf{in}_2(u)$$

Proof. Trivial case-analysis. □

Theorem 5.4.9. *Suppose that $\Psi;\ \Delta \models t' : \tau$. Then the following hold:*

- *There exists t'' such that $t' \to^*_{\beta\varepsilon} t''$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} t'' : \tau$.*
- *If we have $t \to^*_{\beta\varepsilon} t'$, then we also have $\Psi;\ \Delta \models t : \tau$.*
- *If $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau$, then $\Psi;\ \Delta \models t : \tau$.*
- *If $t' \approx_c t''$, then $\Psi;\ \Delta \models t'' : \tau$.*
- *If $t' \to^*_{\beta\varepsilon} t''$, then we also have $\Psi;\ \Delta \models t'' : \tau$.*

Proof. The first point can be proven via an easy case analysis on τ that we skip. The second point we may prove by induction over τ; let us sketch a few representative cases:

- If τ is $\mathbb{0}$, 0, $\mathbf{I}$ or $\top$, this is immediate.
- Suppose that $\tau = \sigma \multimap \kappa$ and that $t \to^*_{\beta\varepsilon} t'$. By definition of $\models$ at $\tau \multimap \kappa$, there is some normal t'' such that $t' \to^*_{\beta\varepsilon} t''$, so we also have $t \to^*_{\beta\varepsilon} t''$ by transitivity. Now suppose that we are given some u and Δ' such that $\Psi;\ \Delta' \models u : \tau$. By definition, we have $\Psi;\ \Delta,\ \Delta' \models t'\ u : \kappa$. Using the induction hypothesis at κ and the fact that $t\ u \to^*_{\beta\varepsilon} t'\ u$, we thus have that $\Psi;\ \Delta,\ \Delta' \models t\ u : \kappa$. We can thus conclude that $\Psi;\ \Delta \models t : \kappa$.
- Suppose that $\tau = \sigma \oplus \kappa$ and that $t \to^*_{\beta\varepsilon} t'$. By definition of $\models$, there is some normal t'' such that $t' \to^*_{\beta\varepsilon} t''$, so we also have $t \to^*_{\beta\varepsilon} t''$ by transitivity. Now if we have u (resp. v) such that $t \to^*_{\beta\varepsilon} \approx_c \mathsf{in}_1(u)$ (resp. $\mathsf{in}_2(v)$), then, thanks to confluence (Theorem 5.4.4) we also have $t' \to^*_{\beta\varepsilon} \approx_c \mathsf{in}_1(u)$ (resp. $\mathsf{in}_2(v)$), so we have $\Psi;\ \Delta \models u : \sigma$ (resp. $\Psi;\ \Delta \models u : \kappa$) by definition. Therefore, we may conclude that $\Psi;\ \Delta \models t : \sigma \oplus \kappa$.

The third point is also proved via a straightforward induction over τ by leveraging Lemma 5.4.8. The last two points follow from induction over τ combined with Theorem 5.4.4 and Lemma 5.4.6. □

Corollary 5.4.10. *If x is a variable of type τ in either Ψ or Δ, we have $\Psi;\ \Delta \models x : \tau$.*

Proof. Immediate as variables are neutral. □

Lemma 5.4.11. *Suppose that we have* $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \otimes \sigma$ *and* $\Psi;\ \Delta', x : \tau, y : \sigma \models u : \kappa$. *Then* $\Psi;\ \Delta,\ \Delta' \models \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \kappa$.

Similarly, if $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \oplus \sigma$, $\Psi;\ \Delta', x : \tau \models u : \kappa$ *and* $\Psi;\ \Delta', y : \sigma \models u' : \kappa$, *we have* $\Psi;\ \Delta, \Delta' \models \mathsf{case}(t, x.u, y.u') : \kappa$.

Finally, if $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \mathbf{I}$ *and* $\Psi;\ \Delta' \models u : \kappa$, *then* $\Psi;\ \Delta,\ \Delta' \models \mathtt{let}\ () = t\ \mathtt{in}\ u : \kappa$.

Proof. By induction over κ. □

Theorem 5.4.12 (Adequacy)**.** *Suppose that we have a non-linear context* $\Psi = x_1 : \sigma_1, \ldots, x_k : \sigma_k$, *a linear context* $\Delta = a_1 : \tau_1, \ldots, a_n : \tau_n$ *and a term* v *such that* $\Psi;\ \Delta \vdash v : \kappa$ *for some type* κ. *Further, assume that* $\Psi', \Delta'_1, \ldots, \Delta'_n$ *and terms* $t_1, \ldots t_k, u_1, \ldots, u_n$ *such that* $\Psi';\ \cdot \models t_i : \sigma_i$ *for* $1 \leq i \leq k$ *and* $\Psi';\ \Delta'_j \models u_j : \tau_j$ *for* $1 \leq j \leq n$. *Then we have*

$$\Psi';\ \Delta'_1, \ldots,\ \Delta'_n \models v[t_1/x_1, \ldots, t_k/x_k, u_1/a_1, \ldots, u_n/a_n] : \kappa$$

Proof. The proof goes by induction over the typing derivation $\Psi;\ \Delta \vdash v : \kappa$; we sketch a few representative subcases below. To keep notations short, we write γ (respectively δ) instead of the sequence of assignments $t_1/x_1, \ldots, t_k/x_k$ (respectively $u_1/a_1, \ldots, u_n/a_n$) and $\Delta' = \Delta'_1, \ldots,\ \Delta'_n$.

- If the last rule used in an axiom, the conclusion is immediate.
- If the last rule used is a linear function application

$$\frac{\Psi;\ \Delta_1 \vdash v : \kappa \multimap \kappa' \qquad \Psi;\ \Delta_2 \vdash v' : \kappa}{\Psi;\ \Delta_1,\ \Delta_2 \vdash v\ v' : \kappa'}$$

  with δ_1, δ_2 and $\Delta''_1,\ \Delta''_2$ the obvious decomposition of δ and Δ', the induction hypothesis yields

$$\Psi';\ \Delta''_1 \models v[\gamma, \delta_1] : \kappa \multimap \kappa' \qquad \text{and} \qquad \Psi';\ \Delta''_2 \models v'[\gamma, \delta_2] : \kappa$$

  By definition of $\models$ for type $\kappa \multimap \kappa'$, we thus have $\Psi';\ \Delta' \models v[\gamma, \delta_1]\ v'[\gamma, \delta_2] : \kappa'$, so we may conclude.
- The case of non-linear function application is entirely analogous.
- If the last rule is the typing of a linear λ-abstraction

$$\frac{\Psi;\ \Delta,\ c : \kappa \vdash v : \kappa'}{\Psi;\ \Delta \vdash \lambda c.v : \kappa \multimap \kappa'}$$

  by the inductive hypothesis, we have $\Psi';\ \Delta', \Delta'' \vdash v[\gamma, \delta, v'/c] : \kappa'$ for any v' and Δ'' such that $\Psi';\ \Delta'' \models v' : \kappa$. We can prove the conjunct defining $\Psi';\ \Delta' \models (\lambda c.v)[\gamma, \delta] : \kappa \multimap \kappa'$ as follows:
  - First, by taking $\Delta'' = c : \kappa$ (Corollary 5.4.10), we obtain that there exists some v'' such that $v[\gamma, \delta] \to^*_{\beta\varepsilon} v''$ and $\Psi;\ \Delta,\ c : \kappa \vdash_{\mathrm{NF}} v'' : \kappa'$. Therefore we have $\lambda c.v[\gamma, \delta] \to^*_{\beta\varepsilon} \lambda c.v'$ and $\Psi;\ \Delta \vdash_{\mathrm{NF}} \lambda c.v' : \kappa \multimap \kappa'$.
  - Then, assume we have some v' and $\Psi;\ \Delta'' \models v' : \kappa$, so that we have $\Psi';\ \Delta', \Delta'' \vdash v[\gamma, \delta, v'/c] : \kappa'$ by the inductive hypothesis. Because

$$(\lambda c.v)[\gamma, \delta]\ v' = (\lambda c.v[\gamma, \delta])\ v' \to_\beta v[\gamma, \delta, v'/c]$$

    we may apply Theorem 5.4.9 to conclude that $\Psi';\ \Delta',\ \Delta'' \models (\lambda c.v)[\gamma, \delta]\ v' : \kappa'$
- The case of the non-linear λ-abstraction for $\to$ is similar.
- If the last rule applied is an introduction of $\otimes$,

$$\frac{\Psi;\ \Delta_1 \vdash t : \tau \qquad \Psi;\ \Delta_2 \vdash u : \sigma}{\Psi;\ \Delta_1,\ \Delta_2 \vdash t \otimes u : \tau \otimes \sigma}$$

  call δ_1, δ_2 the splitting of δ according to the decomposition $\Delta = \Delta_1,\ \Delta_2$. The induction hypothesis yields

$$\Psi';\ \Delta'_1 \models t[\gamma, \delta_1] : \tau \qquad \text{and} \qquad \Psi';\ \Delta'_2 \models u[\gamma, \delta_2] : \sigma$$

By definition it means that we have normal terms t' and u' such that $t[\gamma, \delta_1] \to^*_{\beta\varepsilon} t'$ and $u[\gamma, \delta_2] \to^*_{\beta\varepsilon} u'$, so $(t \otimes u)[\gamma, \delta] \to^*_{\beta\varepsilon} t \otimes u'$. Now suppose that we have $(t \otimes u)[\gamma, \delta] \to^*_{\beta\varepsilon} \approx_c t'' \otimes u''$. It is not difficult to check (by induction over the length of the reductions $\to^*_{\beta\varepsilon}$ and derivation of $\approx_c$) that we have $t[\gamma, \delta_1] \to^*_{\beta\varepsilon} \approx_c t''$ and $u[\gamma, \delta_2] \to^*_{\beta\varepsilon} \approx_c u''$. So by Theorem 5.4.9, we have that $\Psi;\ \Delta_1 \models t'' : \tau$ and $\Psi;\ \Delta_2 \models u'' : \sigma$,so we may conclude.

- If the last rule applied is an elimination of $\otimes$

$$\frac{\Psi;\ \Delta_1 \vdash u : \tau \otimes \sigma \qquad \Psi;\ \Delta_2, x : \tau, y : \sigma \vdash t : \kappa}{\Psi;\ \Delta_1,\ \Delta_2 \vdash \mathtt{let}\ x \otimes y = u\ \mathtt{in}\ t : \kappa}$$

with δ_1, δ_2 the obvious decomposition of δ along $\Delta_1,\ \Delta_2$, the induction hypothesis applied to the first premise yields $\Psi';\ \Delta'_1 \models u[\gamma, \delta_1] : \tau \otimes \sigma$. In particular, this means we have $u[\gamma, \delta_1] \to^*_{\beta\varepsilon} u'$ such that $\Psi';\ \Delta'_1 \vdash_{\mathrm{NF}} u' : \tau \otimes \sigma$. By Theorem 5.4.9, it suffices to show that $\mathtt{let}\ x \otimes y = u'\ \mathtt{in}\ t[\gamma, \delta_2] \to^*_{\beta\varepsilon} v$ such that $\Psi';\ \Delta'_1,\ \Delta'_2 \models v : \kappa$ to conclude. We do so by going by induction over the judgment $\Psi';\ \Delta'_1 \vdash_{\mathrm{NF}} u' : \tau \otimes \sigma$.

  - If we have $\Psi';\ \Delta'_1 \vdash_{\mathrm{NE}} u' : \tau \otimes \sigma$, then we may use the outer inductive hypothesis $\Psi';\ \Delta'_2,\ x : \tau,\ y : \sigma \models t[\gamma, \delta_2] : \kappa$ and apply Lemma 5.4.11.
  - If we have $u' = \mathtt{let}\ x' \otimes y' = u''\ \mathtt{in}\ u'''$, applying the induction hypothesis, we have some v such that

$$\mathtt{let}\ x \otimes y = u'''\ \mathtt{in}\ t[\gamma, \delta_2] \to^*_{\beta\varepsilon} v \qquad \text{and} \qquad \Psi';\ \Delta', x' : \tau', y' : \sigma' \models v : \kappa$$

    We may thus conclude using the sequence of reductions below and Lemma 5.4.11

$$\begin{aligned}
&\mathtt{let}\ x \otimes y = \mathtt{let}\ x' \otimes y' = u''\ \mathtt{in}\ u'''\ \mathtt{in}\ t[\gamma, \delta_2] \\
&\quad \to_\varepsilon \mathtt{let}\ x' \otimes y' = u''\ \mathtt{in}\ \mathtt{let}\ x \otimes y = u'''\ \mathtt{in}\ t[\gamma, \delta_2] \\
&\quad \to^*_{\beta\varepsilon} \mathtt{let}\ x' \otimes y' = u''\ \mathtt{in}\ v
\end{aligned}$$

  - We proceed similarly if $u' = \mathsf{case}(u'', x'.u''', y'.u'''')$, $\mathtt{let}\ () = u''\ \mathtt{in}\ u'''$ or $\pi_i(u'')$.
  - Finally, if $u' = u'' \otimes u'''$, we apply the outer induction hypothesis with the substitution $\gamma, \delta_2, u''/x, u'''/y$ to conclude. □

To finish proving our normalization result (Theorem 5.4.1), instantiate Theorem 5.4.12 in the case of a trivial substitution ($t_i = x_i$ and $u_j = a_j$) using Corollary 5.4.10 and conclude with Theorem 5.4.9.

5.4.2. **More useful syntactic properties.** The material presented here will be useful for the proofs of Proposition 5.1.7 and Lemma 5.2.5.

Definition 5.4.13. Write $\sqsubseteq_+$ for the least preorder relation over $\lambda\ell^{\oplus\&}$ types satisfying the following for every types τ and σ

$$\tau, \sigma \sqsubseteq_+ \tau \otimes \sigma \quad \tau, \sigma \sqsubseteq_+ \tau \oplus \sigma \quad \tau, \sigma \sqsubseteq_+ \tau \,\&\, \sigma \quad \sigma \sqsubseteq_+ \tau \multimap \sigma \quad \sigma \sqsubseteq_+ \tau \to \sigma$$

We say that τ is a *strictly positive subtype* of σ whenever $\tau \sqsubseteq_+ \sigma$.

Definition 5.4.14. A context $\Psi;\ \Delta$ is called *consistent* if there is no term t such that $\Psi;\ \Delta \vdash t : 0$.

Lemma 5.4.15. *A context $\Psi;\ \Delta$ is inconsistent if and only if there is a neutral term t such that $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : 0$. Furthermore, if $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau$, the last typing rule applied has one premise $\Psi';\ \Delta' \vdash_{\mathrm{NE}} u : \tau'$ and $\Psi;\ \Delta$ is consistent, then so is $\Psi';\ \Delta'$.*

Proof. The first point is an easy corollary of Theorem 5.4.1. The second point follows from a case analysis, using the following facts:

- If $\Psi,\ \Psi';\ \Delta,\ \Delta'$ is consistent, then so is $\Psi;\ \Delta$.
- If $\Psi;\ \Delta,\ \Delta'$ is consistent and $\Psi;\ \Delta' \vdash t : \tau$, then $\Psi;\ \Delta,\ x : \tau$ is consistent. □

Lemma 5.4.16. *If $\Psi;\ \Delta$ is consistent and $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau$, then there is a variable in $\Psi;\ \Delta$ of type σ with $\tau \sqsubseteq_+ \sigma$.*

Proof. By induction on the judgement $\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau$.

- If the last rule applied was a variable lookup.

$$\frac{}{\Psi;\ x : \tau \vdash_{\mathrm{NE}} x : \tau} \qquad \frac{}{\Psi, x : \tau;\ \cdot \vdash_{\mathrm{NE}} x : \tau}$$

  then the conclusion immediately follows.
- The more interesting cases are those of the elimination rules for $\multimap$, $\otimes$ and $\&$.

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \multimap \sigma \qquad \Psi;\ \Delta' \vdash_{\mathrm{NF}} u : \tau}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} t\ u : \sigma}$$

$$\frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}} t : \tau \otimes \sigma \qquad \Psi;\ \Delta,\ x : \tau,\ y : \sigma \vdash_{\mathrm{NE}} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \kappa} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \,\&\, \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NE}} \pi_1(t) : \tau}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \,\&\, \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NE}} \pi_2(t) : \sigma} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \sigma \oplus \tau \qquad \Psi;\ \Delta',\ x : \sigma \vdash_{\mathrm{NE}} u : \kappa \qquad \Psi;\ \Delta',\ y : \tau \vdash_{\mathrm{NE}} v : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} \mathsf{case}(t, x.u, y.v) : \kappa}$$

$$\frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}} t : \mathbf{I} \qquad \Psi;\ \Delta \vdash_X u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_X \mathtt{let}\ () = t\ \mathtt{in}\ u : \kappa} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : 0}{\Psi\ \Delta \vdash_{\mathrm{NE}} \mathsf{abort}(t) : \tau}$$

  The treatment of $\&$ and $\mathbf{I}$ is rather straightforward and 0 is ruled out because $\Psi;\ \Delta$ is assumed to be consistent, so we only explain the inductive step for $\multimap$, $\otimes$ and $\oplus$.
  – If the last rule applied is the elimination of a linear arrow

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \tau \multimap \sigma \qquad \Psi;\ \Delta' \vdash_{\mathrm{NF}} u : \tau}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} t\ u : \sigma}$$

    then the induction hypothesis applied to the first premise means that there is a variable x in $\Psi;\ \Delta$ of type κ such that $\tau \multimap \sigma \sqsubseteq_+ \kappa$, and we may conclude since $\sigma \sqsubseteq_+ \tau \multimap \sigma$.
  – If the last rule applied is the elimination of a tensor product

$$\frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}} t : \tau \otimes \sigma \qquad \Psi;\ \Delta,\ x : \tau,\ y : \sigma \vdash_{\mathrm{NE}} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \kappa}$$

    then the induction hypothesis applied to the first premise yields a variable z in $\Psi;\ \Delta,\ x : \tau,\ y : \sigma$ of type ζ with $\kappa \sqsubseteq_+ \zeta$. If $z \notin \{x, y\}$, then z occurs in $\Psi;\ \Delta$ and we may conclude. Otherwise, suppose that $z = x$; applying the induction hypothesis to the second premise, we know that $\tau \otimes \sigma \sqsubseteq_+ \tau \sqsubseteq_+ \zeta'$ for a ζ' being the type of some variable in Δ' or a $\zeta' = !\zeta''$ with ζ'' being the type of some variable in Ψ. Therefore, we may conclude since $\kappa \sqsubseteq_+ \tau \sqsubseteq_+ \tau \otimes \sigma \sqsubseteq_+ \zeta'$. The case of $z = y$ is treated similarly.
- If the last rule applied is the elimination of a coproduct

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}} t : \sigma \oplus \tau \qquad \Psi;\ \Delta',\ x : \sigma \vdash_{\mathrm{NE}} u : \kappa \qquad \Psi;\ \Delta',\ y : \tau \vdash_{\mathrm{NE}} v : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}} \mathsf{case}(t, x.u, y.v) : \kappa}$$

  then, by the inductive hypothesis applied to the second premise, there is ζ with $\kappa \sqsubseteq_+ \zeta$ such that
  (1) either there is a variable in $\Psi;\ \Delta'$ with type ζ
  (2) or $\zeta = \sigma$.
  In the first case, we may directly conclude. Otherwise, the induction hypothesis applied to the first premise states that there is ζ'' with $\sigma \oplus \tau \sqsubseteq_+ \zeta''$ so that ζ'' is a type of some variable in Ψ or Δ. Hence, we have $\kappa \sqsubseteq_+ \sigma \sqsubseteq_+ \sigma \oplus \tau \sqsubseteq_+ \zeta''$ and we may conclude. □

5.4.3. **Proof of Proposition 5.1.7.** Let us show that the Church encoding provides a bijection between ranked trees over $\mathbf{\Sigma}$ and closed inhabitants of $\mathtt{Tree}_{\mathbf{\Sigma}}$.

For the sake of this proof, let us assume that ranked alphabets $\mathbf{\Sigma}$ are ordered. Recall that if $t \in \mathbf{Tree}(\mathbf{\Sigma})$, we write $\underline{t} : \mathtt{Tree}_{\mathbf{\Sigma}}$ for its Church encoding. When $\Sigma = \{a_1, \dots, a_k\}$, $\underline{t}$ has shape $\lambda^! a_1 . \dots . \lambda^! a_k . t^\circ$. for some neutral term t°. Let us adopt this notation for a map $t \mapsto t^\circ$, mapping trees t to terms $\widetilde{\mathbf{\Sigma}}; \cdot \vdash t^\circ : \mathbb{o}$, and let us abbreviate the sequence of λ-abstractions $\lambda^! a_1 . \dots . \lambda^! a_k .$ as $\lambda^! \mathbf{\Sigma}$ for arbitrary (ordered) ranked alphabets $\mathbf{\Sigma}$.

We use those conventions to show that the map $t \mapsto \underline{t}$ is surjective (where it is understood that the codomain consists of terms *up to $\beta\eta$-equivalence*.

Lemma 5.4.17. *Fix a ranked alphabet $\widetilde{\mathbf{\Sigma}}$. For every typed normal term u, we have*

(1) *if $\widetilde{\mathbf{\Sigma}}; \cdot \vdash_{\mathrm{NF}} u : \mathbb{o}$, then there is $t \in \mathbf{Tree}(\mathbf{\Sigma})$ such that $u = t^\circ$.*
(2) *if $\widetilde{\mathbf{\Sigma}}; \cdot \vdash_{\mathrm{NE}} u : \mathbb{o} \multimap \dots \multimap \mathbb{o}$ where the type of u has k arguments, then there exists $a \in \Sigma$, a list of trees $t_1, \dots, t_{|\mathrm{ar}(a)|-k} \in \mathbf{Tree}(\mathbf{\Sigma})$ such that $u = a\ t_1^\circ\ \dots\ t_{|\mathrm{ar}(a)|-k}^\circ$.*

Proof. We proceed by induction over the typing judgment of the normal form u. Many cases are easily seen to not arise (typically, constructor for various datatypes). Most eliminators can also be ignored because of Lemma 5.4.16. For instance, suppose $u = \mathsf{case}(v, x.w, y.w')$. Then it would means that we had some $\widetilde{\mathbf{\Sigma}}; \cdot \vdash_{\mathrm{NE}} v : \tau \oplus \sigma$ for some τ, σ, but that cannot be the case as $\tau \oplus \sigma$ is never a strictly positive subtype of some $\mathbb{o} \multimap \dots \multimap \mathbb{o}$.

As a consequence, there are only two cases of interest: the variable case and (linear) function application.

- If u is a variable of type $\mathbb{o}$, then the first result (and, as a consequence, the second) result is immediate: u is the Church encoding of a tree with a single leaf. If it is a variable of type $\mathbb{o} \multimap \dots \multimap \mathbb{o}$, the first claim is vacuously true and the second is also immediate.
- If u is a function application $v\ w$, then we have that $\widetilde{\mathbf{\Sigma}}; \cdot \vdash_{\mathrm{NE}} v : \mathbb{o} \multimap \dots \multimap \mathbb{o}$ in both cases. So we may apply the inductive hypothesis to obtain some $a \in \Sigma$ and trees $t_1, \dots, t_l$ such that $v = a\ t_1^\circ\ \dots\ t_l^\circ$. We also have that $\widetilde{\mathbf{\Sigma}}; \cdot \vdash_{\mathrm{NF}} w : \mathbb{o}$, so we have some tree t_{l+1} such that $w =_{\beta\eta} t_{l+1}^\circ$. Altogether, we thus have $u = a\ t_1^\circ\ \dots\ t_{l+1}^\circ$ as expected of the second item. If the first item is not vacuously true, we have that $l + 1 = |A|$, and thus, $a\ t_1^\circ\ \dots\ t_{l+1}^\circ = (a(t_1, \dots, t_{l+1}))^\circ$ as required. □

Given two *ordered* ranked alphabets $\mathbf{\Sigma}$ and $\mathbf{\Gamma}$, write $\mathbf{\Sigma} \otimes \mathbf{\Gamma}$ for the ordered ranked alphabet with letters $\Sigma + \Gamma$ determined by $\mathsf{in}_1(a) < \mathsf{in}_2(b)$ for $a \in \Sigma$, $b \in \Gamma$ and where the order is lifted from $\mathbf{\Sigma}$ and $\mathbf{\Gamma}$ otherwise.

Lemma 5.4.18. *Fix ranked alphabets $\mathbf{\Sigma}, \mathbf{\Gamma}$. If we have $\widetilde{\mathbf{\Sigma}}; \cdot \vdash u : \mathtt{Tree}_{\mathbf{\Gamma}}$, then there exists some $t \in \mathbf{Tree}(\mathbf{\Sigma} \otimes \mathbf{\Gamma})$ such that $u =_{\beta\eta} \lambda^! \mathbf{\Gamma} . t^\circ$.*

Proof. First, we use Theorem 5.4.1 to suppose that u is under normal form, and we proceed by induction over the size of $\mathbf{\Gamma}$. If it is empty, then the result follows from the first item of Lemma 5.4.17. Otherwise $\mathbf{\Gamma} = \mathbf{S} \otimes \mathbf{\Gamma}'$ for some singleton alphabet $\mathbf{S}$ with letter b. Then, a quick case analysis shows that, as in Lemma 5.4.17, most cases can be ignored due to the typing of u, and because of considerations based on Lemma 5.4.16. There is only one interesting case which is the non-linear λ-abstraction.

$$\frac{\widetilde{\mathbf{\Sigma}},\ \widetilde{\mathbf{S}}; \cdot \vdash v : \mathtt{Tree}_{\mathbf{\Gamma}}}{\widetilde{\mathbf{\Sigma}}; \cdot \vdash \lambda^! b . v : \mathtt{Tree}_{\mathbf{S} \otimes \mathbf{\Gamma}}}$$

We may apply the induction hypothesis as $\widetilde{\mathbf{\Sigma} \otimes \mathbf{S}} = \widetilde{\mathbf{\Sigma}}, \widetilde{\mathbf{S}}$ and get that $u =_{\beta\eta} \lambda^! b . v = \lambda^! b . \lambda^! \mathbf{\Gamma} . t^\circ = \lambda^! \mathbf{S} \otimes \mathbf{\Gamma} . t^\circ$. □

By instantiating this latest lemma with $\mathbf{\Gamma}$ empty, we can thus deduce that the map $t \mapsto \underline{t}$ is surjective. We may also show that it is injective by exhibiting a left-inverse map, using a semantic interpretation of $\lambda\ell^{\oplus\&}$ into Set: with $\mathbf{\Sigma}$ fixed, use the cartesian-closed structure and coproducts to interpret $\lambda\ell^{\oplus\&}$ with the interpretation of $\mathbb{o}$ being $\mathbf{Tree}(\mathbf{\Sigma})$. This yields a map from terms $t : \mathtt{Tree}_{\mathbf{\Sigma}}$ to $(\mathbf{Tree}(\mathbf{\Sigma}) \to \ldots \to \mathbf{Tree}(\mathbf{\Sigma})) \to \ldots \to \mathbf{Tree}(\mathbf{\Sigma})$, where the arguments correspond to the arity of tree constructors; feed the actual constructors to this function to recover a tree in $\mathbf{Tree}(\mathbf{\Sigma})$.

It is straightforward to check that this map is indeed a left inverse of $t \mapsto \underline{t}$, by induction over t. Hence the map $t \mapsto \underline{t}$ is bijective.

5.4.4. **Proof of Lemma 5.2.5.** We now analyse the terms of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\sigma] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$.

Lemma 5.4.19. *Let $\tau = \kappa_1 \to \ldots \to \kappa_k \to \kappa'$ be a type and s a distinguished variable of type τ. Let $\mathbf{\Sigma}$ be a ranked alphabet such that $\widetilde{\mathbf{\Sigma}};\ s : \tau$ is consistent. Then, if there is $k' < k$ such that $\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t : \kappa_{k'+1} \to \ldots \to \kappa_k \to \kappa$, there are also terms $d_1, \ldots, d_{k'}$ such that*

$$t =_{\beta\eta} s\ d_1\ \ldots\ d_{k'} \quad \text{and} \quad \widetilde{\mathbf{\Sigma}};\ \cdot \vdash_{\mathrm{NF}} d_i : \kappa_i \qquad \text{for } i \in \{1, \ldots, k'\}$$

Proof. By induction over k'. Note that t being neutral is essential here. □

Lemma 5.4.20. *Let $\tau = \kappa_1 \to \ldots \to \kappa_k \to \kappa'$ be a type with κ' purely linear and s a distinguished variable of type τ. Let $\mathbf{\Sigma}$ be a ranked alphabet such that $\widetilde{\mathbf{\Sigma}};\ s : \tau$ is consistent and t be a term such that $\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t : \sigma$ for some σ such that $\sigma \sqsubseteq_+ \kappa'$ or of the shape $\mathbb{o} \multimap \ldots \multimap \mathbb{o}$. Then, there are terms o, d_1, …, d_k such that $t =_{\beta\eta} o\ (s\ d_1\ \ldots\ d_k)$ and*

$$\widetilde{\mathbf{\Sigma}};\ \cdot \vdash o : \kappa' \multimap \sigma \qquad\qquad \widetilde{\mathbf{\Sigma}};\ \cdot \vdash_{\mathrm{NF}} d_i : \kappa_i \quad \text{for } i \in \{1, \ldots, k\}$$

Proof. We proceed by induction over a derivation of $\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t : \sigma$. Note that to apply the induction hypothesis, we need to ensure that every context under consideration is consistent. We keep this check implicit as it always follows from Lemma 5.4.15.

- If the last rule applied is a variable lookup, then the term in question must be s itself. Furthermore, we must have $k = 0$, so we may simply take $o = \lambda x.x$ to conclude.
- If the last rule considered is the following instance of the application rule

$$\frac{\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t : \sigma' \multimap \sigma \qquad \widetilde{\mathbf{\Sigma}};\ \cdot \vdash_{\mathrm{NF}} u : \sigma'}{\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t\ u : \sigma}$$

  then, by Lemma 5.4.16 (applied on the first premise), it means that we have

$$\text{either } \sigma' \multimap \sigma \sqsubseteq_+ \tau \qquad \text{or } \sigma \multimap \sigma' = \mathbb{o} \multimap \ldots \multimap \mathbb{o}$$

  In the first case, we can further see that $\sigma' \multimap \sigma \sqsubseteq_+ \kappa'$, so in both cases the induction hypothesis can be applied to the first premise to yield some o' and $d_1, \ldots, d_k$ such that $o'\ (s\ d_1 \ldots d_k) =_{\beta\eta} t$, and we may set $o = \lambda s.\ o'\ s\ u$ to conclude.
- If the last rule considered is the following instance of the application rule

$$\frac{\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t : \sigma' \to \sigma \qquad \widetilde{\mathbf{\Sigma}};\ \cdot \vdash_{\mathrm{NF}} u : \sigma'}{\widetilde{\mathbf{\Sigma}};\ s : \tau \vdash_{\mathrm{NE}} t\ u : \sigma}$$

  then, by Lemma 5.4.16 (applied on the first premise), it means that we have

$$\text{either } \sigma' \to \sigma \sqsubseteq_+ \tau \qquad \text{or } \sigma \to \sigma' = \mathbb{o} \multimap \ldots \multimap \mathbb{o}$$

  The second alternative is absurd, and the first leads to $\sigma = \kappa'$ and $\sigma' = \kappa_k$. Therefore, we may apply Lemma 5.4.19 to get terms $d_1, \ldots, d_{k-1}$ in normal form such that $t =_{\beta\eta} s\ d_1\ \ldots\ d_{k-1}$. We then set d_k to be u and o to be the identity to conclude.

- If the last rule considered is the other instance of the application rule

$$\frac{\widetilde{\Sigma};\ \Delta \vdash_{\mathrm{NE}} t : \sigma' \multimap \sigma \qquad \widetilde{\Sigma};\ \Delta',\ s : \tau \vdash_{\mathrm{NF}} u : \sigma'}{\widetilde{\Sigma};\ \Delta,\ \Delta',\ s : \tau \vdash_{\mathrm{NE}} t\ u : \sigma}$$

  by Lemma 5.4.16 applied to the first premise, we know that $\sigma' = \mathbb{0}$. Therefore, we may apply the induction hypothesis to the second premise to obtain $d_1, \ldots, d_k$ and o' such that $u =_{\beta\eta} o'\ (s\ d_1\ \ldots\ d_k)$, in which case, $t\ u =_{\beta\eta} (\lambda x.\ t\ (o\ x))\ (s\ d_1\ \ldots\ d_k)$. We conclude by setting $o = \lambda z.\ t\ (o'\ z)$.
- If the last rule considered is the following instance of the elimination of tensor products

$$\frac{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} t : \zeta_1 \otimes \zeta_2 \qquad \widetilde{\Sigma};\ x_1 : \zeta_1,\ x_2 : \zeta_2 \vdash_{\mathrm{NE}} u : \sigma}{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \sigma}$$

  then, by the induction hypothesis (which is applicable because of Lemma 5.4.16), there are $d_1, \ldots, d_k$ and o' such that $t =_{\beta\eta} o'\ (s\ d_1\ \ldots\ d_k)$, in which case, we conclude by setting $o = \lambda z.\ \mathtt{let}\ x \otimes y = o'\ z\ \mathtt{in}\ u$.
- The last rule considered cannot be the following instance of the elimination of tensor products

$$\frac{\widetilde{\Sigma};\ \cdot \vdash_{\mathrm{NE}} t : \zeta_1 \otimes \zeta_2 \qquad \widetilde{\Sigma};\ s : \tau,\ x_1 : \zeta_1,\ x_2 : \zeta_2 \vdash_{\mathrm{NE}} u : \sigma}{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \sigma}$$

  as Lemma 5.4.16 would require that $\zeta_1 \otimes \zeta_2$ be a strictly positive subtype of some $\mathbb{0} \multimap \ldots \multimap \mathbb{0}$.
- If the last rule considered types a projection

$$\frac{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} t : \sigma_1\ \&\ \sigma_2}{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} \pi_i(t) : \sigma_i}$$

  then the induction hypothesis yields terms o' and $d_1, \ldots, d_k$ such that $t =_{\beta\eta} o'\ (s\ d_1 \ldots d_k)$. We may set $o = \lambda z.\ \pi_i(o'\ z)$ to conclude.
- If the last rule applied is an elimination of a coproduct

$$\frac{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} t : \zeta_1 \oplus \zeta_2 \qquad \widetilde{\Sigma};\ x : \zeta_1 \vdash_{\mathrm{NE}} u : \sigma \qquad \widetilde{\Sigma};\ y : \zeta_2 \vdash_{\mathrm{NE}} v : \sigma}{\widetilde{\Sigma};\ \cdot \vdash_{\mathrm{NE}} \mathsf{case}(t, x.u, y.v) : \sigma}$$

  then, the induction hypothesis (applicable because of Lemma 5.4.16) yields terms o' and $d_1, \ldots, d_k$ such that $t =_{\beta\eta} o'\ (s\ d_1 \ldots d_k)$. We may set $o = \lambda z.\ \mathsf{case}(o'\ z, x.u, y.v)$ to conclude.
- The last rule applied cannot be one of the following instances of the elimination of a coproduct because of Lemma 5.4.16 applied to the first premise:

$$\frac{\widetilde{\Sigma};\ \cdot \vdash_{\mathrm{NE}} t : \zeta_1 \oplus \zeta_2 \qquad \widetilde{\Sigma};\ s : \tau,\ x : \zeta_1 \vdash_{X} u : \sigma \qquad \widetilde{\Sigma};\ y : \zeta_2 \vdash_{X} v : \sigma}{\widetilde{\Sigma};\ s : \tau \vdash_{X} \mathsf{case}(t, x.u, y.v) : \sigma}$$

$$\frac{\widetilde{\Sigma};\ \cdot \vdash_{\mathrm{NE}} t : \zeta_1 \oplus \zeta_2 \qquad \widetilde{\Sigma};\ x : \zeta_1 \vdash_{X} u : \sigma \qquad \widetilde{\Sigma};\ s : \tau,\ y : \zeta_2 \vdash_{X} v : \sigma}{\widetilde{\Sigma};\ s : \tau \vdash_{X} \mathsf{case}(t, x.u, y.v) : \sigma}$$

- If the last rule applied is an elimination of $\mathbf{I}$

$$\frac{\widetilde{\Sigma};\ s : \tau \vdash_{\mathrm{NE}} t : \mathbf{I} \qquad \widetilde{\Sigma};\ \cdot \vdash_{\mathrm{NE}} u : \sigma}{\widetilde{\Sigma};\ \cdot \vdash_{\mathrm{NE}} \mathtt{let}\ () = t\ \mathtt{in}\ u : \sigma}$$

  then, the induction hypothesis (applicable because of Lemma 5.4.16) yields terms o' and $d_1, \ldots, d_k$ such that $t =_{\beta\eta} o'\ (s\ d_1 \ldots d_k)$. We may set $o = \lambda z.\ \mathtt{let}\ () = o'\ z\ \mathtt{in}\ u$ to conclude.
- The last rule applied cannot be one of the other instances of $\mathbf{I}$ because of Lemma 5.4.16.

- Finally, since the context under consideration are assumed to be consistent, the last rule applied cannot be an elimination of 0. □

Lemma 5.4.21. *Let* $\mathbf{\Sigma}$ *and* $\mathbf{\Gamma}$ *be ranked alphabets. If the context* $\widetilde{\mathbf{\Gamma}}$; $s : \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]$ *is inconsistent, then* $\mathbf{Tree}(\mathbf{\Sigma}) = \varnothing$.

Proof. Let us write $[\![-]\!]$ for a semantic interpretation of $\lambda\ell^{\oplus\&}$ types as (classical) propositions following the usual type/proposition mapping (i.e., $[\![\tau \multimap \sigma]\!] = [\![\tau \to \sigma]\!] = [\![\tau]\!] \Rightarrow [\![\sigma]\!]$, $[\![\tau \otimes \sigma]\!] = [\![\tau \,\&\, \sigma]\!] = [\![\tau]\!] \wedge [\![\sigma]\!]$, $[\![0]\!] = \bot$, …) and such that

$$[\![\mathbb{o}]\!] \;\Leftrightarrow\; \mathbf{Tree}(\mathbf{\Gamma}) \neq \varnothing$$

It is easy to check that, under the usual conjunctive interpretation of contexts, if Ψ; $\Delta \vdash t : \tau$, then $[\![\Psi]\!] \wedge [\![\Delta]\!] \Rightarrow [\![\tau]\!]$ holds. Further, our choice for $[\![\mathbb{o}]\!]$ means that $[\![\widetilde{\mathbf{\Gamma}}]\!]$ holds, so, if our context $\widetilde{\mathbf{\Gamma}}$; $s : \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]$ is inconsistent, $\neg[\![\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]]\!]$ holds. Now, assume that $\mathbf{Tree}(\mathbf{\Sigma})$ has an inhabitant t. There is a corresponding Church encoding $\underline{t}$ of type $\mathtt{Tree}_{\mathbf{\Sigma}}$, which has also type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]$. Hence, $[\![\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]]\!]$ also holds, leading to a contradiction. □

We can now prove Lemma 5.2.5. Assume that we have a closed $\lambda\ell^{\oplus\&}$-term t of type $\mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \multimap \mathtt{Tree}_{\mathbf{\Gamma}}$ with $\mathbf{Tree}(\mathbf{\Sigma}) \neq \varnothing$, so that we may safely assume $\widetilde{\mathbf{\Gamma}}$; $s : \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa]$ to be consistent. By η-expansion, we have is of the shape

$$t =_{\beta\eta} \lambda w.\lambda^{!} b_1 . \ldots \lambda^{!} b_k . t\ w\ b_1 \ldots b_k$$

and $\widetilde{\mathbf{\Gamma}}$; $s : \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \vdash t\ w\ b_1 \ldots b_k : \mathbb{o}$. By normalization of $\lambda\ell^{\oplus\&}$, there is t' such that

$$t' =_{\beta\eta} t\ w\ b_1 \ldots b_k \qquad \text{and} \qquad \widetilde{\mathbf{\Gamma}};\ s : \mathtt{Tree}_{\mathbf{\Sigma}}[\kappa] \vdash_{\mathrm{NF}} t' : \mathbb{o}$$

For the latter, note that an easy case analysis shows that every (open) term of type $\mathbb{o}$ is in fact neutral, so we may conclude by applying Lemma 5.4.20 and the fact that $=_{\beta\eta}$ is a congruence over terms.

5.4.5. **Focusing.** We now introduce (partially) *focused normal forms*. These terms are split into three categories defined by mutual recursion:

- the general focused normal forms ($\mathrm{NF}_{\mathfrak{f}}$). We write Ψ; $\Delta \vdash_{\mathrm{NF}_{\mathfrak{f}}} t : A$ to say that t is a focused normal form such that Ψ; $\Delta \vdash t : A$.
- the focused neutral terms ($\mathrm{NE}_{\mathfrak{f}}$). We write Ψ; $\Delta \vdash_{\mathrm{NE}_{\mathfrak{f}}} t : A$ to say that t is a focused neutral form such that Ψ; $\Delta \vdash t : A$.
- the focused negative neutral terms ($\mathrm{NE}_{\mathfrak{f}}^{-}$). We write Ψ; $\Delta \vdash_{\mathrm{NE}_{\mathfrak{f}}^{-}} t : A$ to say that t is a negative focused neutral form such that Ψ; $\Delta \vdash t : A$.

The inductive rules for building those normal forms are presented in Figure 5.4.5.

Theorem 5.4.22. *Any typed term* t *is* $\beta\eta$*-equivalent to a term in* focused *normal form.*

Proof sketch. We must prove that for every term t such that Ψ; $\Delta \vdash t : \tau$, we have some $\beta\eta$-equivalent t' such that Ψ; $\Delta \vdash_{\mathrm{NF}_{\mathfrak{f}}} t' : \tau$.

Focusing and normalization can be done simultaneously, but let us sketch how to get focused forms from our previous notion of normal forms (§5.4.1); this allows to conclude using the normalization theorem that we already have (Theorem 5.4.1). Suppose that we have Ψ; $\Delta \vdash_{\mathrm{NF}} t : \tau$. We first claim that we have t' such that $t \to_{\varepsilon}^{*} t'$ and $t' \not\to_{\varepsilon}$; this can be shown by noticing that the following $\mathbb{N}$-valued complexity measure $\| - \|$ on $\lambda\ell^{\oplus\&}$-terms is

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}_\mathfrak{f}} t : \tau}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} t : \tau} \qquad \frac{\Psi;\ \Delta,\ x : \tau \vdash_{\mathrm{NF}_\mathfrak{f}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} \lambda x.t : \tau \multimap \sigma} \qquad \frac{\Psi,\ x : \tau;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} \lambda^{!} x.t : \tau \to \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} t : \tau \quad \Psi;\ \Delta' \vdash_{\mathrm{NF}_\mathfrak{f}} u : \sigma}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NF}_\mathfrak{f}} t \otimes u : \tau \otimes \sigma}$$

$$\frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau \otimes \sigma \quad \Psi;\ \Delta,\ x : \tau,\ y : \sigma \vdash_{\mathrm{NF}_\mathfrak{f}} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NF}_\mathfrak{f}} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \kappa} \qquad \frac{}{\Psi;\ \cdot \vdash_{\mathrm{NF}_\mathfrak{f}} () : \mathbf{I}}$$

$$\frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \mathbf{I} \qquad \Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NF}_\mathfrak{f}} \mathtt{let}\ () = t\ \mathtt{in}\ u : \kappa} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} t : \tau \qquad \Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} u : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} \langle t, u \rangle : \tau \,\&\, \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} t : \tau}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} \mathsf{in}_1(t) : \tau \oplus \sigma} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} \mathsf{in}_2(t) : \tau \oplus \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \sigma \oplus \tau \qquad \Psi;\ \Delta',\ x : \sigma \vdash_{\mathrm{NF}_\mathfrak{f}} u : \kappa \qquad \Psi;\ \Delta',\ y : \tau \vdash_{\mathrm{NF}_\mathfrak{f}} v : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NF}_\mathfrak{f}} \mathsf{case}(t, x.u, y.v) : \kappa}$$

$$\frac{}{\Psi;\ \Delta \vdash_{\mathrm{NF}_\mathfrak{f}} \langle\rangle : \top}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau}{\Psi;\ \Delta \vdash_{\mathrm{NE}_\mathfrak{f}} t : \tau} \qquad \frac{\Psi;\ \Delta' \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau \otimes \sigma \quad \Psi;\ \Delta,\ x : \tau,\ y : \sigma \vdash_{\mathrm{NE}_\mathfrak{f}} u : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}_\mathfrak{f}} \mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u : \kappa}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \sigma \oplus \tau \qquad \Psi;\ \Delta',\ x : \sigma \vdash_{\mathrm{NE}_\mathfrak{f}} u : \kappa \qquad \Psi;\ \Delta',\ y : \tau \vdash_{\mathrm{NE}_\mathfrak{f}} v : \kappa}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}_\mathfrak{f}} \mathsf{case}(t, x.u, y.v) : \kappa}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : 0}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}_\mathfrak{f}} \mathsf{abort}(t) : \sigma}$$

$$\frac{}{\Psi;\ \Delta,\ x : \tau \vdash_{\mathrm{NE}^-_\mathfrak{f}} x : \tau} \qquad \frac{}{\Psi,\ x : \tau;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} x : \tau} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau \,\&\, \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} \pi_1(t) : \tau}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau \,\&\, \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} \pi_2(t) : \sigma} \qquad \frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau \multimap \sigma \qquad \Psi;\ \Delta' \vdash_{\mathrm{NF}_\mathfrak{f}} u : \tau}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NE}^-_\mathfrak{f}} t\ u : \sigma}$$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t : \tau \to \sigma \qquad \Psi;\ \cdot \vdash_{\mathrm{NF}_\mathfrak{f}} u : \tau}{\Psi;\ \Delta \vdash_{\mathrm{NE}^-_\mathfrak{f}} t\ u : \sigma}$$

FIGURE 5.4.5. Focused normal forms (top) / neutral forms (middle) / negative neutral forms (bottom) for $\lambda\ell^{\oplus\&}$-terms.

strictly decreasing along $\to_\varepsilon$:

$$\begin{array}{lcllcl}
\|x\| & = & 0 & \|()\| & = & 0 \\
\|\langle\rangle\| & = & 0 & \|\mathsf{in}_1(t)\| & = & \|t\| \\
\|\mathsf{in}_2(t)\| & = & \|t\| & \|\pi_1(t)\| & = & \|t\| \\
\|\pi_2(t)\| & = & \|t\| & \|t\ u\| & = & \|t\| + \|u\| \\
\|t \otimes u\| & = & \|t\| + \|u\| & \|\langle t, u\rangle\| & = & \|t\| + \|u\| \\
\|\mathsf{abort}(t)\| & = & 1 + 2\|t\| & \|\mathtt{let}\ x \otimes y = t\ \mathtt{in}\ u\| & = & 1 + 2\|t\| + \|u\| \\
\|\mathsf{case}(t, x.u, y.v)\| & = & 1 + 2\|t\| + \|u\| + \|v\| & &
\end{array}$$

By Lemma 5.4.2, we have that $\Psi;\ \Delta \vdash_{\mathrm{NF}} t' : \tau$. This combined with the fact that we have $t' \not\to_\varepsilon$ allows to show that we have $\Psi;\ \Delta \vdash_{\mathrm{NF_f}} t' : \tau$ by induction on the derivation. □

5.4.6. **Proof of Lemma 5.3.9.** This is the last thing that needs to be proved before we can be done with this chapter. Let t be a $\lambda\ell^{\oplus\&}$-term with $\widetilde{\Sigma},\ s : \mathtt{Str}_\Gamma[\sigma];\ \cdot\ \vdash t : \sigma'$ for σ and σ' purely linear. Since Lemma 5.3.9 depends only on the $\beta\eta$-equivalence class of t, we may replace it with a focused normal form. The following result shows that Lemma 5.3.9 can be established by a straightforward induction on the number of occurrences of $s : \mathtt{Str}_\Gamma[\sigma]$ in t – a parameter which is *not* invariant under $=_{\beta\eta}$.

Lemma 5.4.23. *Let $\tau = \kappa_1 \to \ldots \to \kappa_k \to \kappa'$ be a type with κ' purely linear and s a distinguished variable of type τ. Let Δ and Ψ be purely linear contexts and t be a term such that $\Psi,\ s : \tau; \Delta \vdash_{\mathrm{NF_f}} t : \sigma$ for some purely linear σ. Suppose further that there is at least one occurrence of s in t. Then, there are terms o, d_1, …, d_k such that $t =_{\beta\eta} o\ \langle s\ d_1\ \ldots\ d_k, ()\rangle$ and*

$$\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NF_f}} o : (\kappa' \,\&\, \mathbf{I}) \multimap \sigma \qquad \Psi, s : \tau;\ \cdot \vdash_{\mathrm{NF_f}} d_i : \kappa_i \quad \text{for } i \in \{1, \ldots, k\}$$

Furthermore, there are no more occurrences of s in $o\ \langle s\ d_1\ \ldots\ d_k, ()\rangle$ than in t.

(For our intended application, take $\kappa_1 = \cdots = \kappa_{|\Sigma|} = (\sigma \multimap \sigma)$ and $\kappa_{|\Sigma|+1} = \kappa' = \sigma$.)

To prove Lemma 5.4.23 we will need two additional observations, for which we do not provide detailed proofs. Recall that the strictly positive subtype relation $\sqsubseteq_+$ has been introduced in Section 5.4.2.

Lemma 5.4.24. *If $\Psi;\ \Delta \vdash_{\mathrm{NE_f^-}} t : \tau$, then there is a variable in $\Psi;\ \Delta$ of type σ with $\tau \sqsubseteq_+ \sigma$.*

Proof idea. By induction over the derivation, much like Lemma 5.4.16. Note that we need not to assume that the context be consistent as the $\mathsf{abort}(t)$ is excluded by our restricting to negative neutral forms. □

Lemma 5.4.25. *Let $\tau = \kappa_1 \to \ldots \to \kappa_k \to \kappa'$ be a type and s a distinguished variable of type τ and $l < k$. Let $\Psi;\ \Delta$ be a purely linear contexts such that and t be a term such that $\Psi,\ s : \tau; \Delta \vdash_{\mathrm{NE_f^-}} t : \sigma_{l+1} \to \ldots \to \sigma_k \to \sigma'$ for some purely linear σ'. Then $\sigma' = \kappa'$ and $\sigma_i = \kappa_i$ for $i \geq l$ there are $d_1, \ldots, d_l$ such that*

$$t =_{\beta\eta} s\ d_1\ \ldots\ d_{k-1} \qquad \text{and} \qquad \Psi;\ \cdot \vdash_{\mathrm{NF_f}} d_i : \kappa_i \quad \text{for } i \in \{1, \ldots, k-1\}$$

Furthermore, there are no more occurrences of s in $s\ d_1\ \ldots d_{k-1}$ than in t.

Proof idea. By induction on the judgment $\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t : \sigma_{l+1} \to \ldots \to \sigma'$. This is a simplification of Lemma 5.4.19; the hypotheses can be simplified since we restrict to negative neutral forms. □

Proof of Lemma 5.4.23. We proceed by induction over the derivation of the typing judgment $\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NF_f}} t : \sigma$ to produce

$$\Psi,\ s : \tau; \Delta,\ \alpha : \kappa' \,\&\, \mathbf{I} \vdash_{\mathrm{NF_f}} o : \sigma \qquad \Psi,\ s : \tau;\ \cdot \vdash_{\mathrm{NF_f}} d_i : \kappa_i$$

such that $o[\langle s\ d_1\ \ldots\ d_k), ()\rangle\,/\alpha] =_{\beta\eta} t$ and that there be the same number of occurrences of s in both t and $(\lambda\alpha.o)\ \langle s\ d_1\ \ldots\ d_k, ()\rangle$. Since we shall need to often emulate the weakening of the variable α, we write $w_\alpha(u)$ for the term $\texttt{let}\ () = \pi_2(\alpha)\ \texttt{in}\ u$, noting that it is focused normal as long as u is either $\mathrm{NF_f}$ or $\mathrm{NE_f^-}$. For brevity, we use the notations $o', d_1 \ldots$ throughout for data obtained by applying the induction hypotheses without recalling all of the relevant property and merely explain how to build a suitable o.

- If the last rule applied is a variable lookup, then the term in question must be s itself. Furthermore, we must have $k = 0$, so we may simply take $o = \alpha$ to conclude.
- If the last rule considered is the following instance of the application rule

$$\frac{\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t : \sigma' \multimap \sigma \qquad \Psi,\ s : \tau;\ \Delta' \vdash_{\mathrm{NF_f}} u : \sigma'}{\Psi,\ s : \tau;\ \Delta,\ \Delta' \vdash_{\mathrm{NE_f^-}} t\ u : \sigma}$$

  then, we have two subcases, according to whether there is an occurrence of s in t or not.
  - If there is an occurrence of s in t, we apply the induction hypothesis to t to obtain some o' and d_is and set $o = o'\ u$ to conclude.
  - Otherwise, we may apply the induction hypothesis to u to obtain some o' and d_is and set $o = t\ o'$ to conclude.
- If the last rule considered is the following instance of the application rule

$$\frac{\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t : \sigma' \to \sigma \qquad \Psi,\ s : \tau;\ \cdot \vdash_{\mathrm{NF_f}} u : \sigma'}{\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t\ u : \sigma}$$

  then, we use Lemma 5.4.25 to obtain $d_1, \ldots, d_{k-1}$, set $d_k = u$ and $o = \alpha$ to conclude.
- If the last rule applied is a linear λ-abstraction

$$\frac{\Psi;\ \Delta,\ x : \sigma' \vdash_{\mathrm{NF_f}} t : \sigma}{\Psi;\ \Delta \vdash_{\mathrm{NF_f}} \lambda x.t}$$

  we may apply the induction hypothesis to obtain $o', d_1, \ldots$ such that $t = o'[s\ d_1\ \ldots]$. Note that x does not occur in any of the d_i and that we have $\Psi;\ \Delta,\ x : \sigma' \vdash o' : \kappa' \multimap \sigma$. So we may set $o = \lambda x.o'$ to conclude.
- t cannot be non-linear λ-abstraction as σ' is assumed to be purely linear.
- If the last rule applied is a $\otimes$-introduction

$$\frac{\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NF_f}} t : \sigma \qquad \Psi,\ s : \tau;\ \Delta' \vdash_{\mathrm{NF_f}} t' : \sigma'}{\Psi,\ s : \tau;\ \Delta,\ \Delta' \vdash_{\mathrm{NF_f}} t \otimes t' : \sigma \otimes \sigma'}$$

  then apply the induction hypothesis to the first premise if possible to get $o', d_1, \ldots$ and set $o = o' \otimes t'$; otherwise, do the analogous operation with the second premise.
- If the last rule applied is a $\otimes$-elimination

$$\frac{\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t : \sigma \otimes \sigma' \qquad \Psi,\ s : \tau;\ \Delta',\ x : \sigma,\ y : \sigma' \vdash_{\mathrm{NF_f}} u : \sigma''}{\Psi,\ s : \tau;\ \Delta \vdash_{\mathrm{NF_f}} \texttt{let}\ x \otimes y = t\ \texttt{in}\ u : \sigma''}$$

  first note that we have we have σ and σ' purely linear because of Lemma 5.4.24 applied to the first premise. We have then two subcases, according to whether s occurs in t or not. If it does, apply the induction hypothesis to the first premise to get $o', d_1, \ldots$ and set $o = \texttt{let}\ x \otimes y = o'\ \texttt{in}\ u$, otherwise apply it to the second premise and set $o = \texttt{let}\ x \otimes y = t\ \texttt{in}\ o'$.

- If the last rule applied is a pairing

$$\frac{\Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NF_f}} t:\sigma \qquad \Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NF_f}} t':\sigma'}{\Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NF_f}} \langle t,t'\rangle : \sigma \,\&\, \sigma'}$$

  then apply the induction hypothesis to the first premise $o', d_1, \ldots$ and set $o = \langle o', w_\alpha(t')\rangle$. If not possible because there is no occurrences of s in t, do the analogous thing with t'.
- If the last rule applied is a projection

$$\frac{\Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t:\sigma_1 \,\&\, \sigma_2}{\Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NE_f^-}} t:\sigma_i}$$

  we apply the induction hypothesis to obtain $o', d_1, \ldots$ and set $o = \pi_i\ o'$.
- The last rule applied cannot be an introduction of $\top$ because we need an occurrence of s.
- If the last rule applied is an introduction of $\oplus$

$$\frac{\Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NF_f}} t:\sigma_i}{\Psi,\ s:\tau;\ \Delta \vdash_{\mathrm{NF_f}} \mathsf{in}_i(t):\sigma_1 \oplus \sigma_2}$$

  we may simply apply the induction hypothesis to get o' and set $o = \mathsf{in}_i(o')$.
- If the last rule applied is an elimination of $\oplus$

$$\frac{\Psi;\ \Delta \vdash_{\mathrm{NE_f^-}} t:\sigma\oplus\sigma' \qquad \Psi;\ \Delta',\ x:\sigma \vdash_{\mathrm{NF}} u:\sigma'' \qquad \Psi;\ \Delta',\ y:\sigma' \vdash_{\mathrm{NF}} v:\sigma''}{\Psi;\ \Delta,\ \Delta' \vdash_{\mathrm{NF}} \mathsf{case}(t,x.u,y.v):\sigma''}$$

  first note that σ and σ' are necessarily purely linear because of Lemma 5.4.24. We then try to apply the induction hypothesis to one of the premise (at least one of them has an occurrence of s) to get some $o', d_1, \ldots$
  - If we can apply it to the first premise so that $t =_{\beta\eta} o'[\langle s\ d_1 \ldots d_k, ()\rangle/\alpha]$, we can conclude by setting $o = \mathsf{case}(o', x.u, y.v)$.
  - If it is applicable to the second premise so that $u =_{\beta\eta} o'[\langle s\ d_1 \ldots d_k, ()\rangle/\alpha]$, we set $o = \mathsf{case}(t, x.o', y.w_\alpha(v))$.
  - Otherwise, $v =_{\beta\eta} o'[\langle s\ d_1\ \ldots\ d_k, ()\rangle/\alpha]$ and we set $o = \mathsf{case}(t, x.w_\alpha(u), y.o')$. □

CHAPTER 6

Regular tree functions

The goal of this chapter is now to prove the last remaining item in Theorem 1.2.3: a tree-to-tree function is $\lambda\ell^{\oplus\&}$-definable (cf. Definition 5.1.9) if and only if it is regular. To do so, we follow a similar approach to Chapter 5 – which, in turn, relied on the material of Chapter 4. A lot of similar notions and theorems reappear, so we take the liberty of eliding certain proofs adaptating statements proven in those previous chapters.

The obvious starting point is to extend the notion of "automaton parameterized by a category" to tree automata (§6.1); this increased generality in inputs, however, requires some additional conditions on the category. We then present in Section 6.3 a basic *multicategory* (cf. §6.2) of registers $\mathcal{TR}^{\mathrm{m}}$ containing "trees with holes", and the corresponding streaming setting $\mathfrak{TR}$ on top of a category $\mathcal{TR}$. The usual restriction to registers containing "trees with at most one hole" is also discussed and shown to be no less expressive thanks to our basic results on the coproduct completion. This is followed by Section 6.4, which explains how the notion of single-use-restricted bottom-up (ranked|register) tree transducer (BRTT) introduced in [AD17] – cf. §2.6 – can be shown to have the same expressiveness as "stateful tree automata over $\mathfrak{TR}_{\&}$". We shall call the latter $\mathfrak{TR}_{\&}$-BRTTs. Finally, in Section 6.5, we show that $\mathcal{TR}_{\oplus}$ has internal homsets $\iota_{\oplus}(R) \multimap \iota_{\oplus}(S)$ and conclude that $\mathcal{TR}_{\oplus\&}$ is monoidal closed. After a few more general categorical considerations (§6.6), we conclude the proof of our main theorem on trees in Section 6.7.

For the rest of this chapter, we fix a ranked alphabet $\mathbf{\Gamma}$ so that we may focus on outputs contained in $\mathbf{Tree}(\mathbf{\Gamma})$.

Remark 6.0.1. We were drawn to work with BRTTs because of our previous success with SSTs and the title "Streaming tree transducers" of [AD17] (we already recalled the explanation for this title at the beginning of Section 2.6). But the much older model of *macro tree transducers* (MTTs) [EV85] can also be seen as a version of streaming string transducers for trees; it is actually rather similar to BRTTs. In fact, single-use-restricted MTTs with regular lookahead [EM99], which compute regular tree functions, are much closer than single-use-restricted BRTTs to our categorical $\mathfrak{TR}_{\oplus\&}$-BRTTs! Had we worked with MTTs from the beginning, we would therefore have saved some efforts. But we had already written up a technical development relying on BRTTs before this realization; it still contains some interesting categorical constructions involving *coherence spaces*.

It was not immediately apparent to us that MTTs generalize SSTs to trees for the following reason: while MTTs can be reformulated as an isomorphic register-based machine model, they were originally introduced with rather different intuitions. What is called a "register" in an SST/BRTT corresponds, counterintuitively, to a "state" in an MTT, and the bottom-up recombination of registers is described instead as a top-down propagation of states. Furthermore, copyless MTTs (called "strongly single use restricted" in [EM99]) over unary trees (i.e. strings) actually do not compute all regular functions [EM99, Theorem 5.6], while copyless SSTs compute all regular string functions; this is due to a small detail (with dramatic consequences) regarding how the output is produced (an MTT has an initial state, not an initial combination of states, while the output function of an SST can combine multiple registers).

6.1. Tree streaming settings and $\mathfrak{C}$-BRTTs

We first define a generalized notion of BRTT parameterized by a category plus some auxiliary data, analogously to the $\mathfrak{C}$-SSTs for strings. To work with trees, a monoidal product will be necessary.

Definition 6.1.1. Let X be a set. A *tree streaming setting* with output X is a tuple $\mathfrak{C} = (\mathcal{C}, \otimes, \mathbf{I}, \perp\!\!\!\perp, (\!|-|\!))$ where

- $(\mathcal{C}, \otimes, \mathbf{I})$ is a *symmetric monoidal* category
- $\perp\!\!\!\perp$ is an object of $\mathcal{C}$
- $(\!|-|\!)$ is a set-theoretic map $\mathsf{Hom}_{\mathcal{C}}(\mathbf{I}, \perp\!\!\!\perp) \to X$

In the rest of this chapter, we may omit the "tree" in when discussing streaming settings.

This notion essentially differs by asking that the underlying category be equipped with a symmetric monoidal product, which is used in defining the semantics of $\mathfrak{C}$-BRTTs. The tensor product is used to fit the branching structure of trees and $\mathbf{I}$ is used for terminal nodes (so there is no need of a distinguished initial object $\top\!\!\!\top$ as in string streaming settings).

Definition 6.1.2. Let $\mathfrak{C} = (\mathcal{C}, \otimes, \mathbf{I}, \perp\!\!\!\perp, (\!|-|\!))$ be a tree streaming setting with output X. A $\mathfrak{C}$-BRTT with input (ranked) alphabet $\mathbf{\Sigma}$ and output X is a tuple (Q, R, δ, o) where

- Q is a finite set of states
- R is an object of $\mathcal{C}$
- δ is a function $\displaystyle\prod_{a \in \Sigma} \left(Q^{\mathrm{ar}(a)} \to Q \times \mathsf{Hom}_{\mathcal{C}} \left(\bigotimes_{\mathrm{ar}(a)} R \text{,} R \right) \right)$
- $o \in \mathsf{Hom}_{\mathcal{C}}(R, \perp\!\!\!\perp)$ is an output morphism

Its semantics is a set-theoretic map $\mathbf{Tree}(\mathbf{\Sigma}) \to X$ defined as follows: writing $\delta_Q(a, (t_i)_{i \in \mathrm{ar}(a)})$ for $\pi_1(\delta(a, (t_i)_{i \in \mathrm{ar}(a)}))$ and $\delta_{\mathcal{C}}(a, (t_i)_{i \in \mathrm{ar}(a)})$ for $\pi_2(\delta(a, (t_i)_{i \in \mathrm{ar}(a)}))$, define auxiliary functions $\delta_Q^* : \mathbf{Tree}(\mathbf{\Sigma}) \to Q$ and $\delta_{\mathcal{C}}^* : \mathbf{Tree}(\mathbf{\Sigma}) \to \mathsf{Hom}_{\mathcal{C}}(\mathbf{I}, R)$ by iterating δ:

$$\begin{aligned} \delta_Q^*\left(a\left((t_i)_{i \in \mathrm{ar}(a)}\right)\right) &= \delta_Q\left(a, (\delta_Q^*(t_i))_{i \in \mathrm{ar}(a)}\right) \\ \delta_{\mathcal{C}}^*\left(a\left((t_i)_{i \in \mathrm{ar}(a)}\right)\right) &= \delta_{\mathcal{C}}\left(a, (\delta_Q^*(t_i))_{i \in \mathrm{ar}(a)}\right) \circ \left(\bigotimes_{i \in \mathrm{ar}(a)} \delta_{\mathcal{C}}^*(t_i) \right) \circ \varphi_{\mathrm{ar}(a)} \end{aligned}$$

where $\varphi_{\mathrm{ar}(a)}$ is the unique isomorphism $\mathbf{I} \xrightarrow{\sim} \bigotimes_{i \in \mathrm{ar}(a)} \mathbf{I}$ generated by the associator and unitors of $(\mathcal{C}, \otimes, \mathbf{I})$. The output function $[\![\mathcal{T}]\!] : \mathbf{Tree}(\mathbf{\Sigma}) \to X$ is defined as $[\![\mathcal{T}]\!] = (\!|o \circ \delta_{\mathcal{C}}^*|\!)$.

Remark 6.1.3. Strictly speaking, we do not need the monoidal product to be symmetric for the notion to make sense, but it would require using a fixed order over the input ranked alphabet $\mathbf{\Sigma}$. Although one could choose an arbitrary total order over $\mathbf{\Sigma}$, different orders might define different classes of functions $\mathbf{Tree}(\mathbf{\Sigma})$ when the monoidal product is not symmetric. This is why we work in symmetric monoidal categories.

As with string streaming settings, it is convenient to define a notion of morphism of tree streaming settings to compare the expressiveness of classes of BRTTs.

Definition 6.1.4. Let $\mathfrak{C} = (\mathcal{C}, \mathbf{I}_{\mathcal{C}}, \otimes_{\mathcal{C}}, \perp\!\!\!\perp_{\mathcal{C}}, (\!|-|\!)_{\mathfrak{C}})$ and $\mathfrak{D} = (\mathcal{D}, \mathbf{I}_{\mathcal{D}}, \otimes_{\mathcal{C}}, \perp\!\!\!\perp_{\mathcal{D}}, (\!|-|\!)_{\mathfrak{D}})$ be tree streaming settings with output X. A morphism of tree streaming settings is given by a lax monoidal functor $F : (\mathcal{C}, \otimes_{\mathcal{C}}, \mathbf{I}_{\mathcal{C}}) \to (\mathcal{D}, \otimes_{\mathcal{D}}, \mathbf{I}_{\mathcal{D}})$ and a $\mathcal{D}$-arrow $o : F(\perp\!\!\!\perp_{\mathcal{C}}) \to \perp\!\!\!\perp_{\mathcal{D}}$ such that, for every $f \in \mathsf{Hom}_{\mathcal{C}}(\top\!\!\!\top_{\mathcal{C}}, \perp\!\!\!\perp_{\mathcal{C}})$, we have

$$(\!|o \circ F(f) \circ i|\!)_{\mathcal{D}} = (\!|f|\!)_{\mathcal{C}}$$

where $i : \mathbf{I}_{\mathcal{D}} \to F(\mathbf{I}_{\mathcal{C}})$ is obtained as part of the lax monoidal functor structure over F.

Observe that we do not require those functors to commute with the symmetry morphisms for the monoidal products, as promised in Section 4.1. This is consistent with the fact that the symmetries are not really involved in computing the image of a tree by a $\mathfrak{C}$-BRTT, according to Remark 6.1.3: it is only their mere existence that matters. The point of the above definition is, of course:

Lemma 6.1.5. *If there is a morphism of tree streaming settings $\mathfrak{C} \to \mathfrak{D}$, then $\mathfrak{D}$-BRTTs subsume $\mathfrak{C}$-BRTTs and single-state $\mathfrak{D}$-BRTTs subsume single-state $\mathfrak{C}$-BRTTs.*

Proof. Let us give the proof for single-state BRTTs; the proof for general BRTTs would be notationally heavier but not much more insightful. Suppose that we have some single-state $\mathfrak{C}$-BRTT $\mathcal{T} = (R, \delta, o)$ – where we leave the only state implicit and regard the transition function δ and output function o as elements of $\prod_{a\in\Sigma} \mathsf{Hom}_{\mathcal{C}}\left(\bigotimes_{\mathrm{ar}(a)} R, R\right)$ and $\mathsf{Hom}_{\mathcal{C}}(R, \perp\!\!\!\perp)$ respectively – and that the morphism under consideration is composed of a lax monoidal functor $F : \mathcal{C} \to \mathcal{D}$ and a $\mathcal{D}$-arrow $o' : F(\perp\!\!\!\perp_{\mathcal{C}}) \to \perp\!\!\!\perp_{\mathcal{D}}$.

Since F is lax monoidal, we have a family of natural transformations

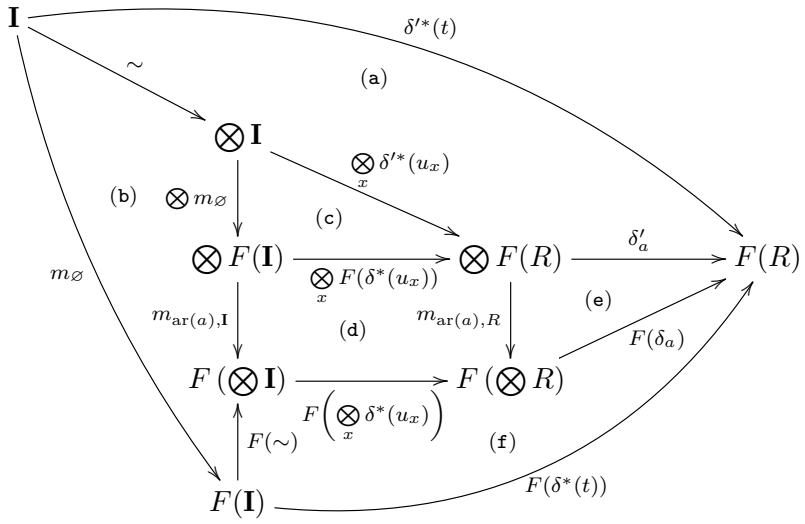
$$m_{I,A} \;:\; \bigotimes_I F(A) \;\longrightarrow\; F\left(\bigotimes_I A\right)$$

where I ranges over all finite sets[1] and A over objects of $\mathcal{C}$; this family is compatible with the associators and unitors in $\mathcal{C}$ and $\mathcal{D}$. Furthermore, $m_{\varnothing,A} : \mathbf{I} \to F(\mathbf{I})$ is the same for all $A \in \mathsf{Obj}(C)$, so we shall abbreviate it as $m_\varnothing$.

We claim that $[\![\mathcal{T}]\!] = [\![\mathcal{T}']\!]$, where $\mathcal{T}'$ is the single-state $\mathfrak{D}$-BRTT $(F(R), \delta', o' \circ F(o))$ with the same typing conventions and $\delta'_a = F(\delta_a) \circ m_{\mathrm{ar}(a),R}$. Let us prove this. To do so we first consider the iterations of the transition functions

$$\delta^* : \mathbf{Tree}(\mathbf{\Sigma}) \to \mathsf{Hom}_{\mathcal{C}}(\mathbf{I}, R) \qquad \text{and} \qquad \delta'^* : \mathbf{Tree}(\mathbf{\Sigma}) \to \mathsf{Hom}_{\mathcal{D}}(\mathbf{I}, F(R))$$

and show that $\delta'^*(t) = F(\delta(t)) \circ m_\varnothing$ by induction over $t \in \mathbf{Tree}(\mathbf{\Sigma})$. So suppose that $t = a((u_x)_{x\in\mathrm{ar}(a)})$ and that the inductive hypothesis holds (this also takes care of the base case: when $\mathrm{ar}(a) = \varnothing$, the inductive hypothesis is vacuous). In such a case, let us show that each face in the following diagram commutes (where all tensor products have arity $\mathrm{ar}(a)$):



[1] Recall from Section 2.1 that an operation $\bigotimes_{i\in I}(-)$ – here, a functor – is associated to every finite indexing set I by choosing an arbitrary total order over I.

Faces `(a)` and `(f)` commute by definition of iterated transition functions and face `(e)` corresponds to the definition of δ'. Face `(d)` commutes because of the naturality of $m_{-,-}$ and face `(b)` because of its compatibility with associators. Finally, face `(c)` corresponds to the inductive hypothesis. Therefore, the topmost and bottommost paths coincide, so we have $\delta'^*(t) = F(\delta(t)) \circ m_\varnothing$, which concludes our inductive argument. We can then conclude since we have, for every tree t,

$$\begin{array}{rcll}
[\![\mathcal{T}']\!](t) & = & (\!|o' \circ F(o) \circ (\delta'^*(t))|\!)_{\mathfrak{D}} & \text{by definition} \\
& = & (\!|o' \circ F(o) \circ F(\delta^*(t)) \circ m_\varnothing|\!)_{\mathfrak{D}} & \text{inductive argument} \\
& = & (\!|o' \circ F(o \circ \delta^*(t)) \circ m_\varnothing|\!)_{\mathfrak{D}} & \text{by functoriality} \\
& = & (\!|o \circ (\delta^*(t))|\!)_{\mathfrak{C}} & \text{since } (F, o') \text{ is part of a morphism } \mathfrak{C} \to \mathfrak{D} \\
& = & [\![\mathcal{T}]\!](t) & \text{by definition} \qquad \square
\end{array}$$

Similarly as for strings, the coproduct completion of a category induces a morphism of tree streaming settings $\mathfrak{C} \mapsto \mathfrak{C}_\oplus$. Furthermore, the expressiveness of $\mathfrak{C}$ and $\mathfrak{C}_\oplus$ remain the same under similar hypotheses as Theorem 4.3.8.

Theorem 6.1.6. *Let $\mathfrak{C}$ be a tree streaming setting whose monoidal product is affine and such that all objects of the underlying category have unitary support. Then $\mathfrak{C}$-BRTTs, $\mathfrak{C}_\oplus$-BRTTs and single-state $\mathfrak{C}_\oplus$-BRTTs are equi-expressive.*

The proof is an unsurprising adaptation of the one of Theorem 4.3.8 and of the material on state-dependent memory SSTs (Section 4.3.2). We leave it to the interested reader.

6.2. Multicategorical preliminaries

Rather than starting from the category corresponding exactly to the BRTTs from [AD17], we will first study a more convenient streaming setting based on the idea of trees with multiple holes. For this, it will be convenient to introduce the notion of *multicategory*, which is essentially a notion of category where morphisms are allowed to have multiple input objects.

This section is therefore devoted to spelling out the formal definition of the notion of multicategory that we use, and to describing how to (freely) generate affine monoidal categories from multicategories. This material is rather dry and should maybe only be skimmed over at first reading.

Definition 6.2.1. A (weak symmetric) multicategory $\mathcal{M}$ consists of

- a class of objects $\mathsf{Obj}(\mathcal{M})$
- a class of *multimorphisms* going from pairs $(I, (A_i)_{i\in I})$ of a finite index set I and a family $(A_i)_{i\in I}$ of objects to objects B. We omit the first component of the source and write $\mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right)$ for the set of these multimorphisms.
- for every object A, a distinguished identity multimorphism $\mathrm{id}_A \in \mathsf{Hom}_{\mathcal{M}}\left((A)_{*\in 1}, A\right)$.
- for every set-theoretic map $f : I \to J$, families $(A_i)_{i\in I}$, $(B_j)_{j\in J}$ and object C, a composition operation

$$\begin{array}{ccccc}
\mathsf{Hom}_{\mathcal{M}}\left((B_j)_{j\in J}, C\right) & \times & \left[\prod_{j\in J} \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in f^{-1}(j)}, B_j\right)\right] & \longrightarrow & \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, C\right) \\
\alpha & , & (\beta_j)_{j\in J} & \longmapsto & \alpha *_f \beta
\end{array}$$

- for every bijection $\sigma : I' \to I$ between finite sets, a family of actions

$$\sigma^* : \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right) \to \mathsf{Hom}_{\mathcal{M}}\left((A_{\sigma(i')})_{i'\in I'}, B\right)$$

  correspond to reindexing along symmetries.

Furthermore, the above data is required to obey the following laws.

- The identity morphism be a neutral for composition: for any $\alpha \in \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right)$,
$$\mathrm{id}_B *_! (\alpha)_{*\in 1} \quad = \quad \alpha \quad = \quad \alpha *_{\mathrm{id}_I} (\mathrm{id}_{A_i})_{i\in I}$$
- Composition is associative: for any finite sets I, J and K, functions $f : K \to J$ and $g : J \to I$, families of objects $(A_k)_{k\in K}$, $(B_j)_{j\in J}$, $(C_i)_{i\in I}$, D and families of morphisms
$$\begin{array}{l}\alpha \in \prod_{j\in J} \mathsf{Hom}_{\mathcal{M}}\left((A_k)_{k\in f^{-1}(j)}, B_j\right)\\ \beta \in \prod_{i\in I} \mathsf{Hom}_{\mathcal{M}}\left((B_j)_{j\in g^{-1}(i)}, C_i\right)\\ \gamma \in \mathsf{Hom}_{\mathcal{M}}\left((C_i)_{i\in I}, D\right)\end{array}$$
the following equation holds
$$\gamma *_g (\beta *_f \alpha) = \left(\gamma *_{g_{|f^{-1}(j)}} \beta\right)_j *_f \alpha$$
- Permutations act functorially: for any $(A_i)_{i\in I}$, B and bijections $\sigma : I' \to I$ and $\sigma' : I'' \to I'$, the following commute
$$\begin{array}{ccc}\mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right) & \longrightarrow & \mathsf{Hom}_{\mathcal{M}}\left((A_{\sigma(i')})_{i'\in I'}, B\right)\\ & \searrow & \downarrow\\ & & \mathsf{Hom}_{\mathcal{M}}\left((A_{\sigma(\sigma'(i''))})_{i''\in I''}, B\right)\end{array}$$
and $\mathrm{id}_I^* : \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right) \to \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right)$ is the identity.
- Composition is compatible with permutations: for every commuting square
$$\begin{array}{ccc}J' & \xrightarrow{g} & I'\\ \downarrow & & \downarrow{\scriptstyle \sigma}\\ J & \xrightarrow[f]{} & I\end{array}$$
in FinSet such that the vertical arrows be bijections, note in particular that for every $i \in I$, there is a bijection $\sigma_i : g^{-1}(I') \to f^{-1}(I)$; we thus require that
$$\alpha *_f \left(\beta_{j\in f^{-1}(i)}\right)_{i\in I} \quad = \quad \sigma^*(\alpha) *_g \left(\sigma_i^* \left(\beta_{j\in f^{-1}(i)}\right)\right)_{i\in I}$$
for every suitable α and β_js.

Every symmetric monoidal category $\mathcal{C}$ can be mapped to a multicategory $\mathcal{C}_{\mathrm{mcat}}$ by taking
$$\mathsf{Hom}_{\mathcal{C}_{\mathrm{mcat}}}\left((A_i)_{i\in I}, B\right) \quad = \quad \mathsf{Hom}_{\mathcal{C}}\left(\bigotimes_{i\in I} A_i \,,\, B\right)$$
We may make this map functorial, provided we equip the class of multicategories and the class of symmetric monoidal categories with categorical structures.

Definition 6.2.2. Given two weak multicategories $\mathcal{M}$ and $\mathcal{N}$, a functor $F : \mathcal{M} \to \mathcal{N}$ consists of maps of objects $F : \mathsf{Obj}(\mathcal{M}) \to \mathsf{Obj}(\mathcal{N})$ and of multimorphisms
$$F \;:\; \mathsf{Hom}_{\mathcal{M}}\left((A_i)_{i\in I}, B\right) \;\longrightarrow\; \mathsf{Hom}_{\mathcal{N}}\left((F(A_i))_{i\in I}, F(B)\right)$$
such that $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$, $F\left(\alpha *_f (\beta_j)_{j\in f^{-1}(i)}\right) = F(\alpha) *_f (F(\beta_j))_{j\in f^{-1}(i)}$ and $F(\sigma * (\alpha)) = \sigma^*(F(\alpha))$ for all suitable objects, index sets, set-theoretic functions and morphisms.

Definition 6.2.2 gives a class of arrows for a large category MCat of multicategories. Calling Aff the category whose objects are symmetric affine monoidal categories and morphisms are strong monoidal functors, the map $\mathcal{C} \to \mathcal{C}_{\mathrm{mcat}}$ extends to a functor $\mathsf{Aff} \to \mathsf{MCat}$. We are now

interested in the inverse process of generating freely a symmetric affine monoidal category out of a weak multicategory.

Definition 6.2.3. Let $\mathcal{M}$ be a weak multicategory. The *free affine symmetric monoidal category* generated by $\mathcal{M}$ is the category $\mathcal{M}_{\mathrm{aff}}$ such that

- objects are pairs $(I, (A_i)_{i\in I})$ of a finite set I and a family of objects of $\mathcal{M}$; write $\bigotimes_{i\in I} A_i$ for such objects.
- morphisms $\bigotimes_{i\in I} A_i \to \bigotimes_{j\in J} B_j$ are pairs $(f, (\alpha_j)_{j\in J})$ where f is a partial function $I \rightharpoonup J$ and α_j is a $\mathcal{M}$ multimorphism $(A_i)_{i\in f^{-1}(J)} \to B_j$.
- identities $\bigotimes_{i\in I} A_i \to \bigotimes_{i\in I} A_i$ are the pairs $(\mathrm{id}_I, (\mathrm{id}_{A_i})_{i\in I})$.
- the composition of

$$(f, (\alpha_j)_{j\in J}) : \bigotimes_{i\in I} A_i \to \bigotimes_{j\in J} B_j \text{ and } (g, (\beta_k)_{k\in K}) : \bigotimes_{j\in J} B_j \to \bigotimes_{k\in K} C_k$$

is $\left(g \circ f, \left(\beta_k * (\alpha_j)_{j\in g^{-1}(k)}\right)_{k\in K}\right)$.

For any bijection $\sigma : I' \to I$, we have canonical isomorphisms $\bigotimes_{i\in I} A_i \to \bigotimes_{i\in I} A_{\sigma(i)}$. We take the binary tensor product to be

$$\left(\bigotimes_{i\in I} A_i\right) \otimes \left(\bigotimes_{j\in J} B_j\right) = \bigotimes_{x\in I+J} \begin{cases} A_i & \text{if } x = \mathsf{in}_1(i) \\ B_j & \text{if } x = \mathsf{in}_2(j) \end{cases}$$

and the unit to be the terminal object, which is the nullary family $\bigotimes_{\varnothing}$. The associator and symmetries are induced by the isomorphisms $(I + J) + K \cong I + (J + K)$ and $I + J \cong J + I$ respectively, and the units by $I + \varnothing \cong I \cong \varnothing + I$. The axioms of weak symmetric multicategories then imply that this indeed endows $\mathcal{M}_{\mathrm{aff}}$ with a symmetric affine monoidal structure. We skip checking the details.

6.3. The combinatorial multicategory $\mathcal{TR}^{\mathrm{m}}$

We are now ready to give a smooth definition of a category of register transitions for trees, generalizing Proposition 4.2.12. As announced, we find it more convenient to first give a multicategory $\mathcal{TR}^{\mathrm{m}}$ and then move to monoidal categories by taking $\mathcal{TR} = (\mathcal{TR}^{\mathrm{m}})_{\mathrm{aff}}$. We then discuss the restriction consisting of limiting the number of holes in the tree expressions stored in register to at most one and show that it is not limiting.

Notations Recall that we regard ranked alphabets $\mathbf{R}$ as pairs (R, ar) where R is a finite set of letters and $(\mathrm{ar}(a))_{a\in R}$ is a family $R \to \mathsf{FinSet}$ of arities. Given two ranked alphabets $\mathbf{R}$ and $\mathbf{S}$, we suggestively write $\mathbf{R} \otimes \mathbf{S}$ for the ranked alphabet $(R + S, [\mathrm{ar}, \mathrm{ar}])$. Given a finite set U, call $\mathcal{O}(U)$ the ranked alphabet $(U, (\varnothing)_{u\in U})$ consisting of $|U|$-many terminal letters and $\mathcal{I}(U)$ for the ranked alphabet consisting of a single letter of arity U. Given a ranked alphabet $\mathbf{\Sigma} = (\Sigma, \mathrm{ar})$ and a subset $X \subseteq \Sigma$, we write $\mathbf{\Sigma} \restriction X$ for the restriction $(X, \mathrm{ar} \restriction X)$.

Before giving the definition of $\mathcal{TR}^{\mathrm{m}}$, we first need to make formal a notion of trees with linearly many occurrences of certain constructors.

Definition 6.3.1. Let $\mathbf{R}$ be a ranked alphabet. We define the set $\mathbf{LTree}_{\mathbf{\Gamma}}(\mathbf{R})$ of $\mathbf{R}$-linear trees as the subset of $\mathbf{Tree}(\mathbf{\Gamma} \otimes \mathbf{R})$ comprising the trees where all constructors of $\mathbf{R}$ appear exactly once.

Definition 6.3.2. Define $\mathcal{TR}^{\mathrm{m}}(\mathbf{\Gamma})$ (abbreviated $\mathcal{TR}^{\mathrm{m}}$ in the sequel) as the multicategory

- whose class of objects $\mathsf{Obj}(\mathcal{TR}^{\mathrm{m}})$ is FinSet.

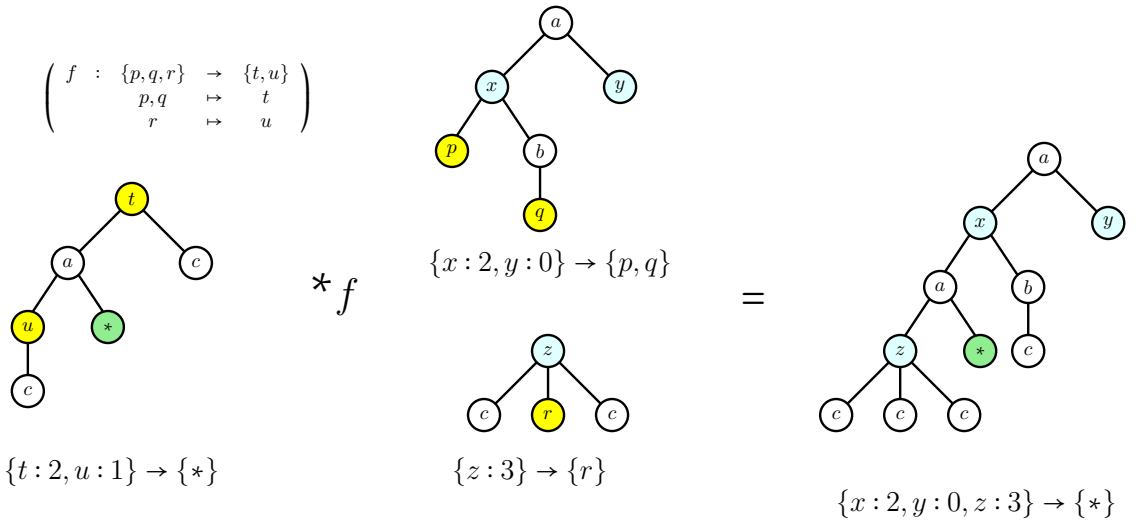


FIGURE 6.3.1. Composition of some multimorphisms of $\mathcal{TR}^{\mathrm{m}}$.

- whose class of multimorphisms from $(A_i)_{i\in I}$ to B is the set of linear trees over the joint alphabet $(I,A)\otimes\mathcal{O}(B)$ (recall that (I,A) can formally be regarded as a ranked alphabet: its set of letters is I and the arity of $i\in I$ is $A(i)=A_i$).

$$\mathsf{Hom}_{\mathcal{TR}^{\mathrm{m}}}((A_i)_{i\in I},B) \quad = \quad \mathbf{LTree}_{\mathbf{\Gamma}}((I,A)\otimes\mathcal{O}(B))$$

- whose composition operations are given by substitution: given a map $f:I\to J$ and multimorphisms

$$t\in\mathbf{LTree}_{\mathbf{\Gamma}}((J,B)\otimes\mathcal{O}(C)) \qquad \text{and} \qquad u\in\prod_{j\in J}\mathbf{LTree}_{\mathbf{\Gamma}}((f^{-1}(j),(A_i)_i)\otimes\mathcal{O}(B_j))$$

the composite $t*_f u$ is defined by recursion over t:
– if $t=a((t'_k)_k)$ for some $a=\mathsf{in}_1(b)$ with $b\in\Gamma$ or $a=\mathsf{in}_2(\mathsf{in}_2(c))$ for $c\in C$, then

$$t*_f u = a((t'_k *_f u)_k)$$

– otherwise $t=\mathsf{in}_2(\mathsf{in}_1(j))((t'_b)_{b\in B_j})$ with $j\in J$ and t'_b in some $\mathbf{LTree}_{\mathbf{\Gamma}}((J_b,B_b)\otimes\mathcal{O}(C_j))$ for $\bigcup_{b\in B_j}J_b=J\setminus\{j\}$, $B_b=B\restriction J_b$ and $\bigcup_{b\in B_j}C_b=C$. In such a case, we set

$$t*_f u = u_j[(t'_b *_{\mathrm{id}_{J_b}}(u_{j'})_{j'\in J_b})/b]_{b\in B_j}$$

where $[-/-]_{-\in-}$ denotes the more usual substitution of leaves by subtrees (recall that every $b\in B$, and a fortiori B_j has arity $\varnothing$).

While the definition of composition of multimorphisms in $\mathcal{TR}^{\mathrm{m}}$ looks daunting, we claim it is rather natural. Figure 6.3.1 depicts the composition $\alpha *_f (\beta_x)_{x\in\{t,u\}}$ with

$$\alpha=t(a(u(c()),*()),c()) \qquad \beta_t=a(x(p(),b(q())),y()) \quad \text{and} \quad \beta_u=z(c(),r(),c())$$

We now set $\mathcal{TR}=(\mathcal{TR}^{\mathrm{m}})_{\mathrm{aff}}$; while objects of $(\mathcal{TR}^{\mathrm{m}})_{\mathrm{aff}}$ are supposed to be families of finite sets $(A_i)_{i\in I}$, in the sequel, we sometimes identify them with the ranked alphabets (I,A) in $\mathcal{TR}$ for notational convenience. As such, the notation $\mathbf{R}\otimes\mathbf{S}$ corresponds to the expected tensorial product in $\mathcal{TR}$.

We are now ready to define our first tree streaming setting.

Definition 6.3.3. $\mathfrak{TM}$ is the tree streaming setting $(\mathcal{TR}, \otimes, \top, \mathcal{I}(\varnothing), (\!|-|\!))$, where $(\!|-|\!)$ is the canonical isomorphism $\mathsf{Hom}_{\mathcal{TR}}(\top, \mathcal{I}(\varnothing)) \cong \mathsf{Hom}_{\mathcal{TR}^{\mathrm{m}}}((\,)_\varnothing, \varnothing) \cong \mathbf{LTree}(\varnothing) \cong \mathbf{Tree}(\mathbf{\Gamma})$ (where $(\,)_\varnothing$ is the empty family).

Call $\mathcal{TR}^{\mathrm{m},\leq 1}$ the full submulticategory of $\mathcal{TR}^{\mathrm{m}}$ whose objects are empty or singleton sets, and $\mathcal{TR}^{\leq 1} \cong \mathcal{TR}^{\mathrm{m},\leq 1}_{\mathrm{aff}}$ to be the corresponding full subcategory of $\mathcal{TR}$. The monoidal structure of $\mathcal{TR}$ restricts to $\mathcal{TR}^{\leq 1}$ without any difficulty, and that $\mathcal{I}(\varnothing)$ is an object of $\mathcal{TR}^{\leq 1}$. This means that $\mathfrak{TM}$ has a restriction to a streaming setting $\mathfrak{TM}^{\leq 1}$.

While $\mathfrak{TM}^{\leq 1}$ turns out to be more elementary and a good building block toward the definition of usual BRTTs, it is easier to show the monoidal closure of $\mathfrak{TM}_{\oplus\&}$ than $\mathfrak{TM}^{\leq 1}$. Thankfully, it turns out that the expressiveness of BRTTs over $\mathfrak{TM}$ and $\mathfrak{TM}^{\leq 1}$ is the same.

For one direction, there is a morphism $\mathfrak{TM}^{\leq 1} \to \mathfrak{TM}$ corresponding to the embedding $\mathcal{TR}^{\leq 1} \to \mathcal{TR}$. For the other direction, we exploit Theorem 6.1.6. The proof involves some combinatorics, but nothing surprising as it amounts to the classical decomposition of multi-hole trees into families of single-hole trees as found in e.g. [AD17, §3.5].

Lemma 6.3.4. *There is a morphism of streaming settings $\mathfrak{TM} \to \mathfrak{TM}^{\leq 1}_{\oplus}$.*

Proof sketch. We focus on giving enough ingredients to define the underlying (strong) monoidal functor $F : \mathcal{TR} \to \mathcal{TR}^{\leq 1}_{\oplus}$, which is going to preserve $\perp\!\!\!\perp$ (i.e., we will have $F(\mathcal{I}(\varnothing)) \cong \iota_{\oplus}(\mathcal{I}(\varnothing))$).

Rather than giving a direct explicit construction of F (which is rather tedious over morphisms), we obtain it as a composition of two strong monoidal functors: the strong monoidal embedding $\iota_{\oplus} : \mathcal{TR} \to \mathcal{TR}_{\oplus}$ and a functor $R : \mathcal{TR}_{\oplus} \to \mathcal{TR}^{\leq 1}_{\oplus}$ right adjoint to the inclusion $I : \mathcal{TR}^{\leq 1}_{\oplus} \to \mathcal{TR}_{\oplus}$.

$$\mathcal{TR} \xrightarrow{\iota_{\oplus}} \mathcal{TR}_{\oplus} \underset{I}{\overset{R}{\rightleftarrows}} \mathcal{TR}^{\leq 1}_{\oplus} \qquad (\top)$$

I is strong symmetric monoidal. Therefore, by [Mel09, Proposition 14, Section 5.17], once we construct R right adjoint to I, it comes equipped with a canonical lax monoidal structure. Furthermore, since we want $I \dashv R$, we can use the implicit characterization of adjoints given in [Mac98, item (iv), Theorem 2, Section IV.1]: to define R, it suffices to give the value of $R(A)$ for every object $A \in \mathsf{Obj}(\mathcal{TR}_{\oplus})$ and counit maps $\epsilon_A : I(R(A)) \to A$ such that, for every object $B \in \mathsf{Obj}(\mathcal{TR}^{\leq 1}_{\oplus})$ and map $h \in \mathsf{Hom}_{\mathcal{TR}_{\oplus}}(I(B), A)$, there is a unique $\widetilde{h} \in \mathsf{Hom}_{\mathcal{TR}^{\leq 1}_{\oplus}}(R(A), B)$ such that the following diagram commutes

$$\begin{array}{ccc} I(B) & \overset{I(\widetilde{h})}{\dashrightarrow} & I(R(A)) \\ & \searrow^{h} & \downarrow{\scriptstyle \epsilon_A} \\ & & A \end{array}$$

So we only need to define $R(A)$ and ϵ_A to obtain our functor R; once those are defined, we leave checking that the universal property holds to the reader. We first focus on the case where $A = \iota_{\oplus}(\mathcal{I}(U))$ for some finite set U. Recall that a single-letter alphabet $\mathcal{I}(U)$, when seen as an object of $\mathcal{TR}$, should be should be intuitively regarded as a register containing a tree with U-many holes. If $|U| \leq 1$, we may simply take $R(\iota_{\oplus}(\mathcal{I}(U))) = \iota_{\oplus}(\mathcal{I}(U))$. Otherwise, $|U| \geq 2$ and $\mathcal{I}(U)$ is not an object $\mathcal{TR}^{\leq 1}$; in that case, we use the following

recursive definition

$$R(\iota_\oplus(\mathcal{I}(U))) \;=\; \mathcal{I}(1) \otimes \bigoplus_{b\in\Gamma} \bigoplus_{\substack{f:U\to \mathrm{ar}(b)\\ \text{nonconstant}}} \bigotimes_{x\in\mathrm{ar}(b)} R(\iota_\oplus(\mathcal{I}(f^{-1}(x))))$$

Note that this definition is well-founded because the function f in the second sum is taken to be non constant, so that $|f^{-1}(x)| < |U|$ for every x. While this suffices as a definition of $R(\iota_\oplus(\mathcal{I}(U)))$, this might be a bit opaque without having the definition of $\epsilon_{R(\iota_\oplus(\mathcal{I}(U)))}$. Before giving that, let us attempt to give an intuitive rationale behind this definition: there is an isomorphism[2]

$$\mathsf{Hom}_{\mathcal{TR}^{\leq 1}_\oplus}(\top, R(\iota_\oplus(\mathcal{I}(U)))) \cong \mathbf{LTree}_{\mathbf{\Gamma}}(U)$$

which can be nicely pictured, provided we actually compute recursively $R(\iota_\oplus(\mathcal{I}(U)))$ and spell out a normal form

$$R(\iota_\oplus(\mathcal{I}(U))) \;\cong\; \bigoplus_{t\in\mathbf{PT}(U)} \bigotimes_{n\in N(t)} A_n$$

with all $A_n = \mathcal{I}(\varnothing)$ or $A_n \cong \mathcal{I}(1)$. It is always possible to build a suitable set $\mathbf{PT}(U)$ simply because all objects of $\mathcal{TR}^{\leq 1}_\oplus$ have this shape, but an intuitive definition of what one might call a set of *partitioning trees over* U is also possible for $\mathbf{PT}$, and $N(t)$ would then correspond to the nodes of the trees. We skip defining this notion formally, but note that the announced bijection would then match trees with U-many holes with pairs $(t, (u_i)_{i\in N(t)})$ of a partitioning tree t and a family of trees with at most one-hole $(u_i)_{i\in N(t)}$. This bijective correspondence is pictured in Figure 6.3.2.

Now, we define $\epsilon_{R(\iota_\oplus(\mathcal{I}(U)))} \in \mathsf{Hom}_{\mathcal{TR}^{\leq 1}_\oplus}(I(R(\iota_\oplus(\mathcal{I}(U)))), \iota_\oplus(\mathcal{I}(U)))$, by induction over the size of U. If $|U| \leq 1$, we take $\epsilon_{R(\iota_\oplus(\mathcal{I}(U)))} : \iota_\oplus(\mathcal{I}(U)) \to \iota_\oplus(\mathcal{I}(U))$ to be the identity. Otherwise, we need to define a map

$$g_U \;:\; I\left(\bigoplus_{b\in\Gamma} \bigoplus_{\substack{f:U\to \mathrm{ar}(b)\\ \text{nonconstant}}} \bigotimes_{x\in\mathrm{ar}(b)} R(\iota_\oplus(\mathcal{I}(f^{-1}(x))))\right) \longrightarrow \iota_\oplus(\mathcal{I}(U))$$

or, equivalently, a family of $\mathcal{TR}_\oplus$-maps indexed by $b \in \Gamma$ and $f : U \to \mathrm{ar}(b)$ non-constant

$$g_{b,f} \;:\; \bigotimes_{x\in\mathrm{ar}(b)} I(R(\iota_\oplus(\mathcal{I}(f^{-1}(x))))) \longrightarrow \iota_\oplus(\mathcal{I}(U))$$

By the induction hypothesis, we have a family of $\mathcal{TR}_\oplus$-maps $(g_x)_{x\in\mathrm{ar}(b)}$

$$g_x \;:\; I(R(\iota_\oplus(\mathcal{I}(f^{-1}(x))))) \longrightarrow \iota_\oplus(\mathcal{I}(f^{-1}(x)))$$

We define $g_{b,f}$ as the composite

$$\bigotimes_{x\in\mathrm{ar}(b)} I(R(\iota_\oplus(\mathcal{I}(f^{-1}(x))))) \xrightarrow{\otimes_x g_x} \bigotimes_{x\in\mathrm{ar}(b)} \iota_\oplus(\mathcal{I}(f^{-1}(x))) \xrightarrow{\bar{b}} \iota_\oplus(\mathcal{I}(U))$$

where $\bar{b}$ is obtained from the map of $\mathcal{TR}^{\mathrm{m}}$ which intuitively takes a family $(t_x)_{x\in\mathrm{ar}(b)}$ of trees into a single tree $b((t_x)_{x\in\mathrm{ar}(b)})$ (officially, the tree $b((*)_{x\in\mathrm{ar}(b)}) \in \mathbf{LTree}_{\mathbf{\Gamma}}(\mathcal{O}(U))$.

Now, R is defined on objects of the shape $\iota_\oplus(\mathcal{I}(U))$, as well as ϵ, so we need to extend this to the whole category $\mathcal{TR}_\oplus$. Recall that every object A of $\mathcal{TR}_\oplus$ can be written as

[2] Which we may later on define formally as a composite

$$\mathsf{Hom}_{\mathcal{TR}^{\leq 1}_\oplus}(\top, R(\iota_\oplus(\mathcal{I}(U)))) \xrightarrow{I} \mathsf{Hom}_{\mathcal{TR}_\oplus}(\top, R(\iota_\oplus(\mathcal{I}(U)))) \to \mathsf{Hom}_{\mathcal{TR}_\oplus}(\top, \mathcal{I}(U)) \xrightarrow{\sim} \mathbf{LTree}_{\mathbf{\Gamma}}(U)$$

where the mediating arrow is the post-composition by $\epsilon_{\iota_\oplus(\mathcal{I}(U))}$.

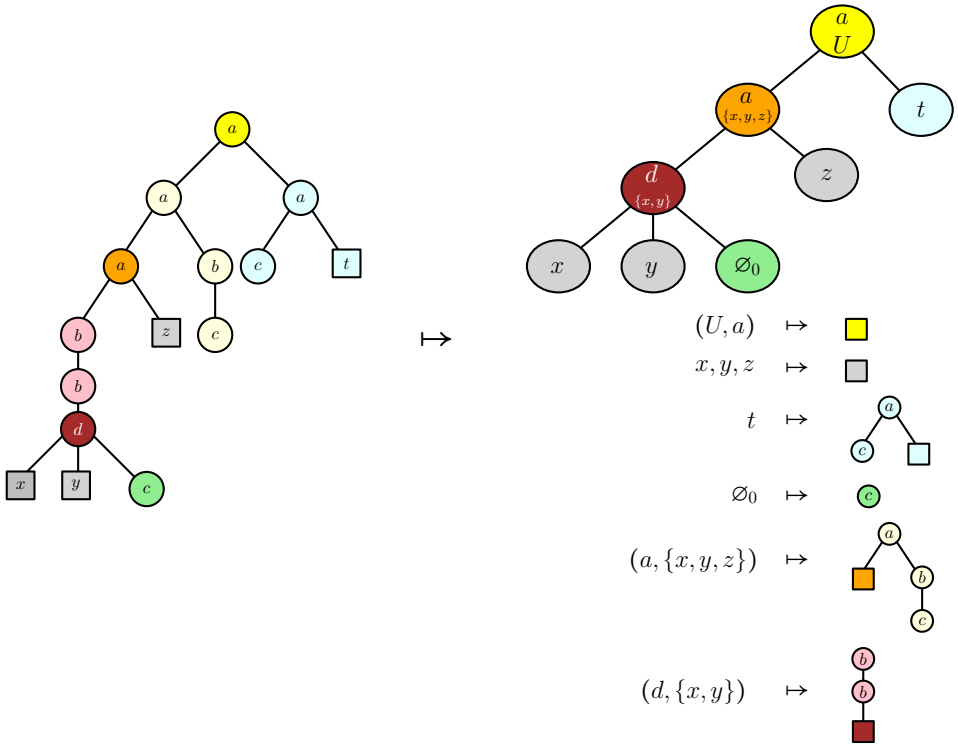


FIGURE 6.3.2. Decomposition of a multi-hole tree $\mathbf{LTree}_{\mathbf{\Sigma}}(\{x, y, z, t\})$ (where $\mathbf{\Sigma} = \{a : 2, b : 1, c : 0, d : 3\}$) as a tuple consisting of a partitioning tree and trees with at most one hole.

$A = \bigoplus_{v \in V} \bigotimes_{j \in J_u} \iota_{\oplus}(\mathcal{I}(U_j))$. In the end, the functor R is expected to be strong monoidal, and we may force it to preserve coproducts, so we set

$$R(A) \;=\; \bigoplus_{v \in V} \bigotimes_{j \in J_u} R(\iota_{\oplus}(\mathcal{I}(U_j))) \qquad\qquad \epsilon_A \;=\; \bigoplus_{v \in V} \bigotimes_{j \in J_u} \epsilon_{\iota_{\oplus}(\mathcal{I}(U_j))}$$

□

6.4. $\mathfrak{TM}_{\&}$-BRTTs coincide with regular functions, via coherence spaces

We define a streaming setting $\mathfrak{TR}^{\asymp}$ and its restriction $\mathfrak{TR}^{\asymp,\leq 1}$ (with respective underlying categories $\mathcal{TR}^{\asymp}$, $\mathcal{TR}^{\asymp,\leq 1}$) so that $\mathfrak{TR}^{\asymp,\leq 1}$-BRTTs coincide with Alur and D'Antoni's notion of single-use-restricted BRTT [AD17], which they showed to characterize regular tree functions. We then show that there are morphisms $\mathfrak{TM}_{\&} \to \mathfrak{TR}^{\asymp} \to \mathfrak{TM}_{\&}$ of streaming settings and thus establish that $\mathfrak{TM}_{\&}$-BRTTs capture exactly regular tree functions.

Much like $\mathcal{TR}$, the category $\mathcal{TR}^{\asymp}$ is obtained by applying a generic construction to $\mathcal{TR}^{\mathrm{m}}$, taking weak symmetric multicategories to symmetric affine monoidal categories. In particular, objects of $\mathcal{TR}^{\asymp}$ will consist of formal tensor products of objects of $\mathcal{TR}^{\mathrm{m}}$. The main difference is that morphisms of $\mathcal{TR}^{\asymp}$ will induce a dependency *relation* $D \subseteq I \times J$ over indexing sets, rather than a partial function $J \rightharpoonup I$. This corresponds to a relaxation of the copylessness condition. However, objects of $\mathcal{TR}^{\asymp}$ will also be equipped with a *conflict relation* $\asymp$ over their indexed sets, and D will be required to satisfy a linearity constraint. Calling $\frown$ the dual *coherence relation* such that $x \frown y$ is equivalent to $x = y \vee \neg(x \asymp y)$, if

we have $(i,j) \in D$ and $(i',j') \in D$, the linearity constraint enforces

$$j \subset_J j' \;\Rightarrow\; i \subset_I i' \qquad \text{and} \qquad i \asymp_I i' \;\Rightarrow\; j \asymp_J j'$$

This corresponds to the *single use restriction* imposed on BRTTs [AD17, §2.1], whose introduction was motivated in Section 2.6.2.

Example 6.4.1. The BRTT that we gave in for the "conditional swap" function is single-use-restricted according to the above by taking its two registers (i.e. objects of $\mathcal{TR}^{\mathrm{m}}$) to be in conflict.

But as our choice of notation and vocabulary suggests, this is also related to the category of (finite) *coherence spaces*, the first denotational model of linear logic [Gir87] (predated by a similar semantics for system F [GTL89, Appendix A]). As far as we know, this observation is new (the conflict relation is denoted by η in [AD17], while $\asymp$ comes from the linear logic literature). The coherence semantics of the linear λ-calculus has been used in particular in [GLS20] to analyze a top-down tree transducer model containing linear λ-terms. Unlike them, we do not use coherence spaces *as a semantics* here; what happens here is much closer the use of a coherence/conflict relation to handle additive connectives – we will indeed show a connection with the &-completion – in proof nets, see [Gir96, Appendix A.1] and [HH16].

Definition 6.4.2 (see e.g. [Gir95, §2.2.3])**.** A coherence space I is a pair $(\|I\|, \subset_I)$ of a set $\|I\|$, called the *web*, and a binary reflexive symmetric relation $\subset_I$ over $\|I\|$ called the *coherence relation*. As usual, given a coherence relation $\subset$, we write $\asymp$ for the dual defined by $i \asymp i' \Leftrightarrow (i = i' \vee \neg(i \subset i'))$. Finite coherence spaces are those coherence spaces whose webs are finite. A *linear map* of coherence spaces $f : I \to J$ is a relation $f \subseteq \|I\| \times \|J\|$ such that, whenever $(i,j) \in f$ and $(i',j') \in f$, we have

$$i \subset_I i' \;\Rightarrow\; j \subset_J j' \qquad \text{and} \qquad j \asymp_J j' \;\Rightarrow\; i \asymp_I i'$$

Note that these are the *converse implications* of those stated above for BRTTs.

The diagonal $\{(i,i) \mid i \in \|I\|\}$ is a linear map $I \to I$ and the relational composition of two linear maps $I \to J \to K$ is again a linear map, so that we have a category FinCoh whose objects are coherence spaces and morphisms are linear maps.

FinCoh, equipped with the tensorial product

$$(\|I\|, \subset_I) \otimes (\|J\|, \subset_J) = (\|I\| \times \|J\|, \subset_I \times \subset_J)$$

and dualizing object $(1, 1 \times 1)$, is a well-studied $*$-autonomous category with cartesian products and coproducts. The latter may be defined pointwise as

$$(\|I\|, \subset_I) \oplus (\|J\|, \subset_J) = (\|I\| + \|J\|,\ \subset_{I \oplus J})$$

where $\subset_{I\oplus J}$ is the *smallest* relation such that

$$\mathsf{in}_1(i) \subset_{I\oplus J} \mathsf{in}_1(i') \text{ when } i \subset_I i' \qquad \text{and} \qquad \mathsf{in}_2(j) \subset_{I\oplus J} \mathsf{in}_2(j') \text{ when } j \subset_J j'$$

Dualizing an object corresponds to moving from $\subset$ to $\asymp$, i.e. $(\|I\|, \subset_I)^\perp = (\|I\|, \asymp_I)$, and the product is $I \,\&\, J = (I^\perp \oplus J^\perp)^\perp$.

With this in mind, we can describe how to turn a multicategory into an affine monoidal category where monoidal products may be indexed by coherence spaces. The construction has a vague family resemblance with the *coherence completion of categories* introduced by Hu and Joyal [HJ99], but appears to have quite different properties.

Definition 6.4.3. Let $\mathcal{M}$ be a weak symmetric multicategory. We define $\mathcal{M}_{\mathrm{coh}}$ to be the category

- whose objects are pairs $(X, (R_x)_{x \in \|X\|})$ where X is a finite coherence space and $(R_x)_{x\in\|X\|}$ a family of objects of $\mathcal{M}$. We suggestively write them $\bigodot_{x\in X} R_x$.

- whose morphisms
$$(f, (\alpha_y)_{y \in \|Y\|}) \in \mathsf{Hom}_{\mathcal{M}_{\mathrm{coh}}} \left(\bigodot_{x \in X} R_x, \bigodot_{y \in Y} S_y \right)$$
are pairs consisting of a linear map $f \in \mathsf{Hom}_{\mathsf{FinCoh}}(Y, X)$ and a family of multimorphisms $\alpha_y \in \mathsf{Hom}_{\mathcal{M}}\left((R_x)_{x \in f(y)}, S_y\right)$.
- whose identities are pairs $(\mathrm{id}_X, (\mathrm{id}_{R_x})_{x \in \|X\|})$.
- where the composition of
$$(f, (\alpha_y)_{y \in \|Y\|}) \in \mathsf{Hom} \left(\bigodot_{x \in X} R_x, \bigodot_{y \in Y} S_y \right) \text{ and } (g, (\beta_z)_{z \in \|Z\|}) \in \mathsf{Hom} \left(\bigodot_{y \in Y} S_y, \bigodot_{z \in Z} T_z \right)$$
is $(f \circ g, (\beta_z * (\alpha_y)_{y \in g(z)})_{z \in Z})$.

Definition 6.4.4. We set $\mathcal{TR}^{\succ} = (\mathcal{TR}^{\mathrm{m}})_{\mathrm{coh}}$ and $\mathcal{TR}^{\succ, \leq 1}$ to be its full subcategory consisting of objects $\bigodot_{x \in X} A_x$ where each A_x is either empty or a singleton (so that $\mathcal{TR}^{\succ, \leq 1}$ is isomorphic to $(\mathcal{TR}^{\mathrm{m}, \leq 1})_{\mathrm{coh}}$).

Proposition 6.4.5. *BRTTs over the restricted tree streaming setting $\mathfrak{TR}^{\succ, \leq 1}$ compute exactly the regular tree functions.*

Proof. By virtue of being equivalent to Alur and D'Antoni's notion of single-use-restricted BRTT [AD17]. We point the reader to Section 2.6.3 for a self-contained definition of those not involving categories, and leave it as an exercise to formally match those two descriptions. Although [AD17] and our Section 2.6.3 only consider BRTTs over *binary* trees, the proof of equivalence between the latter and regular tree functions goes through *macro tree transducers* (with regular look-ahead and single use restriction) which are known to compute regular functions for trees over arbitrary ranked alphabets [EM99], so everything can be made to work out with arbitrary arities in the end. (See also Remark 6.0.1.) □

This being done, the remainder of this section does not depend on $\mathcal{TR}^{\mathrm{m}}$; the arguments apply to any weak symmetric multicategory $\mathcal{M}$ and designated object $\perp\!\!\!\perp \in \mathsf{Obj}(\mathcal{M})$.

Accordingly, fix such an $\mathcal{M}$ and a $\perp\!\!\!\perp$ for the remainder of this section. Fix also a set O and a map $(\!|-|\!) : \mathsf{Hom}_{\mathcal{M}}((\cdot)_{\varnothing}, \perp\!\!\!\perp) \to O$.

Proposition 6.4.6. *$\mathcal{M}_{\mathrm{coh}}$ has a terminal object, given by the unique family over the empty coherence space, and can be equipped with a symmetric monoidal affine structure $(\otimes, \top)$ where*
$$\left(\bigodot_{i \in I} A_i \right) \otimes \left(\bigodot_{j \in J} B_j \right) = \bigodot_{x \in I \& J} \begin{cases} A_i & \text{if } x = \mathsf{in}_1(i) \\ B_j & \text{if } x = \mathsf{in}_2(j) \end{cases}$$
and $I \,\&\, J$ designates the cartesian product in FinCoh.

Proof. Left to the reader. Strictly speaking, later developments will depend on the precise structure itself and not merely on its existence, but there is a single sensible choice of bifunctorial action and structural morphisms making the above a monoidal product. □

Remark 6.4.7. To start making sense of the use of the cartesian product of FinCoh, there is a useful analogy with $\mathcal{M}_{\mathrm{aff}}$ here. There is a projection functor $\mathcal{M}_{\mathrm{aff}} \to \mathsf{PartFinSet}$ where PartFinSet is the category of finite sets and partial functions. The tensorial product of $\mathcal{M}_{\mathrm{aff}}$ required a coproduct at the level of indices. Here, we have a projection functor $\mathcal{M}_{\mathrm{coh}} \to \mathsf{FinCoh}^{\mathrm{op}}$, and we again use a coproduct at the level of indices (which becomes a product due to the contravariance).

We call $\mathfrak{M}_{\mathrm{aff}}$ the tree streaming setting based on $\mathcal{M}_{\mathrm{aff}}$, $\perp\!\!\!\perp$ and $(\!|-|\!)$, and $\mathfrak{M}_{\mathrm{coh}}$ the corresponding tree streaming setting based on $\mathcal{M}_{\mathrm{coh}}$.

Proposition 6.4.8. *There is a full and faithful strong monoidal functor $\mathcal{M}_{\mathrm{aff}} \to \mathcal{M}_{\mathrm{coh}}$ extending to a morphism of streaming setting $\mathfrak{M}_{\mathrm{aff}} \to \mathfrak{M}_{\mathrm{coh}}$.*

Proof. Call F this functor, and, for any set I, write Δ for the functor $\mathsf{PartFinSet} \to \mathsf{FinCoh}$ taking a set I to the discrete coherence space $\Delta(I) = (I, \{(i,i) \mid i \in I\})$. Note that we have $\Delta(I)^\perp = (I, I \times I)$, which may be regarded as the codiscrete coherence space generated by I. On objects of $\mathcal{M}_{\mathrm{aff}}$, we define F as

$$F\left(\bigotimes_{i\in I} A_i\right) = \bigodot_{i \in \Delta(I)^\perp} A_i$$

For morphisms $(f, (\alpha_j)_{j\in J}) \in \mathsf{Hom}_{\mathcal{M}_{\mathrm{aff}}}\left(\bigotimes_{i\in I} A_i, \bigotimes_{j\in J} B_j\right)$, we set

$$F(f, (\alpha_j)_{j\in J}) = (\{(j,i) \mid j = f(i)\}, (\alpha_j)_{j\in J})$$

It is rather straightforward to check that F is indeed full, faithful and strong monoidal, and the extension to a morphism $\mathfrak{M}_{\mathrm{aff}} \to \mathfrak{M}_{\mathrm{coh}}$ is immediate. □

Proposition 6.4.9. *$\mathcal{M}_{\mathrm{coh}}$ also has cartesian products, which may be defined as*

$$\left(\bigodot_{i\in I} A_i\right) \,\&\, \left(\bigodot_{j\in J} B_j\right) = \bigodot_{x \in I \oplus J} \begin{cases} A_i & \text{if } x = \mathsf{in}_1(i) \\ B_j & \text{if } x = \mathsf{in}_2(j) \end{cases}$$

(The proof is left to the reader.) Therefore, we can extend Proposition 6.4.8:

Corollary 6.4.10. *There is a functor $E : (\mathcal{M}_{\mathrm{aff}})_{\&} \to \mathcal{M}_{\mathrm{coh}}$ that is full, faithful and lax (but* not *strong) monoidal, extending to a morphism of streaming settings $(\mathfrak{M}_{\mathrm{aff}})_{\&} \to \mathfrak{M}_{\mathrm{coh}}$.*

In the following proof and the rest of this section, we write explicitly $\mathbf{I}_{\mathrm{coh}}$ for the monoidal unit of $\otimes$ in $\mathcal{M}_{\mathrm{coh}}$ and $\mathbf{I}_{\mathrm{aff}\&}$ for the unit in $(\mathcal{M}_{\mathrm{aff}})_{\&}$.

Proof idea. The universal property of the free product completion defines E as the unique product-preserving functor extending the functor of Proposition 6.4.8. It remains to equip it with a lax monoidal functor structure. The map $m^0 : \mathbf{I}_{\mathrm{coh}} \to E(\mathbf{I}_{\mathrm{aff}\&})$ is an obvious isomorphism, while the natural transformation $m^2_{A,B} : E(A) \otimes E(B) \to E(A \otimes B)$ can be obtained via the canonical map

$$\left(\mathop{\&}_{i\in I} A_i\right) \otimes \left(\mathop{\&}_{j\in J} B_j\right) \to \mathop{\&}_{(i,j)\in I\times J} A_i \otimes B_j$$

in $\mathcal{M}_{\mathrm{coh}}$ (it exists in all monoidal categories with products). □

We can now go the other way around.

Lemma 6.4.11. *There is a strong monoidal functor $\mathcal{M}_{\mathrm{coh}} \to (\mathcal{M}_{\mathrm{aff}})_{\&}$, which extends to a morphism of streaming settings $\mathfrak{M}_{\mathrm{coh}} \to (\mathfrak{M}_{\mathrm{aff}})_{\&}$.*

Proof. For a coherence space $(\|X\|, \frown_X)$, write $\mathrm{Cl}(X) \subseteq \mathcal{P}(X)$ the set of *cliques* of X

$$\mathrm{Cl}(X) = \{S \in \mathcal{P}(X) \mid \forall x\ y \in S.\ x \frown_X y\}$$

We now define the functor $F : \mathcal{M}_{\mathrm{coh}} \to (\mathcal{M}_{\mathrm{aff}})_{\&}$ on objects as

$$F\left(\bigodot_{x\in X} A_x\right) = \mathop{\&}_{S\in \mathrm{Cl}(X)} \bigotimes_{x\in S} A_x$$

As for morphisms, first recall that a morphism $(R,\alpha) \in \mathsf{Hom}_{\mathcal{M}_{\mathrm{coh}}}\left(\bigodot_{x\in X} A_x, \bigodot_{y\in Y} B_y\right)$ consists of a linear map $R \in \mathsf{Hom}_{\mathsf{FinCoh}}(Y,X)$ and a family

$$(\alpha_y)_{y\in\|Y\|} \in \prod_{y\in\|Y\|} \mathsf{Hom}_{\mathcal{M}}\left((A_x)_{(y,x)\in R}, B_y\right)$$

We set out to define

$$F(R,\alpha) \in \mathsf{Hom}_{(\mathcal{M}_{\mathrm{aff}})_{\&}}\left(F\left(\bigotimes_{x\in S} A_x\right), F\left(\bigotimes_{y\in S'} B_y\right)\right)$$

recalling that

$$\mathsf{Hom}_{(\mathcal{M}_{\mathrm{aff}})_{\&}}\left(F\left(\bigotimes_{x\in S} A_x\right), F\left(\bigotimes_{y\in S'} B_y\right)\right)$$

$$= \prod_{S'\in \mathrm{Cl}(Y)} \sum_{S\in\mathrm{Cl}(X)} \sum_{f:S\twoheadrightarrow S'} \prod_{y\in S'} \mathsf{Hom}_{\mathcal{M}}\left((A_x)_{x\in f^{-1}(x)}, B_y\right)$$

So fixing $S' \in \mathrm{Cl}(Y)$ and recall that R being linear means that we have

$$(y,x)\in R \wedge (y',x)\in R \quad \Rightarrow \quad \begin{cases} y \frown_Y y' & \Rightarrow \quad x \frown_X x' \quad (1) \\ x = x' & \Rightarrow \quad y \asymp_Y y' \quad (2) \end{cases}$$

In particular, (1) implies that $\{x \in \|X\| \mid (y,x)\in R\}$ is a clique; we take that to be S. (2), and the fact that $y \frown_Y y' \wedge y \asymp_Y y' \Rightarrow y = y'$, imply that R determines a (total) function $S \twoheadrightarrow S'$, which we take to be f. Finally, once $y \in S'$ is fixed, we pick the component α_y to complete the definition, which makes sense as $f^{-1}(y) = \{x \in S \mid (y,x)\in R\} = \{x \in \|X\| \mid (y,x)\in R)\}$. This completes the definition of $F(R,\alpha)$; we leave checking functoriality to the reader.

Now, we turn to defining a morphism $\mathfrak{M}_{\mathrm{coh}} \to (\mathfrak{M}_{\mathrm{aff}})_{\&}$ from F. To this end, we first equip F with a strong monoidal structure (though a lax one would suffice for our definition of morphism of tree streaming settings). The key computation that allows this is as follows:

$$\begin{aligned} F\left(\bigodot_{x\in X} A_x\right) \otimes F\left(\bigodot_{y\in Y} B_y\right) &\cong \left(\mathop{\&}_{S\in\mathrm{Cl}(X)} \bigotimes_{x\in S} A_x\right) \otimes \left(\mathop{\&}_{S'\in\mathrm{Cl}(Y)} \bigotimes_{y\in S'} B_y\right) \\ &\cong \mathop{\&}_{\substack{S\in\mathrm{Cl}(X)\\ S'\in\mathrm{Cl}(Y)}} \left(\bigotimes_{x\in X} A_x\right) \otimes \left(\bigotimes_{y\in Y} B_y\right) \\ &\cong \mathop{\&}_{S\in\mathrm{Cl}(X\&Y)} \bigotimes_{z\in S} C_z \quad \text{where } C_{\mathsf{in}_1(x)} = A_x,\ C_{\mathsf{in}_2(y)} = B_y \\ &\cong F\left(\bigodot_{z\in X\&Y} C_z\right) \quad \cong \quad F\left(\left(\bigodot_{x\in X} A_x\right) \otimes \left(\bigodot_{y\in Y} B_y\right)\right) \end{aligned}$$

Finally, there is a canonical isomorphism $F(\perp\!\!\!\perp) \cong \perp\!\!\!\perp$ which completes the definition of our morphism $\mathfrak{M}_{\mathrm{coh}} \to (\mathfrak{M}_{\mathrm{aff}})_{\&}$. □

We thus conclude this section by first specializing the above to the case $\mathcal{M} = \mathcal{TR}^{\leq 1}$, and then making a final tangential observation.

Lemma 6.4.12. *There are morphisms of streaming settings* $\mathfrak{TR}^{\leq 1}_{\&} \to \mathfrak{TR}^{\asymp,\leq 1} \to \mathfrak{TR}^{\leq 1}_{\&}$. *In particular,* $\mathfrak{TR}^{\leq 1}_{\&}$*-BRTTs compute exactly the regular functions.*

Remark 6.4.13. One idea that one could take from Hu and Joyal's "coherence completion" of categories [HJ99] – but that we do not explore further here – is to look at objects whose indexing coherence spaces are (up to isomorphism) generated from singletons by the $\&/\oplus$ operations of FinCoh. (Those are called "contractible" in [HJ99, Section 4], and considering coherence spaces as undirected graphs, this corresponds to the classical notion of *cograph* in combinatorics.)

In the case of the coherence completion of some category $\mathcal{C}$, the full subcategory spanned by such objects turns out to be the free completion of $\mathcal{C}$ under finite products and coproducts (which differs from our $(-)_{\oplus\&}$ in not making '&' distribute over '$\oplus$'); this is formalized as a universal property in [HJ99, Theorem 4.3]. In the same vein, we conjecture that the full subcategory of $\mathcal{M}_{\mathrm{coh}}$ consisting of cograph-indexed objects – that is, of objects that are generated from those of $\mathcal{M}$ by means of the operations $\otimes/\&$ in $\mathcal{M}_{\mathrm{coh}}$ – is in some way the *free affine symmetric monoidal category with products generated by the multicategory* $\mathcal{M}$.

6.5. $\mathcal{TR}_{\oplus\&}$ IS MONOIDAL CLOSED

Now, we consider the category $\mathcal{TR}_{\oplus\&}$. Much like for $\mathcal{SR}_{\oplus\&}$ (§4.5), we have:

Theorem 6.5.1. *The category* $\mathcal{TR}_{\oplus\&}$ *has cartesian products, coproducts and a symmetric monoidal* closed *structure.*

Remark 6.5.2. Given the close relationship between the constructions $((-)_{\mathrm{aff}})_{\&}$ and $(-)_{\mathrm{coh}}$, it seems plausible that $(\mathcal{TR}_{\mathrm{coh}})_{\oplus}$ could be monoidal closed (we know that it has products, coproducts and a symmetric monoidal structure). We leave this question to further work.

This structure over $\mathcal{TR}_{\oplus\&}$ is obtained in the same way as for strings: the monoidal product over $\mathcal{TR}$ is defined as distributing over formal sums and products and the usual products and coproducts are created by the $(-)_{\oplus\&}$ completion. Similarly, monoidal closure can be obtained by applying the generic Theorem 4.5.3 once we show that the objects coming from $\mathcal{TR}$ have internal homsets $\mathcal{TR}_{\oplus}$ (echoing Lemma 4.3.14):

Lemma 6.5.3. *For any* $\mathbf{R}, \mathbf{S} \in \mathsf{Obj}(\mathcal{TR})$*, there is an internal hom* $\iota_{\oplus}(\mathbf{R}) \multimap \iota_{\oplus}(\mathbf{S})$ *in* $\mathcal{TR}_{\oplus}$.

The rest of this section is dedicated to proving this fact, whose proof relies on decomposing linear trees in a similar way as in Lemma 6.3.4.

Proof. First, we treat the special case where $\mathbf{S} = \mathcal{I}(U)$ for some finite set U. To make sense of the definition of $\iota_{\oplus}(\mathbf{R}) \multimap \iota_{\oplus}(\mathcal{I}(U))$ it is helpful to notice that it will ultimately induce an isomorphism

$$\mathsf{Hom}_{\mathcal{TR}_{\oplus}}(\top, \iota_{\oplus}(\mathbf{R}) \multimap \iota_{\oplus}(\mathcal{I}(U))) \cong \mathsf{Hom}_{\mathcal{TR}}(\mathbf{R}, \mathcal{I}(U)) \cong \mathbf{LTree}(\mathbf{R} \otimes \mathcal{O}(U))$$

so, recalling that objects of $\mathcal{TR}_{\oplus}$ are of the shape $\bigoplus_{i\in I} \bigotimes_{j\in J_i} \mathcal{I}(V_j)$ for V_j being finite sets, the operational intuition is that one may code trees with "holes with arity" into some bounded finitary data (which we may informally call a partitioning tree) plus finitely many trees containing holes "without arity"; this bijection is pictured in Figure 6.5.1. As with Lemma 6.3.4, we will not use this as our official definition for the internal homset, but rather use the following recursive definition:

- If $\mathbf{R} = \top$, set $\iota_{\oplus}(\mathbf{R}) \multimap \iota_{\oplus}(\mathcal{I}(U)) \;=\; \iota_{\oplus}(\mathcal{I}(U))$.
- Otherwise, define $\iota_{\oplus}(\mathbf{R}) \multimap \iota_{\oplus}(\mathcal{I}(U))$ as

$$\bigoplus_{U=V\uplus W} \iota_{\oplus}(\mathcal{I}(V+1)) \otimes \left(\bigoplus_{b\in\mathbf{\Gamma}} (\mathbf{R} \multimap_b \mathcal{I}(W)) \oplus \bigoplus_{r\in R} (\mathbf{R} \multimap_r \mathcal{I}(W)) \right)$$

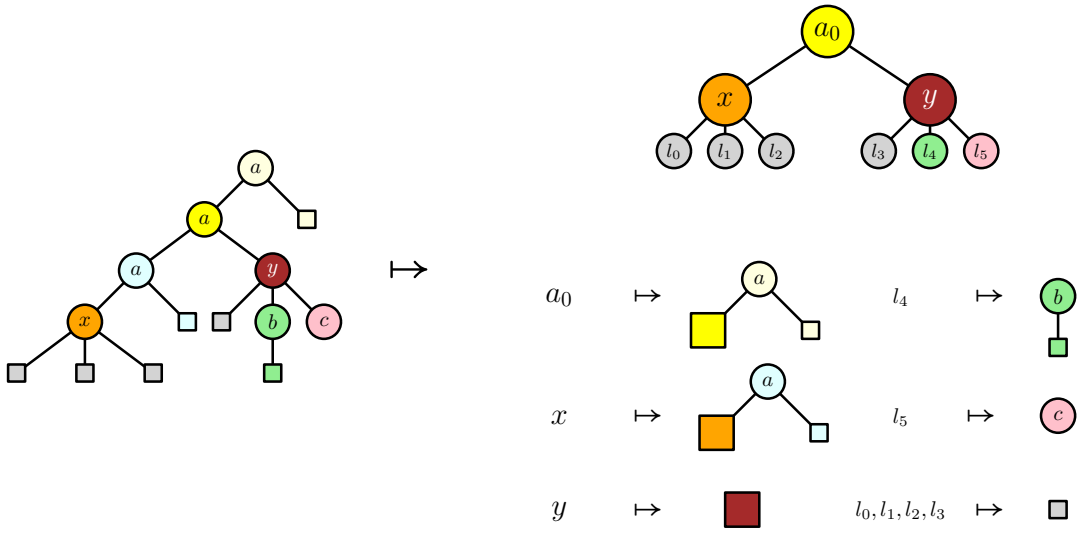


FIGURE 6.5.1. Decomposition of a map of $\mathsf{Hom}_{\mathcal{TR}(\mathbf{\Sigma})}(\{x:3, y:3\}, \mathcal{I}(7))$ (which is defined as $\mathbf{LTree}_{\mathbf{\Sigma}}(\{x:3, y:3\} \otimes \mathcal{O}(7))$, for $\mathbf{\Sigma} = \{a:2, b:1, c:0\}$) as a tuple consisting of a partitioning tree and trees without the letters x and y.

where $\mathbf{R} \multimap_r \mathcal{I}(W)$ and $\mathbf{R} \multimap_b \mathcal{I}(W)$ are auxiliary definitions which correspond to the following situations (recalling that morphisms can be regarded as trees):

– $\iota_\oplus(\mathcal{I}(V+1)) \otimes (\mathbf{R} \multimap_r \mathcal{I}(W))$ correspond to morphisms such that there is a *unique* minimal path leading from the root to a node labelled by a letter in R, and that letter is r. The second component $\mathbf{R} \multimap_r \mathcal{I}(W)$ is meant to include the immediate subtrees of that node while the first $\iota_\oplus(\mathcal{I}(V+1))$ contains the tree where a nullary node labeled is inserted instead of that node. The combination of both these data and r allows to recover the original morphism.

– $\bigoplus_{b \in \mathbf{\Gamma}} \iota_\oplus(\mathcal{I}(V+1)) \otimes \mathbf{R} \multimap_b \mathcal{I}(W)$ correspond to all the other morphisms. In such a case there is a topmost node labelled by some letter b of $\mathbf{\Gamma}$ with which has at least two distinct immediate subtrees which have at least one node of R each. Similarly to the first subcase, $\mathbf{R} \multimap_b \mathcal{I}(W)$ is intended to include the immediate subtrees of that node labeled by b while $\iota_\oplus(\mathcal{I}(V+1))$ contains the tree where a nullary node labeled is inserted instead of that node. The combination of these data and b allows to recover the original morphism.

Their formal definition is as follows:

$$\mathbf{R} \multimap_b \mathcal{I}(W) \quad = \quad \bigoplus_{\substack{f:W \to \mathrm{ar}(b) \\ g:R \to \mathrm{ar}(b) \\ g \text{ nonconstant}}} \bigotimes_{x \in \mathrm{ar}(\Gamma)} \iota_\oplus(\mathbf{R} \restriction g^{-1}(x)) \multimap \iota_\oplus(f^{-1}(x))$$

$$\mathbf{R} \multimap_r \mathcal{I}(W) \quad = \quad \bigoplus_{\substack{f:W \to \mathrm{ar}(r) \\ g:R \setminus \{r\} \to \mathrm{ar}(r)}} \bigotimes_{x \in \mathrm{ar}(r)} \iota_\oplus(\mathbf{R} \restriction g^{-1}(x)) \multimap \iota_\oplus(f^{-1}(x))$$

Note that the definitions of $\iota_\oplus(\mathbf{R}) \multimap \iota_\oplus(\mathcal{I}(U))$, $\mathbf{R} \multimap_b \mathcal{I}(W)$ and $\mathbf{R} \multimap_r \mathcal{I}(W)$ mutually depend on one another. Still this is is well-defined as the definitions of $\mathbf{R} \multimap_b \mathcal{I}(W)$ and $\mathbf{R} \multimap_r \mathcal{I}(W)$ only require $\iota_\oplus(\mathbf{S}) \multimap \iota_\oplus(\mathcal{I}(V))$ for $\mathbf{S}$ strictly smaller than $\mathbf{R}$.

We now describe the associated evaluation map

$$\mathrm{ev}_{\mathbf{R},\mathcal{I}(U)} \ : \ (\iota_{\oplus}(\mathbf{R}) \multimap \iota_{\oplus}(\mathcal{I}(U))) \otimes \mathbf{R} \ \longrightarrow \ \mathcal{I}(U)$$

also by recursion over $\mathbf{R}$.

- If $\mathbf{R} = \top$, it is the identity.
- Otherwise, we need to provide maps

$$\left(\mathcal{I}(V+1) \otimes \left(\bigoplus_{b\in\mathbf{\Gamma}} (\mathbf{R} \multimap_b \mathcal{I}(W)) \oplus \bigoplus_{r\in R} (\mathbf{R} \multimap_r \mathcal{I}(W))\right)\right) \otimes \mathbf{R} \ \longrightarrow \ \mathcal{I}(U)$$

for every decomposition $U = V \uplus W$, the intuition being that $\mathcal{I}(V+1)$ is a context containing the top of the tree corresponding to the function we want to apply. Therefore, once we provide a map

$$\left(\bigoplus_{b\in\mathbf{\Gamma}} (\mathbf{R} \multimap_b \mathcal{I}(W)) \oplus \bigoplus_{r\in R} (\mathbf{R} \multimap_r \mathcal{I}(W))\right) \otimes \mathbf{R} \ \longrightarrow \ \mathcal{I}(W)$$

we may post-compose it with $\mathcal{I}(V+1) \otimes \mathcal{I}(W) \to \mathcal{I}(V+W) \cong \mathcal{I}(V \uplus W) = \mathcal{I}(U)$ to define $\mathrm{ev}_{\mathbf{R}\multimap\mathcal{I}(U)}$. Recalling that $\otimes$ distributes over $\oplus$, it suffices to provide specialized maps

$$\mathrm{ev}^b_{\mathbf{R},\mathcal{I}(W)} : \mathbf{R} \multimap_b \mathcal{I}(W) \otimes \mathbf{R} \to \mathcal{I}(W) \quad \text{and} \quad \mathrm{ev}^r_{\mathbf{R},\mathcal{I}(W)} : \mathbf{R} \multimap_r \mathcal{I}(W) \otimes \mathbf{R} \to \mathcal{I}(W)$$

for $b \in \Gamma$ and $r \in R$, which we describe now.

– For $\mathrm{ev}^b_{\mathbf{R},\mathcal{I}(W)}$, it suffices to define a family of $\mathcal{TR}$-maps indexed by $f : W \to \mathrm{ar}(b)$ and $g : R \to \mathrm{ar}(b)$ with g non-constant

$$\left(\bigotimes_{x\in\mathrm{ar}(\Gamma)} (\mathbf{R} \upharpoonright g^{-1}(x)) \multimap \mathcal{I}(f^{-1}(x))\right) \otimes \mathbf{R} \ \longrightarrow \ \mathcal{I}(W)$$

Recall that b can be seen as tree constructor and induces a canonical map

$$\overline{b} \ : \ \bigotimes_{x\in\mathrm{ar}(b)} \mathcal{I}(f^{-1}(x)) \ \longrightarrow \ \mathcal{I}(W)$$

Using the induction hypothesis, we have evaluation maps

$$\big((\mathbf{R} \upharpoonright g^{-1}(x)) \multimap \mathcal{I}(f^{-1}(x))\big) \otimes (\mathbf{R} \upharpoonright g^{-1}(x)) \ \longrightarrow \ \mathcal{I}(W)$$

We can then compose $\overline{b}$ with the product of those maps over $x \in \mathrm{ar}(b)$ and then the isomorphism $\mathbf{R} \cong \bigotimes_{x\in\mathrm{ar}(b)} \mathbf{R} \upharpoonright f^{-1}(x)$ to conclude the definition of $\mathrm{ev}^b_{\mathbf{R},\mathcal{I}(W)}$.

– For $\mathrm{ev}^r_{\mathbf{R},\mathcal{I}(W)}$, it suffices to define a family of $\mathcal{TR}$-maps indexed by $f : W \to \mathrm{ar}(r)$ and $g : R \setminus \{r\} \to \mathrm{ar}(r)$

$$\left(\bigotimes_{x\in\mathrm{ar}(\Gamma)} (\mathbf{R} \upharpoonright g^{-1}(x)) \multimap \mathcal{I}(f^{-1}(x))\right) \otimes \mathbf{R} \ \longrightarrow \ \mathcal{I}(W)$$

By exploiting the isomorphism $\mathbf{R} \cong \mathcal{I}(\mathrm{ar}(r)) \otimes \bigotimes_{x\in\mathrm{ar}(r)} (\mathbf{R} \upharpoonright g^{-1}(x))$ and using the inductive hypothesis as in the previous case, we obtain a map

$$\left(\bigotimes_{x\in\mathrm{ar}(\Gamma)} (\mathbf{R} \upharpoonright g^{-1}(x)) \multimap \mathcal{I}(f^{-1}(x))\right) \otimes \mathbf{R} \ \longrightarrow \ \mathcal{I}(\mathrm{ar}(r)) \otimes \bigotimes_{x\in\mathrm{ar}(r)} \mathcal{I}(f^{-1}(x))$$

and we may conclude by post-composing by the map

$$\mathcal{I}(\mathrm{ar}(r)) \otimes \bigotimes_{x \in \mathrm{ar}(r)} \mathcal{I}(f^{-1}(x)) \to \mathcal{I}(W)$$

which is induced by the depth-2 tree whose root corresponds to the first component, whose children correspond to the successive elements of $\bigotimes_{x \in \mathrm{ar}(r)} \mathcal{I}(f^{-1}(x))$ and other leaves are in W.

While the definition is a bit wordy, there is then little difficulty in checking that this yields the expected universal property.

$$\begin{array}{ccc} \iota_\oplus(\mathbf{R} \multimap \mathcal{I}(U)) \otimes \iota_\oplus(\mathcal{I}(U)) & \longrightarrow & \iota_\oplus(\mathcal{I}(U)) \\ {\scriptstyle \Lambda(h)\otimes \mathrm{id}} \uparrow & \nearrow_{h} & \\ A \otimes \iota_\oplus(\mathbf{R}) & & \end{array}$$

One needs then to extend the definition for the general case where $\mathbf{S}$ is not necessarily $\mathcal{I}(U)$; this is done using a similar approach as for strings, by using a coproduct over partial maps $R \rightharpoonup S$ tracking which letter of the input participates in which letter of the output, and employing the particular case where there is one letter in the output[3]

$$\iota_\oplus(\mathbf{R}) \multimap \iota_\oplus(\mathbf{S}) \;=\; \bigoplus_{f : R \rightharpoonup S} \bigotimes_{s \in S} \iota_\oplus(\mathbf{R} \upharpoonright f^{-1}(s)) \multimap \iota_\oplus(\mathcal{I}(\mathrm{ar}(s)))$$

The evaluation function can then be extended and the universal property accordingly lifted to the more general case. □

6.6. Preservation properties of finite completions

We have now proven all of the combinatorial results leading to the main theorem for trees. At this stage, we only need to make explicit a few preservation properties of the completions $(-)_\oplus$ and $(-)_\&$. We only state the minimal requirements that we need to proceed[4].

Lemma 6.6.1. *Let $\square \in \{\oplus, \&, \oplus\&\}$ and $\mathfrak{C}, \mathfrak{D}$ be streaming settings. Then if there is a morphism $\mathfrak{C} \to \mathfrak{D}$, there is also a morphism $\mathfrak{C}_\square \to \mathfrak{C}_\square$.*

Proof idea. Let us only discuss the case $\square = \oplus$ and assume that we have a streaming setting morphism whose underlying lax monoidal functor is $F : \mathcal{C} \to \mathcal{D}$. Then we consider $F_\oplus : \mathcal{C}_\oplus \to \mathcal{D}_\oplus$ defined using the universal property of $\mathcal{C}_\oplus$ from the functor $\iota_\oplus \circ F : \mathcal{C} \to \mathcal{D}_\oplus$ such that

$$F_\oplus\left(\bigoplus_{u \in U} \iota_\oplus(C_u)\right) \;=\; \bigoplus_{u \in U} \iota_\oplus(F(C_u))$$

so that $F_\oplus$ inherits a lax monoidal structure from F by lifting the relevant maps functorially. For instance, we have $\iota_\oplus(\mathbf{I}) \to \iota_\oplus(F(\mathbf{I})) = F_\oplus(\mathbf{I})$ for the unit. We leave checking the coherence diagram, the case of the binary tensor to the reader, as well as checking that the rest of the structure of morphisms of streaming settings lift accordingly. □

Lemma 6.6.2. *Let $\mathfrak{C}$ be a streaming setting. Then there are morphisms of streaming settings $\mathfrak{C}_{\oplus\&} \to (\mathfrak{C}_\oplus)_{\oplus\&}$ and $(\mathfrak{C}_\oplus)_{\oplus\&} \to \mathfrak{C}_{\oplus\&}$.*

[3] This could be factored out as a more general result concerning the existence of internal homsets in categories of the shape $\mathcal{M}_{\mathrm{aff}}$ for multicategories $\mathcal{M}$.

[4] The following, which would entail these requirements, should hold (but we have not checked it): the completions $(-)_\oplus$ and $(-)_\&$ should behave as pseudo-monads over the 2-category of tree streaming settings, admitting a pseudo-distributive law $((-)_\oplus)_\& \to ((-)_\&)_\oplus$ giving rise to the pseudo-monad $(-)_{\oplus\&}$.

PROOF IDEA. The first morphism can be obtained using Lemma 6.6.1 with the morphism of streaming setting $\mathfrak{C} \to \mathfrak{C}_\oplus$ corresponding to $\iota_\oplus$. The second morphism can be written as a composite

$$(\mathfrak{C}_\oplus)_{\oplus\&} \cong ((\mathfrak{C}_\oplus)_\&)_\oplus \xrightarrow{(\mathfrak{Dist}_\mathfrak{C})_\oplus} ((\mathfrak{C}_\&)_\oplus)_\oplus \xrightarrow{\mathfrak{Mu}_{\mathfrak{C}_\&}} (\mathfrak{C}_\&)_\oplus \cong \mathfrak{C}_{\oplus\&}$$

where $(\mathfrak{Dist}_\mathfrak{C})_\oplus$ is obtained from a morphism $\mathfrak{Dist}_\mathfrak{C} : (\mathfrak{C}_\oplus)_\& \to (\mathfrak{C}_\&)_\oplus$ by Lemma 6.6.1. $\mathfrak{Dist}_\mathfrak{C}$ itself is built from a functor $D_\mathcal{C} : (\mathcal{C}_\oplus)_\& \to (\mathcal{C}_\&)_\oplus$ defined on object following the mantra "products distribute over coproducts".

$$D_\mathcal{C}\left(\mathop{\&}_{x\in X} \iota_\& \left(\bigoplus_{u\in U_x} \iota_\oplus(C_{x,u})\right)\right) = \bigoplus_{F\in\prod_x U_x} \iota_\oplus\left(\mathop{\&}_{x\in X} \iota_\&(C_{x,u})\right)$$

On the other hand, the functor $M : \mathcal{C}$ which is part of $\mathfrak{Mu}_\mathfrak{C}$ is obtained by using the universal property of $(-)_\oplus$ so that

$$M\left(\bigoplus_{u\in U} \iota_\oplus\left(\bigoplus_{v\in V} \iota_\oplus(C_{u,v})\right)\right) = \bigoplus_{(u,v)} \iota_\oplus(C_{u,v})$$

□

6.7. $\lambda\ell^{\oplus\&}$-DEFINABLE TREE FUNCTIONS ARE REGULAR

We will now finally prove our last result on the $\lambda\ell^{\oplus\&}$-calculus, that is, the third row in Theorem 1.2.3 whose statement is recalled in the section title.

First, recall from Section 5.2.1 that if $\mathbf{\Gamma} = \{a_1 : A_1, \dots, a_k : A_k\}$ is an output alphabet, we call $\widetilde{\mathbf{\Gamma}}$ the context $a_1 : \mathbb{o} \multimap \dots \multimap \mathbb{o}, \dots, a_k : \mathbb{o} \multimap \dots \multimap \mathbb{o}$ where the type of a_i has $|A_i|$ arguments. Let $\mathcal{L}(\widetilde{\mathbf{\Gamma}})$ (or $\mathcal{L}$ for short when $\widetilde{\mathbf{\Gamma}}$ is clear from context) be the syntactic category defined by replacing the alphabet Γ by a ranked alphabet $\mathbf{\Gamma}$ in Definition 5.2.1. Since $\mathcal{L}$ monoidal product, we may easily adapt Definition 5.2.3 to get a tree streaming setting $\mathfrak{L}(\mathbf{\Gamma})$. Then we may relate $\lambda\ell^{\oplus\&}$-definability to (single-state) $\mathfrak{L}$-BRTT.

Lemma 6.7.1. *Computability by single-state $\mathfrak{L}$-BRTTs and $\lambda\ell^{\oplus\&}$-definability (Definition 5.1.9) are equivalent for functions* $\mathbf{Tree}(\mathbf{\Sigma}) \to \mathbf{Tree}(\mathbf{\Gamma})$.

PROOF IDEA. Similar to Lemma 5.2.4, based on the syntactic Lemma 5.2.5. The proof is even more straightforward as there is no mismatch between the processing of trees by BRTTs and $\lambda\ell^{\oplus\&}$-terms working with Church encodings, whereas SSTs operate top-down rather than bottom-up when regarding strings as unary trees. □

We can now notice that $\mathfrak{L}$-BRTTs are more expressive than $\mathcal{TR}^{\leq 1}$-BRTTs, much like with strings (this generalizes Lemma 5.2.7).

Lemma 6.7.2. *There is a morphism of streaming settings $\mathfrak{TR} \to \mathfrak{L}$.*

PROOF SKETCH. Let us focus on the underlying functor $F : \mathcal{TR} \to \mathcal{L}$. For objects (which are finite sets), we put

$$F\left(\bigotimes_{i\in I} U_i\right) = \bigotimes_{i\in I} \left((\mathbb{o}^{\otimes U_i} \multimap \mathbb{o}) \,\&\, \mathbf{I}\right)$$

A multimorphism $f \in \mathsf{Hom}_{\mathcal{TR}^{\mathrm{m}}}((U_i)_{i\in I}, V)$ is an element of $\mathbf{LTree}_\mathbf{\Gamma}\left(\left(\bigotimes_{i\in I} \mathcal{I}(U_i)\right) \otimes \mathcal{O}(V)\right)$ which has a Church encoding $\underline{f}$ which has a type isomorphic to $\bigotimes_{i\in I}\left(\mathbb{o}^{\otimes U_i} \multimap \mathbb{o}\right) \multimap \mathbb{o}$, and thus embeds into $F\left(\bigotimes_{i\in I} U_i\right)$ through well-typed term ι. We take $F(f_{\mathrm{aff}}) = \lambda x.\iota\, \underline{f}$, and extend this definition to arbitrary morphisms $(f, (\alpha_j)_j) : \bigotimes_{i\in I} U_i \to \bigotimes_{j\in J} V_j$ in $\mathcal{TR}$ by first

using the second projection π_2 to restrict to the case where $\mathsf{dom}(f) = I$, and then by piecing together the $F((\alpha_j)_{\mathrm{aff}})$. □

Corollary 6.7.3. *There is a morphism of streaming settings $\mathfrak{TR}_{\oplus\&} \to \mathfrak{L}$.*

Proof idea. Starting from Lemma 6.7.2, we have a functor $\mathcal{TR} \to \mathcal{L}$. Since $\mathcal{L}$ has all products and coproducts, the universal properties of the $(-)_{\&}$ and $(-)_{\oplus}$ completion yield a functor $F : \mathcal{TR}_{\oplus\&} \to \mathcal{L}$. The monoidal structure of the initial functor $\mathcal{TR} \to \mathcal{L}$ can be lifted accordingly. For any finite family of objects $(((A_k)_{k\in K_j})_{j\in J_i})_{i\in I}$ sitting in a symmetric monoidal closed category with products and coproducts, there are canonical morphisms

$$\left(\bigoplus_{u\in U} \mathop{\&}_{x\in X_u} A_x\right) \otimes \left(\bigoplus_{v\in V} \mathop{\&}_{v\in Y_v} B_y\right) \longrightarrow \bigoplus_{(u,v)\in U\times V} \mathop{\&}_{(x,y)\in X_u\times Y_v} A_x \otimes B_y$$

which are not isomorphisms in general, but constitute the non-trivial part of the lax monoidal structure of F; $m^0 : \mathbf{I} \to F(\mathbf{I})$ is actually the identity. □

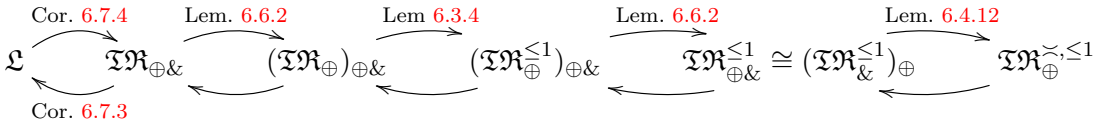
For the converse, we use Theorem 6.5.1, according to which $\mathcal{TR}_{\oplus\&}$ has all the categorical structure needed to model the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus. By a suitable adaptation of Lemma 5.2.6 to tree streaming settings, we then have:

Corollary 6.7.4. *There is a morphism of streaming settings $\mathfrak{L} \to \mathfrak{TR}_{\oplus\&}$.*

The proof of our main theorem for trees can now be summarized as

$\lambda\ell^{\oplus\&}$	=	single-state $\mathfrak{L}$-BRTTs	by Lemma 6.7.1
	=	single-state $\mathfrak{TR}_{\oplus}^{\asymp,\leq 1}$-BRTTs	(†)
	=	$\mathfrak{TR}^{\asymp,\leq 1}$-BRTTs	by Theorem 6.1.6
	=	regular tree functions	by Proposition 6.4.5

Note that applying Theorem 6.1.6 in the last step requires that objects of $\mathcal{TR}^{\asymp,\leq 1}$ have unitary support, which is easily checked for output alphabets $\mathbf{\Gamma}$ containing at least one letter of arity $\varnothing$. It remains to justify step (†), i.e., that single-state $\mathfrak{L}$-BRTTs and $\mathfrak{TR}_{\oplus}^{\asymp,\leq 1}$-BRTTs are equi-expressive. By Lemma 6.1.5, it suffices to have morphisms of streaming settings both ways $\mathfrak{L} \leftrightarrow \mathfrak{TR}_{\oplus}^{\asymp,\leq 1}$ to conclude. They may be obtained as the following composites (where we leave implicit the uses of Lemma 6.6.1 and some easily inferrable steps).



CHAPTER 7

Star-free languages in non-commutative linear logic

This is the final technical chapter of this dissertation, and it is devoted to Theorem 1.2.1: our characterization of star-free languages using a non-commutative linear λ-calculus. But whence comes non-commutativity? Consider the λ-terms

$$\mathtt{id}' = \lambda f.\ \lambda x.\ \lambda y.\ f\ x\ y \qquad \mathtt{flip} = \lambda f.\ \lambda x.\ \lambda y.\ f\ y\ x$$

Then iterating $\mathtt{flip}$ yields $\mathtt{flip}, \mathtt{id}', \mathtt{flip}, \mathtt{id}', \dots$ up to β-equivalence. In other words, the sequence defined by the recurrence $\mathtt{flip}^{n+1} = \lambda f.\ \mathtt{flip}\ (\mathtt{flip}^n\ f)$ is described by the closed forms $\mathtt{flip}^{2n+1} =_\beta \mathtt{flip}$ and $\mathtt{flip}^{2n+2} =_\beta \mathtt{id}'$; it has period two since $\mathtt{flip} \neq_\beta \mathtt{id}$. To enforce the *aperiodicity* of certain monoids of λ-terms, we must therefore exclude $\mathtt{flip}$, whose only action is to *exchange* the two arguments of its input f; thus, we are led to require that functions *use their arguments in the same order that they are given in.*

It is well-known that in order to make such a restriction behave well, one needs to restrict duplication in some way. Indeed, consider the β-reduction

$$\lambda f.\ \lambda x.\ \lambda y.\ (\lambda a.\ f\ a\ a)\ (x\ y) \quad \longrightarrow_\beta \quad \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

While the term on the left-hand side satisfies the non-commutativity condition, its reduct on the right-hand side does not, and this is due to the two copies of the subterm $(x\ y)$. This led us to use a non-commutative *affine* λ-calculus in the paper [NP20], upon which this chapter is largely based.

However, instead of merely copy-pasting the contents of [NP20], we have chosen here to work with *linear* rather than affine types. While this choice did not make much difference in the case of the $\lambda\ell^{\oplus\&}$-calculus of the two previous chapters (cf. Remark 5.1.1), this was because the $\lambda\ell^{\oplus\&}$-calculus features *additive* connectives $\oplus/\&$. The latter must also be excluded to capture star-free languages: it is not hard to check that any regular language can be defined by a "non-commutative $\lambda\ell^{\oplus\&}$-term" of type $\mathtt{Str}_\Sigma[\mathbf{I} \oplus \cdots \oplus \mathbf{I}] \multimap \mathbf{I} \oplus \mathbf{I}$ (see also [NP20, §5.2] for an encoding using '&'). And without additives, it is much less obvious that the absence of weakening (the ability for functions to discard their arguments) does no harm to the expressive power.

In fact, we will show right now that this is wrong concerning string-to-string functions. In what follows, we speak of the $\lambda\ell$-calculus to refer to the fragment of the $\lambda\ell^{\oplus\&}$-calculus containing only the connectives[1] '$\multimap$' and '$\to$' on types, and the corresponding term constructors. It is shown in [NP20, §5.1] that the affine variant of the $\lambda\ell$-calculus can express all sequential functions (§2.2); in fact, we will see in upcoming work that the functions that are "affine $\lambda\ell$-definable" (in the sense of Definition 5.1.9, *mutatis mutandis*) are exactly the regular functions, as announced in Section 1.4.2 of the introduction. However:

Proposition 7.0.1. *The example of letter-to-letter aperiodic sequential function given in Proposition 2.3.9 is not $\lambda\ell$-definable.*

Proof. It does not satisfy the conclusion of the theorem below. □

[1] The exclusion of the tensor product has no effect on expressiveness: $\sigma \otimes \tau$ can be simulated most of the time by $(\sigma \multimap \tau \multimap \kappa) \multimap \kappa$ for a judicious choice of κ. This trick will reoccur in our definition of $\mathtt{NBool}$.

Theorem 7.0.2. *Let $f : \Gamma^* \to \Sigma^*$ be defined by a $\lambda\ell$-term $t : \mathtt{Str}_\Gamma[\tau] \multimap \mathtt{Str}_\Sigma$ where τ is purely linear. The Parikh vector (containing the number of occurrences of each letter) of $f(w)$ is entirely determined by the Parikh vector of $w \in \Gamma^*$. Moreover, the relationship is given by an affine map (in the sense of linear algebra) with integer coefficients.*

Proof sketch. First, we claim that an analogous property to Lemma 5.2.5 holds in the $\lambda\ell$-calculus: $t =_{\beta\eta} \lambda s.\lambda^! a_1 \dots \lambda^! a_{|\Sigma|}.\lambda^! a_\varepsilon.\ o\ (s\ d_1\ \dots\ d_{|\Gamma|}\ d_\varepsilon)$ where o and the d_i are purely linear $\lambda\ell$-terms with no occurrence of s. Since we only have negative connectives in this fragment, this can be established much more easily than Lemma 5.2.5, by examining β-normal η-long forms.

The idea is then to work with the following "measure" on purely linear $\lambda\ell$-terms with the non-linear free variables $a_1, \dots, a_{|\Sigma|}, a_\varepsilon$: $[\![t]\!]_c$ is the number of occurrences in t of a_c (i.e. a_i when c is the i-th letter of Σ). In fact this is a denotational semantics: it is not hard to check invariance by $=_{\beta\eta}$ manually, but more conceptually, it corresponds to the commutative monoid $(\mathbb{N}, +)$ seen as a one-object monoidal closed (and even compact closed) category.

If $x \in \mathbb{N}^\Gamma$ and $y \in \mathbb{N}^\Sigma$ are the Parikh vectors of w and $f(w)$ respectively, we then have $y = Ax+b$ where $A \in \mathbb{N}^{\Sigma\times\Gamma}$ and $b \in \mathbb{N}^\Sigma$ are defined by $A_{c,c'} = [\![d_{c'}]\!]_c$ and $b_c = [\![d_\varepsilon]\!]_c + [\![o]\!]_c$. □

Despite this, moving from affineness to linearity does not affect the expressive power of our non-commutative λ-calculus which we call the $\lambda\ell\wp$-calculus. In other words, one can express all star-free languages without weakening. Since linearity is more restrictive than affineness, the fact that $\lambda\ell\wp$-definable languages are regular – i.e. the "only if" part of Theorem 1.2.1 – is already a consequence of the work done in [NP20], so **we will skip its proof here**. Instead, this chapter focuses on the converse: we show how to encode star-free languages as $\lambda\ell\wp$-terms. While it is relatively self-contained, it is intended more as a companion to [NP20]; we therefore recommend reading [NP20] as well.

7.1. The $\lambda\ell\wp$-calculus

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

$$\sigma, \tau ::= o \mid \sigma \to \tau \mid \sigma \multimap \tau \qquad t, u ::= x \mid t\,u \mid \lambda^! x.\,t \mid \lambda x.\,t$$

There are two rules for β-reduction (closed under contexts)

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

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

Just as in the case of the $\lambda\ell^{\oplus\&}$-calculus (cf. §5.1), the typing judgements make use of dual contexts (it is indeed a common feature, originating in [Bar96]): they are of the form $\Psi; \Delta \vdash t : \tau$. The key difference with the $\lambda\ell^{\oplus\&}$-calculus is that Δ is an *ordered list* of bindings – this order is essential for non-commutativity. The typing rules are as follows, where $\Delta \cdot \Delta'$ denotes the *concatenation of the ordered lists* Δ *and* Δ':

$$\frac{}{\Psi,\, x : \tau;\, \varnothing \vdash x : \tau} \qquad \frac{\Psi;\, \Delta \vdash t : \sigma \to \tau \qquad \Psi;\, \varnothing \vdash u : \sigma}{\Psi;\, \Delta \vdash t\,u : \tau} \qquad \frac{\Psi,\, x : \sigma;\, \Delta \vdash t : \tau}{\Psi;\, \Delta \vdash \lambda^! x.\,t : \sigma \to \tau}$$

$$\frac{}{\Psi;\, x : \tau \vdash x : \tau} \qquad \frac{\Psi;\, \Delta \vdash t : \sigma \multimap \tau \qquad \Psi;\, \Delta' \vdash u : \sigma}{\Psi;\, \Delta \cdot \Delta' \vdash t\,u : \tau} \qquad \frac{\Psi;\, \Delta \cdot (x : \sigma) \vdash t : \tau}{\Psi;\, \Delta \vdash \lambda x.\,t : \sigma \multimap \tau}$$

This coresponds to the fragment of Polakow and Pfenning's Intuitionistic Non-Commutative Linear Logic [PP99a; PP99b] that contains only and '$\multimap$' are called "intuitionistic functions" ($\to$) and "right ordered functions" ($\multimap$) in the terminology of [PP99a]. (Their system additionally contains "linear [commutative] functions" and "left ordered functions".)

Remark 7.1.1. Morally, the non-affine variables "commute with everything". More formally, one could translate the $\lambda\ell\wp$-calculus into a non-commutative version of Intuitionistic Affine Logic whose exponential modality '!' incorporates the customary rules (see e.g. [Yet90])

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

The following property is analogous to the argument in the proof of Proposition 5.1.16.

Lemma 7.1.2. *If* $\vdash t : \sigma_1[\tau] \multimap \sigma_2$ *and* $\vdash u : \sigma_2[\tau'] \multimap \sigma_3$, *then* $\vdash \lambda x.\, u\,(t\,x) : \sigma_1[\tau[\tau']] \multimap \sigma_3$.

7.2. Non-commutative linear data types

Now we must explain what the types involved in the definition of $\lambda\ell\wp$-definability are. The conventional linear boolean types do not work in the $\lambda\ell\wp$-calculus because of non-commutativity and of the lack of additive connectives. But the following choice of type

$$\mathtt{NBool} = ((\mathbb{o} \multimap \mathbb{o}) \multimap (\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o}) \multimap (\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o}$$

admits exactly two closed inhabitants (this would not be the case in an affine setting!):

$$\mathtt{true} = \lambda k.\, \lambda f.\, k\ (\lambda x.\, x)\ f \qquad \mathtt{false} = \lambda k.\, \lambda f.\, k\ f\ (\lambda x.\, x)$$

It is a sort of continuation-passing-style [Rey93] transformed version of the boolean type $(\mathbb{o} \multimap \mathbb{o}) \multimap (\mathbb{o} \multimap \mathbb{o}) \otimes (\mathbb{o} \multimap \mathbb{o})$ introduced by Matsuoka [Mat15]. (Interestingly, while Matsuoka works in commutative multiplicative linear logic, he mentions the potential relevance of planarity, i.e. non-commutativity, to expressivity questions.)

Let us show that a few pseudo-weakening functions are definable:

$$\mathtt{cstt} = \lambda b.\, \lambda k.\, \lambda f.\, b\ (\lambda g.\, \lambda h.\, k\ \mathtt{id}\ (f \circ g \circ h))\ \mathtt{id} : \mathtt{NBool} \multimap \mathtt{NBool}$$

$$\mathtt{cstf} = \lambda b.\, \lambda k.\, \lambda f.\, b\ (\lambda g.\, \lambda h.\, k\ (f \circ g \circ h)\ \mathtt{id})\ \mathtt{id} : \mathtt{NBool} \multimap \mathtt{NBool}$$

$$\mathtt{proj2} = \lambda b.\, b\ (\lambda g.\, \lambda h.\, g\ (h\ \mathtt{id}))\ \mathtt{id} : \mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}] \multimap \mathbb{o} \multimap \mathbb{o}$$

For $b \in \{\mathtt{true}, \mathtt{false}\}$, we have $\mathtt{cstt}\ b =_{\beta\eta} \mathtt{true}$, $\mathtt{cstf}\ b =_{\beta\eta} \mathtt{false}$ and $\mathtt{proj2}\ b =_{\beta\eta} \mathtt{id}$.

The linear Church encodings of strings are defined as in Section 5.1.2. This works in the $\lambda\ell\wp$-calculus, with all the expected properties, e.g. the bijection between Σ^* and the terms of type $\mathtt{Str}_\Sigma$ up to $\beta\eta$-equivalence. Proving this is, again, just a matter of examining β-normal η-long forms, thanks to the fact that the $\lambda\ell\wp$-calculus only has negative connectives. Note that while our type of strings – which specializes the encodings of ranked trees – is slightly different from the one in [NP20], the two are interconvertible (one of the directions is similar to what happens in Lemma 7.2.3) so this has no significant effect (cf. [NP20, Remark 5.7]).

Example 7.2.1 ($\lambda\ell\wp$-definable languages)**.** Given two closed $\lambda\ell\wp$-terms t_a and t_b of type $\mathtt{NBool} \multimap \mathtt{NBool}$, one can define the term $g = \lambda s.\, s\ t_a\ t_b\ \mathtt{false} : \mathtt{Str}_{\{a,b\}}[\mathtt{NBool}] \multimap \mathtt{NBool}$. Then for any $w = w[1] \dots w[n] \in \{a, b\}^*$, we have $g\ \overline{w} \longrightarrow_\beta^* t_{w[1]}\ (\dots\ (t_{w[n]}\ \mathtt{false}))$.

- For $t_a = \mathtt{cstt}$ and $t_b = \lambda x.\, x$, g decides the language of words in $\{a, b\}^*$ that contain at least one a; this language is indeed star-free as it can be expressed as $\varnothing^c a \varnothing^c$.
- If negation were definable by a $\lambda\ell\wp$-term $\mathtt{not} : \mathtt{NBool} \multimap \mathtt{NBool}$, then for $t_a = t_b = \mathtt{not}$, the language decided by g would consist of words of odd length: a standard example of regular language that is not star-free. Thus, a corollary of Theorem 1.2.1 is that there is no such term $\mathtt{not}$ such that $\mathtt{not}\ \mathtt{true} =_{\beta\eta} \mathtt{false}$ and $\mathtt{not}\ \mathtt{false} =_{\beta\eta} \mathtt{true}$.

Remark 7.2.2. Actually, the following $\lambda\ell\wp$-term does "define negation":

$$\mathtt{not}' = \lambda b.\, b\ (\lambda g.\, \lambda h.\, g\ (\mathtt{cstt}\ (h\ \mathtt{true})))\ \mathtt{cstf} : \mathtt{NBool}[\mathtt{NBool}] \multimap \mathtt{NBool}$$

(and this is part of the proof that normalization for the planar λ-calculus – i.e. the purely linear fragment of the $\lambda\ell\wp$-calculus – is P-complete [Das+21]).

A point of utmost importance is that because of the heterogeneity of the input and output types, this term does not contradict our above observation and *cannot be iterated by a Church-encoded string*. Monomorphism is therefore crucial for us: if our type system had actual polymorphism, one could give $\mathtt{not}'$ the type $(\forall\alpha.\ \mathtt{NBool}[\alpha]) \multimap (\forall\alpha.\ \mathtt{NBool}[\alpha])$, whose input and output types are equal, and then the words of odd length would be $\lambda\ell\wp$-definable. (This is analogous to Remark 1.3.1 concerning the simply typed λ-calculus.) Yet our ersatz of polymorphism still allows for some form of compositionality (Lemma 7.1.2) that will prove useful in several places in our proof of expressiveness.

In addition to the type $\mathtt{Str}$, we will use a type of almost-strings as a helper to encode star-free languages in the $\lambda\ell\wp$-calculus:

$$\mathtt{StrW}_\Sigma = ((\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o} \multimap \mathbb{o}) \to \underbrace{(\mathbb{o} \multimap \mathbb{o}) \to \cdots \to (\mathbb{o} \multimap \mathbb{o})}_{|\Sigma| \text{ times}} \to \underbrace{(\mathbb{o} \multimap \mathbb{o})}_{\substack{\text{we chose } \multimap \text{ instead of } \to \text{ here} \\ \text{which makes no difference}}}$$

A closed inhabitant of $\mathtt{StrW}$ η-expands into $\lambda^! e.\ \lambda^! f_1.\ \ldots\ \lambda^! f_{|\Sigma|}.\ t$. The idea is that:

- When the context Ψ of non-linear variables contains $f_i, \ldots, f_{|\Sigma|} : \mathbb{o} \multimap \mathbb{o}$, any string $w \in \Sigma^*$ can be represented as the open $\lambda\ell\wp$-term $\Psi; \ldots \vdash f_{w[1]} \circ \cdots \circ f_{w[|w|]} : \mathbb{o} \multimap \mathbb{o}$ in that context, and such strings can even be concatenated by function composition. The point is that this gives us a kind of *purely linear type of strings*.
- The variable $e : (\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o} \multimap \mathbb{o}$ therefore is a function from strings to strings, which one should interpret as a sort of admissible weakening: "erase my argument and return the empty string" (the latter is encoded as $\lambda y.\ y : \mathbb{o} \multimap \mathbb{o}$) – the idea will be to feed an argument for e that will actually have this behavior in a function $\mathtt{StrW}_\Sigma[\tau] \multimap \mathtt{NBool}$, or, in a function $\mathtt{StrW}_\Sigma[\tau] \multimap \mathtt{StrW}_\Pi$, to lift an eraser of type $(\mathbb{o} \multimap \mathbb{o}) \multimap \mathbb{o} \multimap \mathbb{o}$ to an eraser of type $(\tau \multimap \tau) \multimap \tau \multimap \tau$.

$\mathtt{StrW}_\Sigma$ is of course not an adequate type of strings in general, since with the above interpretation of its first argument as an eraser, there are infinitely many terms of this type corresponding to a single string. But it will prove to be a useful trick here.

Lemma 7.2.3. *There is a type-cast function* $\mathtt{Str}_\Sigma[\mathbb{o} \multimap \mathbb{o}] \multimap \mathtt{StrW}_\Sigma$.

PROOF. $\lambda s.\ \lambda^! e.\ \lambda^! f_1.\ \ldots \lambda^! f_{|\Sigma|}.\ s\ (\lambda g.\ f_1 \circ g)\ \ldots\ (\lambda g.\ f_{|\Sigma|} \circ g)\ (\lambda x.\ x)$. □

7.3. EXPRESSIVENESS OF THE $\lambda\ell\wp$-CALCULUS

Our goal is to construct, for any star-free language, a closed $\lambda\ell\wp$-term that defines this language, of type $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{NBool}$ for some purely linear τ. To do so, we take a detour through transducers; more precisely, we use *aperiodic sequential functions*, cf. Definition 2.2.1.

Theorem 7.3.1. *Any aperiodic sequential function* $\Sigma^* \to \Gamma^*$ *can be "expressed" (via the informal interpretation of* $\mathtt{StrW}$ *above) by a closed* $\lambda\ell\wp$*-term of type* $\mathtt{StrW}_\Sigma[\tau] \multimap \mathtt{StrW}_\Gamma$ *for some* purely linear *type* τ.

Remark 7.3.2. We diverge from [NP20] by working with $\mathtt{StrW}_\Sigma$ here! Trying to characterize string-to-string function classes (first-order transductions?) using $\mathtt{StrW}_\Sigma[\tau] \multimap \mathtt{StrW}_\Gamma$ would be rather abusive: terms of this type are not even guaranteed a priori to give, for two representations of the same string, two outputs that also represent a common string (but we will build terms for which this property holds). On the other hand, the $\lambda\ell\wp$-calculus, which embeds into the strictly linear $\lambda\ell$-calculus, does not allow us to do much with the type $\mathtt{Str}_\Sigma[\tau] \multimap \mathtt{Str}_\Gamma$ according to Theorem 7.0.2.

The advantage of working with aperiodic sequential functions is that they can be assembled from small "building blocks" by function composition, as the *Krohn–Rhodes*

decomposition (Theorem 2.2.4) tells us. Our proof strategy for the above theorem will consist in encoding these blocks (Lemma 7.3.6) and composing them together (Lemma 7.1.2). To deduce the desired result, we rely on two lemmas:

Lemma 7.3.3 (classical)**.** *If a language $L \subseteq \Sigma^*$ is* star-free*, then its* (string-valued) indicator function $\chi_L : \Sigma^* \to \{1\}^*$*, defined by $\chi_L(w) = 1$ if $w \in L$ and $\chi_L(w) = \varepsilon$ otherwise, is aperiodic sequential.*

Remark 7.3.4. The converse 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.

Lemma 7.3.5. *There exists a $\lambda\ell\wp$-term* $\mathtt{nonempty} : \mathtt{StrW}_{\{1\}}[\mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}]] \multimap \mathtt{NBool}$ *that tests whether its input string is non-empty.*

Proof. This is less obvious than the corresponding [NP20, Lemma 4.3]: we use

- the "constant to $\mathtt{true}$" function, already seen previously – although its principal type is $\mathtt{NBool} \multimap \mathtt{NBool}$, we shall instantiate it here to $\mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}] \multimap \mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}]$;
- an "eraser" $\mathtt{erb} : (\mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}] \multimap \mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}]) \multimap (\mathtt{NBool}[\mathbb{o} \multimap \mathbb{o}] \multimap \mathtt{NBool}[o \multimap o])$ defined by $\mathtt{erb} = \lambda g.\, \lambda b.\, \lambda k.\, \lambda f.\, \lambda x.\, \mathtt{proj2}\ (g\ \mathtt{true})\ (b\ k\ f\ x)$;

Then the $\lambda\ell\wp$-term $\mathtt{nonempty} = \lambda s.\, s\ \mathtt{erb}\ \mathtt{cstt}\ \mathtt{false}$ works. □

Let L be a star-free language. Combining Lemma 7.3.3 and Theorem 7.3.1, χ_L is definable by some $\lambda\ell\wp$-term $\mathtt{indic}_L : \mathtt{StrW}_\Sigma[\tau] \multimap \mathtt{StrW}_{\{1\}}$ where τ is purely linear. To compose this with the non-emptiness test of Lemma 7.3.5 and the type-cast of Lemma 7.2.3, we use Lemma 7.1.2 again to get a $\lambda\ell\wp$-term of type $\mathtt{Str}_\Sigma[\tau[\mathtt{NBool}]] \multimap \mathtt{NBool}$ defining L; and since τ and $\mathtt{NBool}$ are purely linear, so is $\tau[\mathtt{NBool}]$.

7.3.1. **Encoding aperiodic sequential transducers.** Thanks to the Krohn–Rhodes decomposition (§2.2.1) and to Lemma 7.1.2, the following entails Theorem 7.3.1, thus concluding our proof that all star-free languages are $\lambda\ell\wp$-definable

Lemma 7.3.6. *Any function $\Sigma^* \to \Gamma^*$ computed by some aperiodic sequential transducer* with 2 states *can be "expressed" by some $\lambda\ell\wp$-term of type $\mathtt{StrW}_\Sigma[\tau] \multimap \mathtt{StrW}_\Gamma$, for a* purely linear *type τ depending on the function.*

Let us start by exposing the rough idea of the encoding's trick using set-theoretic maps. **This part is the same as in [NP20].** We reuse the notations of Definition 2.2.1 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 $w \in \Gamma^*$. We can represent this "history" by the pair (κ_w, ζ) (resp. (ζ, κ_w)) where

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

For instance, in the case of Example 2.2.3, after reading a string $s = s'b$, the transducer is in the state q_b and has outputted[2] $w = a \cdot \psi(s')$, which we represent as $(\zeta, \kappa_{a\cdot\psi(s')})$ (taking $q_a = 1$ and $q_b = 2$; ψ is described in Example 2.2.3). In general, some key observations are

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

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

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

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

Coming back to Example 2.2.3, let us say that after the transducer has read a prefix $s = s'b$ of its input string as we previously described, the next letter is a. Then the expression $h(\delta_{\mathrm{out}}(1,c))g(\delta_{\mathrm{out}}(2,c))$ above is in this case $\zeta(a)\kappa_{a\cdot\psi(s')}(bb) = \varepsilon\cdot a\cdot\psi(s')\cdot bb = a\cdot\psi(s)$ which is indeed the output that the transducer produces after reading the input prefix $sa = s'ba$.

Next, we must transpose these ideas to the setting of the $\lambda\ell\wp$-calculus. **This is where new ideas compared to [NP20] come into play.**

Proof of Lemma 7.3.6. We define the $\lambda\ell\wp$-term meant to compute our sequential function as

$$\lambda s.\ \lambda^! e.\ \lambda^! f_{a_1}.\ \ldots\ \lambda^! f_{a_{|\Gamma|}}.\ \mathtt{out}\ (s\ \mathtt{elift}\ \mathtt{trans}_{c_1}\ \ldots\ \mathtt{trans}_{c_{|\Sigma|}}) : \mathtt{Str}_\Sigma[\tau] \multimap \mathtt{Str}_\Gamma$$

where $\Sigma = \{c_1, \ldots, c_{|\Sigma|}\}$, $\Gamma = \{a_1, \ldots, a_{|\Gamma|}\}$ and, writing $\Psi = \{f_a : \mathbb{0} \multimap \mathbb{0} \mid a \in \Gamma\} \cup \{e : T\}$ (with $T = (\mathbb{0} \multimap \mathbb{0}) \multimap \mathbb{0} \multimap \mathbb{0}$ as we shall see),

$$\Psi;\varnothing \vdash \mathtt{trans}_c : \tau \multimap \tau \quad (\text{for all } c \in \Sigma) \qquad \Psi;\varnothing \vdash \mathtt{out} : (\tau \multimap \tau) \multimap (\mathbb{0} \multimap \mathbb{0})$$

$$\textbf{(new!)} \quad \Psi;\varnothing \vdash \mathtt{elift} : (\tau \multimap \tau) \multimap (\tau \multimap \tau)$$

In the presence of this non-linear context Ψ, the type $S = \mathbb{0} \multimap \mathbb{0}$ morally serves as a purely linear type of strings, as mentioned before. 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-linearly the variables f_a for $a \in \Gamma$.

We shall take the type τ, at which the input $\mathtt{Str}_\Sigma$ is instantiated, to be $\tau = T \multimap T \multimap S$, which is indeed purely linear 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 $\lambda\ell\wp$-calculus has no actual $\otimes$ connective). Without further ado, let us program; **the key new cases to handle strict linearity (as opposed to affineness) are given by**:

- to implement ζ, we just use the "eraser" that the $\mathtt{StrW}_\Gamma$ takes as the argument e:

$$\mathtt{zeta} = e : (\mathbb{0} \multimap \mathbb{0}) \multimap \mathbb{0} \multimap \mathbb{0} = S \multimap S = T$$

- the new term to implement is the eraser for $\mathtt{StrW}_\Sigma[\tau]$ using the e given to $\mathtt{StrW}_\Gamma$:

$$\mathtt{elift} = \lambda g.\ \lambda h.\ \lambda l.\ \lambda l'.\ \lambda x.\ e\ \mathtt{junk}\ (h\ l\ l'\ x)$$

(it would become a fully η-expanded identity function if $\lambda g.$ and $(e\ \mathtt{junk})$ were removed (note that $h\ l\ l'\ x : o$)), where one can take for $\mathtt{junk}$ any expression of type $S = \mathbb{0} \multimap \mathbb{0}$ using $g : \tau \multimap \tau$ exactly once e.g.

$$\mathtt{junk} = g\ (\lambda j.\ \lambda j'.\ j\ (j'\ (\lambda y.\ y)))\ (\lambda z.\ z)\ (\lambda z.\ z)$$

The remaining cases were already strictly linear in [NP20], so are left unchanged:

- $\mathtt{cat} = \lambda w.\ \lambda w'.\ \lambda x.\ w\ (w'\ x) : S \multimap S \multimap \mathbb{0} \multimap \mathbb{0} = S \multimap S \multimap S = S \multimap T$ plays the roles of both the concatenation operator and of $w \mapsto \kappa_w$ (thanks to currying)
- $u_q : \mathbb{0} \multimap \mathbb{0}$ is the representation of the output word $\delta_{\mathrm{out}}(q,c)$ that corresponds to a given input letter $c \in \Sigma$ and state $q \in Q = \{1,2\}$
- case $\delta_{\mathrm{st}}(q,c) = q$: $\mathtt{trans}_c = \lambda k.\ \lambda h.\ \lambda g.\ k\ (\lambda y.\ h\ (\mathtt{cat}\ u_1\ y))\ (\lambda z.\ g\ (\mathtt{cat}\ u_2\ z))$
  (if we wanted to handle the excluded case $\delta_{\mathrm{st}}(q,c) = 3-q$, we would write a similar term with the occurrences of h and g exchanged $(\lambda k.\ \lambda h.\ \lambda g.\ k\ (\lambda y.\ g\ \ldots)\ (\lambda z.\ h\ \ldots))$, violating the non-commutativity requirement)

- case $\delta_{\mathrm{st}}(q,c) = 1$: $\mathtt{trans}_c = \lambda k.\, \lambda h.\, \lambda g.\, k\,(\mathtt{cat}\,(\mathtt{cat}\,(h\,u_1)\,(g\,u_2)))\,\mathtt{zeta}$
- case $\delta_{\mathrm{st}}(q,c) = 2$: $\mathtt{trans}_c = \lambda k.\, \lambda h.\, \lambda g.\, k\,\mathtt{zeta}\,(\mathtt{cat}\,(\mathtt{cat}\,(h\,u_1)\,(g\,u_2)))$
- $\mathtt{out} = \lambda j.\, j\,(\lambda h.\, \lambda g.\, \mathtt{cat}\,(h\,v_1)\,(g\,v_2))\,(\lambda x.\, x)\,\mathtt{zeta}$, where v_q represents the output suffix $F(q)$ for state $q \in \{1,2\}$, assuming w.l.o.g. that the initial state is 1 (also, here $\lambda x.\, x$ represents κ_ε since the latter is the identity on Γ^*)

We leave it to the reader to check that these $\lambda\ell_{\wp}$-terms have the expected computational behavior. □

This concludes our proof of the $\lambda\ell_{\wp}$-definability of all star-free languages. As we said before, we refer to [NP20] for a strengthened form of the converse, that applies to the affine extension of the $\lambda\ell_{\wp}$-calculus.

> You're still here? Oh, I guess I should tell you math papers generally don't have what you or I might call a "conclusion". They just sort of stop.
> So, yeah, you can, um, go now. But, cheers!
>
> Piper H [H16, p. 127]

Bibliography

[Aar17] Scott Aaronson. "P=?NP." In: *Electronic Colloquium on Computational Complexity* 24 (2017), p. 4. URL: https://eccc.weizmann.ac.il/report/2017/004 (cit. on pp. 8, 35).

[AB18] Clément Aubert and Marc Bagnol. "Unification and Logarithmic Space." In: *Logical Methods in Computer Science* 14.3 (2018). DOI: 10.23638/LMCS-14(3:6)2018 (cit. on p. 35).

[Abe13] Andreas Abel. "Normalization by Evaluation: Dependent Types and Impredicativity." Habilitationsschrift. Ludwig-Maximilians-University Munich, May 2013. URL: http://www.cse.chalmers.se/~abela/habil.pdf (cit. on p. 18).

[Abr03] Samson Abramsky. "Sequentiality vs. Concurrency In Games And Logic." In: *Mathematical Structures in Computer Science* 13.4 (2003), pp. 531–565. DOI: 10.1017/S0960129503003980 (cit. on p. 49).

[Abr07] Samson Abramsky. "Temperley–Lieb Algebra: From Knot Theory to Logic and Computation via Quantum Mechanics." en. In: *Mathematics of Quantum Computation and Quantum Technology*. Ed. by Goong Chen, Louis Kauffman, and Samuel Lomonaco. Vol. 20074453. Chapman and Hall/CRC, Sept. 2007, pp. 515–558. DOI: 10.1201/9781584889007.ch15 (cit. on pp. 16, 25, 44).

[Abr20] Samson Abramsky. "Whither semantics?" In: *Theoretical Computer Science* 807 (2020). In the special issue of TCS in honor of Maurice Nivat, pp. 3–14. DOI: 10.1016/j.tcs.2019.06.029 (cit. on p. 36).

[Abr96] Samson Abramsky. "Retracing Some Paths in Process Algebra." In: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*. Ed. by Ugo Montanari and Vladimiro Sassone. Vol. 1119. Lecture Notes in Computer Science. Springer, 1996, pp. 1–17. DOI: 10.1007/3-540-61604-7_44 (cit. on p. 44).

[ABS16] Clément Aubert, Marc Bagnol, and Thomas Seiller. "Unary Resolution: Characterizing Ptime." In: *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Bart Jacobs and Christof Löding. Vol. 9634. Lecture Notes in Computer Science. Springer, 2016, pp. 373–389. DOI: 10.1007/978-3-662-49630-5_22 (cit. on p. 35).

[AČ10] 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, December 15-18, 2010, Chennai, India*. Ed. by Kamal Lodaya and Meena Mahajan. Vol. 8. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 1–12. DOI: 10.4230/LIPIcs.FSTTCS.2010.1 (cit. on pp. 15, 56–58, 74, 133).

[AD11] Rajeev Alur and Jyotirmoy V. Deshmukh. "Nondeterministic Streaming String Transducers." In: *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. Ed. by Luca Aceto, Monika Henzinger, and Jirí Sgall. Vol. 6756. Lecture Notes in Computer Science. Springer, 2011, pp. 1–20. DOI: 10.1007/978-3-642-22012-8_1 (cit. on pp. 28, 112–114, 133).

[AD17] Rajeev Alur and Loris D'Antoni. "Streaming Tree Transducers." en. In: *Journal of the ACM* 64.5 (Aug. 2017), pp. 1–55. ISSN: 00045411. DOI: 10.1145/3092842 (cit. on pp. 26, 45, 67–69, 170, 173, 177, 179–181).

[ADV21] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. *Interacting Seems Unreasonable, in Time and Space*. Extended abstract of a talk given at the ITRS'21 workshop. 2021. URL: http://www.di.unito.it/~deligu/ITRS2021/ITRS_2021_paper_3.pdf (cit. on p. 45).

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. "Regular combinators for string transformations." en. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) - CSL-LICS '14*. Vienna, Austria: ACM Press, 2014, pp. 1–10. ISBN: 978-1-4503-2886-9. DOI: 10.1145/2603088.2603151 (cit. on p. 36).

[AFT12] Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. "Regular Transformations of Infinite Strings." en. In: *2012 27th Annual IEEE Symposium on Logic in Computer Science*. Dubrovnik, Croatia: IEEE, June 2012, pp. 65–74. DOI: 10.1109/LICS.2012.18 (cit. on pp. 56, 59).

[AJP21] Antoine Amarilli, Louis Jachiet, and Charles Paperman. "Dynamic Membership for Regular Languages." In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. Ed. by Nikhil Bansal, Emanuela Merelli, and James Worrell. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 116:1–116:17. DOI: 10.4230/LIPIcs.ICALP.2021.116 (cit. on p. 11).

[AL16] Beniamino Accattoli and Ugo Dal Lago. "(Leftmost-Outermost) Beta Reduction is Invariant, Indeed." In: *Logical Methods in Computer Science* 12.1 (2016). DOI: 10.2168/LMCS-12(1:4)2016 (cit. on pp. 8, 16).

[Alc19] Aurore Alcolei. "Enriched concurrent games: witnesses for proofs and resource analysis." PhD thesis. École normale supérieure de Lyon, France, Oct. 2019. URL: https://tel.archives-ouvertes.fr/tel-02448974 (cit. on p. 22).

[Alt+01] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. "Normalization by Evaluation for Typed Lambda Calculus with Coproducts." In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001, pp. 303–310. DOI: 10.1109/LICS.2001.932506 (cit. on p. 140).

[ALV21] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. "The (In)Efficiency of interaction." In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–33. DOI: 10.1145/3434332 (cit. on p. 47).

[And92] Jean-Marc Andreoli. "Logic Programming with Focusing Proofs in Linear Logic." In: *Journal of Logic and Computation* 2.3 (1992), pp. 297–347. DOI: 10.1093/logcom/2.3.297 (cit. on p. 37).

[AP21] Antoine Amarilli and Charles Paperman. "Locality and Centrality: The Variety ZG." In: *CoRR* abs/2102.07724 (2021). arXiv: 2102.07724 (cit. on pp. 11, 22).

[APR05] Jean-Marc Andreoli, Gabriele Pulcini, and Paul Ruet. "Permutative Logic." In: *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*. Ed. by C.-H. Luke Ong. Vol. 3634. Lecture Notes in Computer Science. Springer, 2005, pp. 184–199. DOI: 10.1007/11538363_14 (cit. on p. 16).

[AS16a] Clément Aubert and Thomas Seiller. "Characterizing co-NL by a group action." en. In: *Mathematical Structures in Computer Science* 26.4 (May 2016), pp. 606–638. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129514000267 (cit. on p. 35).

[AS16b] Clément Aubert and Thomas Seiller. "Logarithmic Space and Permutations." In: *Information and Computation* 248 (2016), pp. 2–21. DOI: 10.1016/j.ic.2014.01.018 (cit. on p. 35).

[AS21] Samson Abramsky and Nihil Shah. "Relating structure and power: Comonadic semantics for computational resources." In: *Journal of Logic and Computation* 31.6 (2021). Long version of a CSL'18 paper, pp. 1390–1428. DOI: 10.1093/logcom/exab048 (cit. on p. 36).

[Aub+14] Clément Aubert, Marc Bagnol, Paolo Pistone, and Thomas Seiller. "Logic Programming and Logarithmic Space." In: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. Ed. by Jacques Garrigue. Vol. 8858. Lecture Notes in Computer Science. Springer, 2014, pp. 39–57. DOI: 10.1007/978-3-319-12736-1_3 (cit. on p. 35).

[Aub15] Clément Aubert. "An in-between 'implicit' and 'explicit' complexity: Automata." In: *DICE 2015 – Developments in Implicit Computational Complexity*. London, United Kingdom, Apr. 2015. URL: https://hal.archives-ouvertes.fr/hal-01111737 (cit. on p. 9).

[Avr14] Arnon Avron. "What is relevance logic?" In: *Annals of Pure and Applied Logic* 165.1 (2014), pp. 26–48. DOI: 10.1016/j.apal.2013.07.004 (cit. on p. 22).

[Bai15] Patrick Baillot. "On the expressivity of elementary linear logic: Characterizing Ptime and an exponential time hierarchy." In: *Information and Computation* 241 (Apr. 2015), pp. 3–31. ISSN: 0890-5401. DOI: 10.1016/j.ic.2014.10.005 (cit. on p. 48).

[Bak92] Henry G. Baker. "Lively linear Lisp: 'look ma, no garbage!'" In: *ACM SIGPLAN Notices* 27.8 (1992), pp. 89–98. DOI: 10.1145/142137.142162 (cit. on p. 16).

[Bar96] Andrew Barber. *Dual Intuitionistic Linear Logic*. en. 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/ (cit. on pp. 26, 191).

[BB85] Corrado Böhm and Alessandro Berarducci. "Automatic synthesis of typed Λ-programs on term algebras." In: *Theoretical Computer Science*. Third Conference on Foundations of Software Technology and Theoretical Computer Science 39 (Jan. 1985), pp. 135–154. ISSN: 0304-3975. DOI: 10.1016/0304-3975(85)90135-5 (cit. on pp. 17, 36).

[BC08] Mikołaj Bojańczyk and Thomas Colcombet. "Tree-walking automata do not recognize all regular languages." In: *SIAM Journal on Computing* 38.2 (2008), pp. 658–701. DOI: 10.1137/050645427 (cit. on p. 43).

[BC18] Mikołaj Bojańczyk and Wojciech Czerwiński. "An automata toolbox." Lecture notes for a course at the University of Warsaw (version of February 6, 2018). 2018. URL: https://www.mimuw.edu.pl/~bojan/paper/automata-toolbox-book (cit. on pp. 28, 77, 133).

[BC82] Gérard Berry and Pierre-Louis Curien. "Sequential Algorithms on Concrete Data Structures." In: *Theoretical Computer Science* 20 (1982), pp. 265–321. DOI: 10.1016/S0304-3975(82)80002-9 (cit. on p. 20).

[BC89] David A. Mix Barrington and James C. Corbett. "On the Relative Complexity of Some Languages in NC." In: *Information Processing Letters* 32.5 (1989), pp. 251–256. DOI: 10.1016/0020-0190(89)90052-5 (cit. on p. 13).

[BC92] Stephen Bellantoni and Stephen A. Cook. "A New Recursion-Theoretic Characterization of the Polytime Functions." In: *Computational Complexity* 2 (1992), pp. 97–110. DOI: 10.1007/BF01201998 (cit. on p. 33).

[BCL20] Boaz Barak, Raphaëlle Crubillé, and Ugo Dal Lago. "On Higher-Order Cryptography." In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by Artur Czumaj, Anuj Dawar, and Emanuela Merelli. Vol. 168. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 108:1–108:16. DOI: 10.4230/LIPIcs.ICALP.2020.108 (cit. on p. 34).

[BD20] Mikołaj Bojańczyk and Amina Doumane. "First-order tree-to-tree functions." In: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany (online conference), July 8-11, 2020*. Ed. by Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller. ACM, 2020, pp. 252–265. DOI: 10.1145/3373718.3394785 (cit. on pp. 15, 36, 53, 67).

[BDK18] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. "Regular and First-Order List Functions." en. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*. Oxford, United Kingdom: ACM Press, 2018, pp. 125–134. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209163 (cit. on pp. 36, 43, 67).

[BDR18] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. "Characterizing polynomial and exponential complexity classes in elementary lambda-calculus." In: *Information and Computation*. Developments in Implicit Computational Complexity (DICE) 2014 and 2015 261 (Aug. 2018), pp. 55–77. ISSN: 0890-5401. DOI: 10.1016/j.ic.2018.05.005 (cit. on pp. 48, 49).

[BE00] Roderick Bloem and Joost Engelfriet. "A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars." In: *Journal of Computer and System Sciences* 61.1 (Aug. 2000), pp. 1–50. ISSN: 0022-0000. DOI: 10.1006/jcss.1999.1684 (cit. on pp. 15, 67).

[Ben+17] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. "Polynomial automata: Zeroness and applications." en. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Reykjavik, Iceland: IEEE, June 2017, pp. 1–12. ISBN: 978-1-5090-3018-7. DOI: 10.1109/LICS.2017.8005101 (cit. on pp. 62, 104).

[Ber+18] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. "Linear Haskell: practical linearity in a higher-order polymorphic language." In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), 5:1–5:29. DOI: 10.1145/3158093 (cit. on p. 16).

[Ber81] Gérard Berry. "On the Definition of Lambda-Calculus Models." In: *Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, Proceedings*. Ed. by Josep Díaz and Isidro Ramos. Vol. 107. Lecture Notes in Computer Science. Springer, 1981, pp. 218–230. DOI: 10.1007/3-540-10699-5_99 (cit. on p. 20).

[Bes+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." In: *Communications of the ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374 (cit. on p. 10).

[BGS99] Andreas Blass, Yuri Gurevich, and Saharon Shelah. "Choiceless Polynomial Time." In: *Annals of Pure and Applied Logic* 100.1-3 (1999), pp. 141–187. DOI: 10.1016/S0168-0072(99)00005-6 (cit. on p. 34).

[BH21] Vasco Brattka and Peter Hertling, eds. *Handbook of Computability and Complexity in Analysis*. Theory and Applications of Computability. Springer, 2021. ISBN: 978-3-030-59233-2. DOI: 10.1007/978-3-030-59234-9 (cit. on p. 35).

[Bie94] Gavin M. Bierman. *On Intuitionistic Linear Logic*. en. Technical report UCAM-CL-TR-346. University of Cambridge, Computer Laboratory, Aug. 1994. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.588&rep=rep1&type=pdf (cit. on p. 148).

[Bir89] Jean-Camille Birget. "Concatenation of Inputs in a Two-Way Automaton." In: *Theoretical Computer Science* 63.2 (1989), pp. 141–156. DOI: 10.1016/0304-3975(89)90075-3 (cit. on p. 10).

[BKL19] Mikołaj Bojańczyk, Sandra Kiefer, and Nathan Lhote. "String-to-String Interpretations With Polynomial-Size Output." In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Ed. by Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 106:1–106:14. ISBN: 978-3-95977-109-2. DOI: 10.4230/LIPIcs.ICALP.2019.106 (cit. on p. 32).

[BKS21] Mikołaj Bojańczyk, Bartek Klin, and Julian Salamanca. *Monadic monadic second-order logic*. In preparation. Talk recording available at https://youtu.be/gcQB2GtxsmE. 2021 (cit. on p. 31).

[BM10] Patrick Baillot and Damiano Mazza. "Linear Logic by Levels and Bounded Time Complexity." In: *Theoretical Computer Science* 411.2 (Jan. 2010), pp. 470–503. ISSN: 03043975. DOI: 10.1016/j.tcs.2009.09.015 (cit. on p. 40).

[BO09] William Blum and C.-H. Luke Ong. "The Safe Lambda Calculus." en. In: *Logical Methods in Computer Science* 5.1 (Feb. 2009). DOI: 10.2168/LMCS-5(1:3)2009 (cit. on p. 41).

[Boe20] Menno de Boer. *A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory*. 2020. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-181640 (cit. on p. 49).

[Boj+21] Mikołaj Bojańczyk, Gaëtan Douéneau-Tabot, Sandra Kiefer, Lê Thành Dũng Nguyễn, and Pierre Pradic. "On the growth rate of polyregular functions." In preparation. 2021 (cit. on pp. 48, 71).

[Boj18] Mikołaj Bojańczyk. "Polyregular Functions." In: *CoRR* abs/1810.08760 (Oct. 2018). arXiv: 1810.08760 (cit. on pp. 12, 28, 36, 44, 53, 57, 64–66, 72, 74, 81).

[Boj19] Mikołaj Bojańczyk. "Slightly Infinite Sets." 2019. URL: https://www.mimuw.edu.pl/~bojan/paper/atom-book (cit. on p. 45).

[BP16a] Mikołaj Bojańczyk and Michał Pilipczuk. "Definability equals recognizability for graphs of bounded treewidth." In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. Ed. by Martin Grohe, Eric Koskinen, and Natarajan Shankar. ACM, 2016, pp. 407–416. DOI: 10.1145/2933575.2934508 (cit. on p. 32).

[BP16b] Matt Brown and Jens Palsberg. "Breaking through the normalization barrier: a self-interpreter for F-omega." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 5–17. DOI: 10.1145/2837614.2837623 (cit. on p. 15).

[BPT92] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. "Automatic Generation of Linear-Time Algorithms from Predicate Calculus Descriptions of Problems on Recursively Constructed Graph Families." In: *Algorithmica* 7.5&6 (1992), pp. 555–581. DOI: 10.1007/BF01758777 (cit. on p. 32).

[BR10] Jean Berstel and Christophe Reutenauer. *Noncommutative Rational Series with Applications*. Vol. 137. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Oct. 2010, p. 248 (cit. on pp. 59, 94).

[Bro+21] Christopher H. Broadbent, Arnaud Carayol, C.-H. Luke Ong, and Olivier Serre. "Higher-order Recursion Schemes and Collapsible Pushdown Automata: Logical Properties." In: *ACM Transactions on Computational Logic* 22.2 (2021), 12:1–12:37. DOI: 10.1145/3452917 (cit. on p. 38).

[BS20] Mikołaj Bojańczyk and Rafał Stefański. "Single-Use Automata and Transducers for Infinite Alphabets." In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by Artur Czumaj, Anuj Dawar, and Emanuela Merelli. Vol. 168. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 113:1–113:14. DOI: 10.4230/LIPIcs.ICALP.2020.113 (cit. on pp. 15, 43, 45, 46).

[Büc60] J. Richard Büchi. "Weak Second-Order Arithmetic and Finite Automata." In: *Mathematical Logic Quarterly* 6.1--6 (1960), pp. 66–92. DOI: 10.1002/malq.19600060105 (cit. on p. 31).

[Büc62] J. Richard Büchi. "Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic." In: *Logic, Methodology and Philosophy of Science: Proceedings of the 1960 International Congress*. Ed. by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Stanford University Press, 1962, pp. 1–11. DOI: 10.1016/S0049-237X(09)70564-6 (cit. on p. 31).

[Bul20] Andrei A. Bulatov. *A dichotomy theorem for nonuniform CSPs simplified*. Updated and improved version of a FOCS'17 paper. 2020. arXiv: 2007.09099 [cs.CC] (cit. on p. 35).

[Bul21] Andrei A. Bulatov. "Symmetries and Complexity (Invited Talk)." In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021,*

Glasgow, Scotland (Virtual Conference). Ed. by Nikhil Bansal, Emanuela Merelli, and James Worrell. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 2:1–2:17. DOI: 10.4230/LIPIcs.ICALP.2021.2 (cit. on p. 35).

[Cad+21] Michaël Cadilhac, Filip Mazowiecki, Charles Paperman, Michał Pilipczuk, and Géraud Sénizergues. "On Polynomial Recursive Sequences." In: *Theory of Computing Systems* (June 2021). DOI: 10.1007/s00224-021-10046-9 (cit. on pp. 41, 61).

[Cam11] Peter J. Cameron. *Aftermath*. 2011. arXiv: 1111.4050 [math.HO] (cit. on p. 19).

[CC21] Simon Castellan and Pierre Clairambault. *Disentangling Parallelism and Interference in Game Semantics*. 2021. arXiv: 2103.15453 [cs.LO] (cit. on p. 20).

[CD15] Olivier Carton and Luc Dartois. "Aperiodic Two-way Transducers and FO-Transductions." In: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*. Ed. by Stephan Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 160–174. DOI: 10.4230/LIPIcs.CSL.2015.160 (cit. on p. 32).

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic. A language-theoretic approach*. Encyclopedia of Mathematics and its applications, Vol. 138. Collection Encyclopedia of Mathematics and Applications, Vol. 138. Cambridge University Press, June 2012, p. 728. URL: https://hal.archives-ouvertes.fr/hal-00646514 (cit. on p. 32).

[CFI92] Jin-yi Cai, Martin Fürer, and Neil Immerman. "An optimal lower bound on the number of variables for graph identifications." In: *Combinatorica* 12.4 (1992). Long version of a FOCS'89 paper, pp. 389–410. DOI: 10.1007/BF01305232 (cit. on p. 34).

[CGM17] Pierre Clairambault, Charles Grellois, and Andrzej Murawski. "Linearity in Higher-order Recursion Schemes." In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), 39:1–39:29. ISSN: 2475-1421. DOI: 10.1145/3158127 (cit. on p. 38).

[CH00] Pierre-Louis Curien and Hugo Herbelin. "The duality of computation." In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. Ed. by Martin Odersky and Philip Wadler. ACM, 2000, pp. 233–243. DOI: 10.1145/351240.351262 (cit. on p. 22).

[Cho17] Christian Choffrut. "Sequences of words defined by two-way transducers." In: *Theoretical Computer Science* 658 (2017), pp. 85–96. DOI: 10.1016/j.tcs.2016.05.004 (cit. on p. 91).

[Cla+20] Bryce Clarke, Derek Elkins, Jeremy Gibbons, Fosco Loregian, Bartosz Milewski, Emily Pillmore, and Mario Román. *Profunctor optics, a categorical update*. 2020. arXiv: 2001.07488 [cs.PL] (cit. on p. 22).

[CLP15] Thomas Colcombet, Clemens Ley, and Gabriele Puppis. "Logics with rigidly guarded data tests." In: *Logical Methods in Computer Science* 11.3 (2015). DOI: 10.2168/LMCS-11(3:10)2015 (cit. on pp. 45, 46).

[CM19] Pierre Clairambault and Andrzej S. Murawski. "On the Expressivity of Linear Recursion Schemes." In: *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Ed. by Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen. Vol. 138. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 50:1–50:14. ISBN: 978-3-95977-117-7. DOI: 10.4230/LIPIcs.MFCS.2019.50 (cit. on pp. 38, 44).

[Cor04] Leo Corry. *Modern Algebra and the Rise of Mathematical Structures*. 2nd. Birkhäuser Verlag, 2004. ISBN: 978-3-0348-7917-0. DOI: 10.1007/978-3-0348-7917-0 (cit. on p. 20).

[Cou88] Bruno Courcelle. "The Monadic Second-Order Logic of Graphs: Definable Sets of Finite Graphs." In: *Graph-Theoretic Concepts in Computer Science, 14th International Workshop, WG '88, Amsterdam, The Netherlands, June 15-17, 1988, Proceedings*. Ed. by Jan van Leeuwen. Vol. 344. Lecture Notes in Computer Science. Springer, 1988, pp. 30–53. DOI: 10.1007/3-540-50728-0_34 (cit. on p. 32).

[CP17a] Thomas Colcombet and Daniela Petrişan. "Automata and minimization." In: *ACM SIGLOG News* 4.2 (May 2017), pp. 4–27. DOI: 10.1145/3090064.3090066 (cit. on p. 23).

[CP17b] Thomas Colcombet and Daniela Petrişan. "Automata in the Category of Glued Vector Spaces." In: *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*. Ed. by Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin. Vol. 83. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 52:1–52:14. ISBN: 978-3-95977-046-0. DOI: 10.4230/LIPIcs.MFCS.2017.52 (cit. on pp. 23, 27).

[CP20] Thomas Colcombet and Daniela Petrişan. "Automata Minimization: a Functorial Approach." en. In: *Logical Methods in Computer Science* 16.1 (Mar. 2020). DOI: 10.23638/LMCS-16(1:32)2020 (cit. on pp. 23, 103, 104).

[CPS21] Thomas Colcombet, Daniela Petrişan, and Riccardo Stabile. "Learning Automata and Transducers: A Categorical Approach." In: *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*. Ed. by Christel Baier and Jean Goubault-Larrecq. Vol. 183. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 15:1–15:17. DOI: 10.4230/LIPIcs.CSL.2021.15 (cit. on p. 23).

[CVV21] Kostia Chardonnet, Benoît Valiron, and Renaud Vilmart. "Geometry of Interaction for ZX-Diagrams." In: *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*. Ed. by Filippo Bonchi and Simon J. Puglisi. Vol. 202. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 30:1–30:16. DOI: 10.4230/LIPIcs.MFCS.2021.30 (cit. on p. 44).

[Dam82] Werner Damm. "The IO- and OI-Hierarchies." In: *Theoretical Computer Science* 20 (1982), pp. 95–207. DOI: 10.1016/0304-3975(82)90009-3 (cit. on p. 42).

[Dar+17] Luc Dartois, Paulin Fournier, Ismaël Jecker, and Nathan Lhote. "On Reversible Transducers." In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 113:1–113:12. DOI: 10.4230/LIPIcs.ICALP.2017.113 (cit. on p. 44).

[Das+21] Anupam Das, Damiano Mazza, Lê Thành Dũng Nguyễn, and Noam Zeilberger. *Complexity of normalization for subsystems of untyped linear lambda calculus*. In preparation. Slides available at http://noamz.org/talks/smp.2021.06.28.pdf. 2021 (cit. on pp. 25, 192).

[Das20a] Anupam Das. "A circular version of Gödel's T and its abstraction complexity." In: *CoRR* abs/2012.14421 (2020). arXiv: 2012.14421 (cit. on p. 37).

[Das20b] Anupam Das. "On the logical complexity of cyclic arithmetic." In: *Logical Methods in Computer Science* 16.1 (2020). DOI: 10.23638/LMCS-16(1:1)2020 (cit. on p. 37).

[Dav+20] Vrunda Dave, Emmanuel Filiot, Shankara Narayanan Krishna, and Nathan Lhote. "Synthesis of Computable Regular Functions of Infinite Words." In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 43:1–43:17. DOI: 10.4230/LIPIcs.CONCUR.2020.43 (cit. on p. 13).

[Daw20] Anuj Dawar. "Symmetric Computation (Invited Talk)." In: *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*. Ed. by Maribel Fernández and Anca Muscholl. Vol. 152. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 2:1–2:12. DOI: 10.4230/LIPIcs.CSL.2020.2 (cit. on p. 34).

[DFG20] Gaëtan Douéneau-Tabot, Emmanuel Filiot, and Paul Gastin. "Register Transducers Are Marble Transducers." In: *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*. Ed. by Javier Esparza and Daniel Kráľ. Vol. 170. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss

Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 29:1–29:14. ISBN: 978-3-95977-159-7. DOI: 10.4230/LIPIcs.MFCS.2020.29 (cit. on pp. 12, 29, 58, 59, 61, 62, 65, 71, 94, 96).

[DGK18] Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. "Regular Transducer Expressions for Regular Transformations." en. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*. Oxford, United Kingdom: ACM Press, 2018, pp. 315–324. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209182 (cit. on p. 36).

[DGK21] Luc Dartois, Paul Gastin, and Shankara Narayanan Krishna. "SD-Regular Transducer Expressions for Aperiodic Transformations." In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470738 (cit. on p. 36).

[DH11] Ugo Dal Lago and Martin Hofmann. "Realizability models and implicit complexity." In: *Theoretical Computer Science*. Girard's Festschrift 412.20 (Apr. 2011), pp. 2029–2047. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2010.12.025 (cit. on p. 18).

[DJ03] Vincent Danos and Jean-Baptiste Joinet. "Linear logic and elementary time." In: *Information and Computation*. International Workshop on Implicit Computational Complexity (ICC'99) 183.1 (May 2003), pp. 123–137. ISSN: 0890-5401. DOI: 10.1016/S0890-5401(03)00010-5 (cit. on p. 30).

[DJR18] Luc Dartois, Ismaël Jecker, and Pierre-Alain Reynier. "Aperiodic String Transducers." en. In: *International Journal of Foundations of Computer Science* 29.05 (Aug. 2018), pp. 801–824. ISSN: 0129-0541, 1793-6373. DOI: 10.1142/S0129054118420054 (cit. on pp. 56, 59, 60).

[DKS12] Volker Diekert, Manfred Kufleitner, and Benjamin Steinberg. "The Krohn-Rhodes Theorem and Local Divisors." In: *Fundamenta Informaticae* 116.1-4 (2012), pp. 65–77. DOI: 10.3233/FI-2012-669 (cit. on p. 54).

[DLP21] Elena Di Lavore, Wilmer Leal, and Valeria de Paiva. *Dialectica Petri nets*. 2021. arXiv: 2105.12801 [math.CT] (cit. on p. 22).

[DO18] Anupam Das and Isabel Oitavem. "A Recursion-Theoretic Characterisation of the Positive Polynomial-Time Functions." In: *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*. Ed. by Dan R. Ghica and Achim Jung. Vol. 119. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 18:1–18:17. DOI: 10.4230/LIPIcs.CSL.2018.18 (cit. on p. 34).

[Dou17] Amina Doumane. "On the infinitary proof theory of logics with fixed points." PhD thesis. Université Paris 7, June 2017. URL: https://hal.archives-ouvertes.fr/tel-01676953 (cit. on p. 37).

[Dou21] Gaëtan Douéneau-Tabot. "Pebble Transducers with Unary Output." In: *46th International Symposium on Mathematical Foundations of Computer Science (MFCS 2021)*. Ed. by Filippo Bonchi and Simon J. Puglisi. Vol. 202. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 40:1–40:17. ISBN: 978-3-95977-201-3. DOI: 10.4230/LIPIcs.MFCS.2021.40 (cit. on pp. 90, 94).

[Dou66] Adrien Douady. "Le problème des modules pour les sous-espaces analytiques compacts d'un espace analytique donné." fr. In: *Annales de l'Institut Fourier* 16.1 (1966), pp. 1–95. DOI: 10.5802/aif.226 (cit. on p. 24).

[DP16] Henry DeYoung and Frank Pfenning. "Substructural Proofs as Automata." In: *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. Ed. by Atsushi Igarashi. Vol. 10017. Lecture Notes in Computer Science. 2016, pp. 3–22. DOI: 10.1007/978-3-319-47958-3_1 (cit. on p. 36).

[DS16] Ugo Dal Lago and Ulrich Schöpp. "Computation by interaction for space-bounded functional programming." In: *Information and Computation*. Development on Implicit Computational Complexity (DICE 2013) 248 (June 2016), pp. 150–194. ISSN: 0890-5401. DOI: 10.1016/j.ic.2015.04.006 (cit. on pp. 45, 50).

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. "MSO definable string transductions and two-way finite-state transducers." en. In: *ACM Transactions on Computational Logic* 2.2 (Apr. 2001), pp. 216–254. ISSN: 15293785. DOI: 10.1145/371316.371512 (cit. on pp. 32, 67, 74, 75).

[Ehr18] Thomas Ehrhard. "An introduction to differential linear logic: proof-nets, models and antiderivatives." In: *Mathematical Structures in Computer Science* 28.7 (2018), pp. 995–1060. DOI: 10.1017/S0960129516000372 (cit. on p. 19).

[Ehr93] Thomas Ehrhard. "Hypercoherences: A Strongly Stable Model of Linear Logic." In: *Mathematical Structures in Computer Science* 3.4 (1993), pp. 365–385. DOI: 10.1017/S0960129500000281 (cit. on p. 50).

[EHS21] Joost Engelfriet, Hendrik Jan Hoogeboom, and Bart Samwel. "XML navigation and transformation by tree-walking automata and transducers with visible and invisible pebbles." In: *Theoretical Computer Science* 850 (Jan. 2021), pp. 40–97. DOI: 10.1016/j.tcs.2020.10.030 (cit. on p. 74).

[EIM21] Joost Engelfriet, Kazuhiro Inaba, and Sebastian Maneth. "Linear-bounded composition of tree-walking tree transducers: linear size increase and complexity." In: *Acta Informatica* 58.1-2 (2021), pp. 95–152. DOI: 10.1007/s00236-019-00360-8 (cit. on p. 47).

[Elg61] Calvin C. Elgot. "Decision Problems of Finite Automata Design and Related Arithmetics." In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51. ISSN: 00029947. DOI: 10.2307/1993511 (cit. on p. 31).

[EM03a] Joost Engelfriet and Sebastian Maneth. "A comparison of pebble tree transducers with macro tree transducers." In: *Acta Informatica* 39.9 (2003), pp. 613–698. DOI: 10.1007/s00236-003-0120-0 (cit. on pp. 28, 41).

[EM03b] Joost Engelfriet and Sebastian Maneth. "Macro Tree Translations of Linear Size Increase are MSO Definable." In: *SIAM Journal on Computing* 32.4 (2003), pp. 950–1006. DOI: 10.1137/S0097539701394511 (cit. on p. 59).

[EM99] Joost Engelfriet and Sebastian Maneth. "Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations." In: *Information and Computation* 154.1 (Oct. 1999), pp. 34–91. ISSN: 0890-5401. DOI: 10.1006/inco.1999.2807 (cit. on pp. 15, 26, 45, 67, 170, 181).

[ER03] Thomas Ehrhard and Laurent Regnier. "The differential lambda-calculus." In: *Theoretical Computer Science* 309.1-3 (2003), pp. 1–41. DOI: 10.1016/S0304-3975(03)00392-X (cit. on p. 19).

[ER08] Thomas Ehrhard and Laurent Regnier. "Uniformity and the Taylor expansion of ordinary lambda-terms." In: *Theoretical Computer Science* 403.2-3 (2008), pp. 347–372. DOI: 10.1016/j.tcs.2008.06.001 (cit. on p. 47).

[ERS80] Joost Engelfriet, Grzegorz Rozenberg, and Giora Slutzki. "Tree Transducers, L Systems, and Two-Way Machines." In: *Journal of Computer and System Sciences* 20.2 (1980), pp. 150–202. DOI: 10.1016/0022-0000(80)90058-6 (cit. on pp. 15, 61).

[Esc21] Martín Hötzel Escardó. "The Cantor–Schröder–Bernstein Theorem for ∞-groupoids." In: *Journal of Homotopy and Related Structures* 16 (2021), pp. 363–366. DOI: 10.1007/s40062-021-00284-6 (cit. on p. 35).

[EV85] Joost Engelfriet and Heiko Vogler. "Macro Tree Transducers." In: *Journal of Computer and System Sciences* 31.1 (1985), pp. 71–146. DOI: 10.1016/0022-0000(85)90066-2 (cit. on pp. 26, 62, 170).

[EV86] Joost Engelfriet and Heiko Vogler. "Pushdown Machines for the Macro Tree Transducer." In: *Theoretical Computer Science* 42 (1986), pp. 251–368. DOI: 10.1016/0304-3975(86)90052-6 (cit. on p. 41).

[EV88] Joost Engelfriet and Heiko Vogler. "High Level Tree Transducers and Iterated Pushdown Tree Transducers." In: *Acta Informatica* 26.1/2 (1988), pp. 131–192. DOI: 10.1007/BF02915449 (cit. on pp. 41, 42).

[Fag73] Ronald Fagin. "Contributions to the model theory of finite structures." PhD thesis. University of California, Berkeley, 1973 (cit. on p. 32).

[Fér17] Hugo Férée. "Game semantics approach to higher-order complexity." In: *Journal of Computer and System Sciences* 87 (2017), pp. 1–15. DOI: 10.1016/j.jcss.2017.02.003 (cit. on p. 34).

[FG20] Laura Fontanella and Guillaume Geoffroy. "Preserving cardinals and weak forms of Zorn's lemma in realizability models." In: *Mathematical Structures in Computer Science* 30.9 (2020), pp. 976–996. DOI: 10.1017/S0960129521000013 (cit. on p. 22).

[FGL16] Emmanuel Filiot, Olivier Gauwin, and Nathan Lhote. "First-order definability of rational transductions: An algebraic approach." In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. Ed. by Martin Grohe, Eric Koskinen, and Natarajan Shankar. ACM, 2016, pp. 387–396. DOI: 10.1145/2933575.2934520 (cit. on p. 32).

[Fil15] Emmanuel Filiot. "Logic-Automata Connections for Transformations." In: *Logic and Its Applications - 6th Indian Conference, ICLA 2015, Mumbai, India, January 8-10, 2015. Proceedings*. Ed. by Mohua Banerjee and Shankara Narayanan Krishna. Vol. 8923. Lecture Notes in Computer Science. Springer, 2015, pp. 30–57. DOI: 10.1007/978-3-662-45824-2_3 (cit. on p. 32).

[FLO83] Steven Fortune, Daniel Leivant, and Michael O'Donnell. "The Expressiveness of Simple and Second-Order Type Structures." In: *Journal of the ACM* 30.1 (Jan. 1983), pp. 151–185. ISSN: 0004-5411. DOI: 10.1145/322358.322370 (cit. on p. 39).

[FMS14] Julien Ferté, Nathalie Marin, and Géraud Sénizergues. "Word-Mappings of Level 2." en. In: *Theory of Computing Systems* 54.1 (Jan. 2014), pp. 111–148. ISSN: 1433-0490. DOI: 10.1007/s00224-013-9489-5 (cit. on pp. 12, 29, 41, 61).

[FR16] Emmanuel Filiot and Pierre-Alain Reynier. "Transducers, Logic and Algebra for Functions of Finite Words." In: *ACM SIGLOG News* 3.3 (Aug. 2016), pp. 4–19. ISSN: 2372-3491. DOI: 10.1145/2984450.2984453 (cit. on pp. 12, 15).

[FR21] Emmanuel Filiot and Pierre-Alain Reynier. "Copyful Streaming String Transducers." In: *Fundamenta Informaticae* 178.1-2 (Jan. 2021). Journal version of a RP'17 paper, pp. 59–76. DOI: 10.3233/FI-2021-1998 (cit. on pp. 12, 15, 29, 56, 61–63).

[FS06] Séverine Fratani and Géraud Sénizergues. "Iterated pushdown automata and sequences of rational numbers." In: *Annals of Pure and Applied Logic* 141.3 (Sept. 2006), pp. 363–411. ISSN: 0168-0072. DOI: 10.1016/j.apal.2005.12.004 (cit. on p. 41).

[FW21] Emmanuel Filiot and Sarah Winter. "Synthesizing Computable Functions from Rational Specifications over Infinite Words." In: *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021)*. Ed. by Mikołaj Bojańczyk and Chandra Chekuri. Vol. 213. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 43:1–43:16. ISBN: 978-3-95977-215-0. DOI: 10.4230/LIPIcs.FSTTCS.2021.43 (cit. on p. 13).

[Gal20] Zeinab Galal. "A Profunctorial Scott Semantics." In: *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*. Ed. by Zena M. Ariola. Vol. 167. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 16:1–16:18. DOI: 10.4230/LIPIcs.FSCD.2020.16 (cit. on p. 106).

[GG15] Erich Grädel and Martin Grohe. "Is Polynomial Time Choiceless?" In: *Fields of Logic and Computation II – Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 193–209. DOI: 10.1007/978-3-319-23534-9_11 (cit. on p. 34).

[Gha+18] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. "Compositional Game Theory." In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 472–481. DOI: 10.1145/3209108.3209165 (cit. on p. 22).

[Ghy21] Alexis Ghyselen. "Sized Types Methods and their Applications to Complexity Analysis in Pi-Calculus." PhD thesis. École normale supérieure de Lyon, Sept. 2021 (cit. on p. 36).

[Gir03] Jean-Yves Girard. “From foundations to ludics.” In: *Bulletin of Symbolic Logic* 9.2 (2003), pp. 131–168. DOI: 10.2178/bsl/1052669286 (cit. on pp. 30, 35, 37).

[Gir11a] Jean-Yves Girard. “Geometry of Interaction V: Logic in the hyperfinite factor.” In: *Theoretical Computer Science* 412.20 (2011), pp. 1860–1883. DOI: 10.1016/j.tcs.2010.12.016 (cit. on pp. 35, 44).

[Gir11b] Jean-Yves Girard. *The Blind Spot: Lectures on logic*. European Mathematical Society, Sept. 2011. ISBN: 978-3-03719-088-3. DOI: 10.4171/088 (cit. on p. 30).

[Gir12] Jean-Yves Girard. “Normativity in Logic.” en. In: *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*. Logic, Epistemology, and the Unity of Science. Springer, Dordrecht, 2012, pp. 243–263. DOI: 10.1007/978-94-007-4435-6_12 (cit. on p. 35).

[Gir86] Jean-Yves Girard. “The system F of variable types, fifteen years later.” In: *Theoretical Computer Science* 45 (Jan. 1986), pp. 159–192. ISSN: 0304-3975. DOI: 10.1016/0304-3975(86)90044-7 (cit. on p. 49).

[Gir87] Jean-Yves Girard. “Linear logic.” In: *Theoretical Computer Science* 50.1 (Jan. 1987), pp. 1–101. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90045-4 (cit. on pp. 15, 16, 19, 25, 143, 180).

[Gir89a] Jean-Yves Girard. “Geometry of Interaction 1: Interpretation of System F.” In: *Studies in Logic and the Foundations of Mathematics*. Ed. by R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo. Vol. 127. Logic Colloquium ’88. Elsevier, Jan. 1989, pp. 221–260 (cit. on p. 44).

[Gir89b] Jean-Yves Girard. “Towards a Geometry of Interaction.” In: *Categories in Computer Science and Logic*. Ed. by John W. Gray and Andre Scedrov. Vol. 92. Contemporary Mathematics. Proceedings of a Summer Research Conference held June 14–20, 1987. Providence, RI: American Mathematical Society, 1989, pp. 69–108. DOI: 10.1090/conm/092/1003197 (cit. on pp. 16, 19, 44).

[Gir91] Jean-Yves Girard. “A New Constructive Logic: Classical Logic.” In: *Mathematical Structures in Computer Science* 1.3 (1991), pp. 255–296. DOI: 10.1017/S0960129500001328 (cit. on p. 22).

[Gir95] Jean-Yves Girard. “Linear logic: its syntax and semantics.” In: *Advances in Linear Logic*. Ed. by Jean-Yves Girard, Yves Lafont, and Laurent Regnier. Vol. 222. London Mathematical Society Lecture Notes. Cambridge University Press, 1995, pp. 1–42. DOI: 10.1017/CBO9780511629150.002 (cit. on pp. 141, 180).

[Gir96] Jean-Yves Girard. “Proof-nets: The parallel syntax for proof-theory.” In: *Logic and Algebra*. Marcel Dekker, 1996, pp. 97–124 (cit. on p. 180).

[Gir98] Jean-Yves Girard. “Light Linear Logic.” In: *Information and Computation* 143.2 (June 1998), pp. 175–204. ISSN: 0890-5401. DOI: 10.1006/inco.1998.2700 (cit. on pp. 18, 30).

[GK07] Neil Ghani and Alexander Kurz. “Higher Dimensional Trees, Algebraically.” In: *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007, Bergen, Norway, August 20-24, 2007, Proceedings*. Ed. by Till Mossakowski, Ugo Montanari, and Magne Haveraaen. Vol. 4624. Lecture Notes in Computer Science. Springer, 2007, pp. 226–241. DOI: 10.1007/978-3-540-73859-6_16 (cit. on p. 37).

[GK13] Nicola Gambino and Joachim Kock. “Polynomial functors and polynomial monads.” In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 154. Cambridge University Press. 2013, pp. 153–192. DOI: 10.1017/S0305004112000394 (cit. on p. 116).

[GLS20] Paul Gallot, Aurélien Lemay, and Sylvain Salvati. “Linear High-Order Deterministic Tree Transducers with Regular Look-Ahead.” In: *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*. Ed. by Javier Esparza and Daniel Král’. Vol. 170. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 38:1–38:13. DOI: 10.4230/LIPIcs.MFCS.2020.38 (cit. on pp. 26, 180).

[Göd58] Kurt Gödel. “Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes.” In: *Dialectica* 12.3-4 (1958). English translation in [Göd80], pp. 280–287. DOI: 10.1111/j.1746-8361.1958.tb01464.x (cit. on pp. 14, 18, 22, 116, 208).

[Göd80] Kurt Gödel. "On a hitherto unexploited extension of the finitary standpoint." In: *Journal of Philosophical Logic* 9 (May 1980). Annotated translation of [Göd58] by Wilfrid Hodges and Bruce Watson, pp. 133–142. DOI: 10.1007/BF00247744 (cit. on p. 207).

[Gra20] Julien Grange. "On the Expressive Power of Invariant Logics over Sparse Classes of Structures. (Sur le pouvoir d'expression des logiques définies par invariance)." PhD thesis. École Normale Supérieure, Paris, France, 2020. URL: https://tel.archives-ouvertes.fr/tel-02947853 (cit. on p. 33).

[Gre16] Charles Grellois. "Semantics of linear logic and higher-order model-checking." en. PhD thesis. Université Paris 7, Apr. 2016. URL: https://tel.archives-ouvertes.fr/tel-01311150/ (cit. on p. 38).

[Gri90] Timothy Griffin. "A Formulae-as-Types Notion of Control." In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. Ed. by Frances E. Allen. ACM Press, 1990, pp. 47–58. DOI: 10.1145/96709.96714 (cit. on p. 22).

[Gro16] Alexander Grothendieck. "Allons-nous continuer la recherche scientifique ?" fr. In: *Écologie & politique* 52.1 (2016). Transcription of a talk at CERN on January 27th, 1972, with a foreword by Céline Pessis, pp. 159–169. DOI: 10.3917/ecopo1.052.0159 (cit. on p. 24).

[Gro17] Martin Grohe. *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*. Vol. 47. Lecture Notes in Logic. Cambridge University Press, 2017. ISBN: 9781139028868. DOI: 10.1017/9781139028868 (cit. on p. 34).

[GRV09] Marco Gaboardi, Luca Roversi, and Luca Vercelli. "A By-Level Analysis of Multiplicative Exponential Linear Logic." en. In: *Mathematical Foundations of Computer Science 2009*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Aug. 2009, pp. 344–355. DOI: 10.1007/978-3-642-03816-7_30 (cit. on p. 40).

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. "Bounded linear logic: a modular approach to polynomial-time computability." In: *Theoretical Computer Science* 97.1 (Apr. 1992), pp. 1–66. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90386-T (cit. on p. 33).

[GT07] Rajeev Goré and Alwen Tiu. "Classical Modal Display Logic in the Calculus of Structures and Minimal Cut-free Deep Inference Calculi for S5." In: *Journal of Logic and Computation* 17.4 (2007), pp. 767–794. DOI: 10.1093/logcom/exm026 (cit. on p. 30).

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. USA: Cambridge University Press, 1989. ISBN: 0521371813 (cit. on pp. 49, 180).

[Gué19] Armaël Guéneau. "Mechanized Verification of the Correctness and Asymptotic Complexity of Programs." PhD thesis. Université de Paris, Dec. 2019. URL: https://hal.inria.fr/tel-02437532 (cit. on p. 33).

[Gug07] Alessio Guglielmi. "A system of interaction and structure." en. In: *ACM Transactions on Computational Logic* 8.1 (Jan. 2007). ISSN: 15293785. DOI: 10.1145/1182613.1182614 (cit. on p. 16).

[Gui16] Bruno Guillon. "Input- or output-unary sweeping transducers are weaker than their 2-way counterparts." In: *RAIRO – Theoretical Informatics and Applications* 50.4 (2016), pp. 275–294. DOI: 10.1051/ita/2016028 (cit. on p. 91).

[Gur83] Yuri Gurevich. "Algebras of feasible functions." en. In: *24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*. Tucson, AZ, USA, Nov. 1983, pp. 210–214. ISBN: 978-0-8186-0508-6. DOI: 10.1109/SFCS.1983.5 (cit. on p. 33).

[H16] Piper H. "The Equidistribution of Lattice Shapes of Rings of Integers of Cubic, Quartic, and Quintic Number Fields: An Artist's Rendering." en. The author was previously known as Piper Alexis Harron. PhD thesis. Princeton University, 2016. URL: http://arks.princeton.edu/ark:/88435/dsp01ws859j05g (cit. on p. 196).

[Hee+19] Gerco van Heerdt, Tobias Kappé, Jurriaan Rot, Matteo Sammartino, and Alexandra Silva. "Tree Automata as Algebras: Minimisation and Determinisation." In: *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019, June 3-6, 2019, London, United Kingdom*. Ed. by Markus Roggenbach and Ana Sokolova. Vol. 139. LIPIcs. Schloss

Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 6:1–6:22. DOI: 10.4230/LIPIcs.CALCO.2019.6 (cit. on p. 23).

[HH16] Dominic Hughes and Willem Heijltjes. "Conflict nets: Efficient locally canonical MALL proof nets." In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), 2016*. New York, U. S. A.: ACM, July 2016, pp. 437–446. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934559 (cit. on p. 180).

[Hin03] Peter Hines. "A Categorical Framework For Finite State Machines." In: *Mathematical Structures in Computer Science* 13.3 (2003), pp. 451–480. DOI: 10.1017/S0960129503003931 (cit. on pp. 23, 44).

[Hin06] Peter Hines. *Temperley-Lieb Algebras as two-way automata*. Slides of a talk given at the QNET Workshop 2006. 2006. URL: http://www.dcs.gla.ac.uk/~simon/qnet/talks/Hines.pdf (cit. on p. 44).

[HJ99] Hongde Hu and André Joyal. "Coherence completions of categories." In: *Theoretical Computer Science* 227.1 (Sept. 1999), pp. 153–184. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(99)00051-1 (cit. on pp. 180, 184).

[HK96] 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*. IEEE Computer Society, 1996, pp. 253–263. ISBN: 978-0-8186-7463-1. DOI: 10.1109/LICS.1996.561337 (cit. on pp. 16–18, 40, 49).

[HKM11] Markus Holzer, Martin Kutrib, and Andreas Malcher. "Complexity of multi-head finite automata: Origins and directions." In: *Theoretical Computer Science* 412.1-2 (2011), pp. 83–96. DOI: 10.1016/j.tcs.2010.08.024 (cit. on p. 9).

[HKM96] Gerd G. Hillebrand, Paris C. Kanellakis, and Harry G. Mairson. "Database Query Languages Embedded in the Typed Lambda Calculus." In: *Information and Computation* 127.2 (June 1996), pp. 117–144. ISSN: 0890-5401. DOI: 10.1006/inco.1996.0055 (cit. on pp. 15, 17, 40, 49).

[HL21] Martin Hofmann and Jérémy Ledent. "A quantitative model for simply typed λ-calculus." In: *Mathematical Structures in Computer Science* (2021), pp. 1–17. DOI: 10.1017/S0960129521000256 (cit. on p. 18).

[HMP20] Emmanuel Hainry, Damiano Mazza, and Romain Péchoux. "Polynomial Time over the Reals with Parsimony." In: *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*. Ed. by Keisuke Nakano and Konstantinos Sagonas. Vol. 12073. Lecture Notes in Computer Science. Springer, 2020, pp. 50–65. DOI: 10.1007/978-3-030-59025-3_4 (cit. on pp. 29, 35).

[Hof11] Pieter Hofstra. "The dialectica monad and its cousins." en. In: *Models, Logics, and Higher-Dimensional Categories: A Tribute to the Work of Mihály Makkai*. Ed. by Bradd Hart, Thomas Kucera, Anand Pillay, Philip Scott, and Robert Seely. Vol. 53. CRM Proceedings and Lecture Notes. Providence, Rhode Island: American Mathematical Society, Sept. 2011. DOI: 10.1090/crmp/053 (cit. on pp. 27, 116).

[How80] William Alvin Howard. "The Formulae-as-Types Notion of Construction." In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Reprint of an informally circulated paper written in 1969. Academic Press, 1980 (cit. on p. 14).

[Hug04] Dominic J.D. Hughes. "Deep inference proof theory *equals* categorical proof theory *minus* coherence." Unfinished draft. 2004. URL: http://boole.stanford.edu/~dominic/papers/di/di.pdf (cit. on p. 16).

[Hyl10] Martin Hyland. "Some reasons for generalising domain theory." In: *Mathematical Structures in Computer Science* 20.2 (2010), pp. 239–265. DOI: 10.1017/S0960129509990375 (cit. on p. 20).

[Hyl17] Martin Hyland. "Classical lambda calculus in modern dress." In: *Mathematical Structures in Computer Science* 27.5 (2017), pp. 762–781. DOI: 10.1017/S0960129515000377 (cit. on p. 21).

[Imm86] Neil Immerman. “Relational Queries Computable in Polynomial Time.” In: *Information and Control* 68.1-3 (1986). Long version of a STOC’82 paper, pp. 86–104. DOI: 10.1016/S0019-9958(86)80029-8 (cit. on p. 34).

[Imm99] Neil Immerman. *Descriptive Complexity*. Graduate Texts in Computer Ccience. Springer, 1999. ISBN: 978-1-4612-6809-3. DOI: 10.1007/978-1-4612-0539-5 (cit. on pp. 32, 33).

[Jol01] Thierry Joly. “Constant time parallel computations in λ-calculus.” In: *Theoretical Computer Science* 266.1 (Sept. 2001), pp. 975–985. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00380-7 (cit. on p. 39).

[JSV96] André Joyal, Ross Street, and Dominic Verity. “Traced monoidal categories.” In: *Mathematical Proceedings of the Cambridge Philosophical Society* 119.3 (1996), pp. 447–468. DOI: 10.1017/S0305004100074338 (cit. on p. 23).

[Jun+21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “Safe systems programming in Rust.” In: *Communications of the ACM* 64.4 (2021), pp. 144–152. DOI: 10.1145/3418295 (cit. on p. 16).

[Keg08] Jeffrey Kegler. “Perl and Undecidability, Part III.” In: *The Perl Review* 5.0 (2008). URL: http://www.jeffreykegler.com/Home/perl-and-undecidability (cit. on p. 10).

[Ker18] Marie Kerjean. “Reflexive spaces of smooth functions: a logical account of linear partial differential equations.” PhD thesis. Université Sorbonne Paris Cité, Oct. 2018. URL: https://tel.archives-ouvertes.fr/tel-02386294 (cit. on p. 19).

[Kie20a] Sandra Kiefer. “Power and limits of the Weisfeiler-Leman algorithm.” PhD thesis. RWTH Aachen University, 2020. DOI: 10.18154/RWTH-2020-03508 (cit. on p. 34).

[Kie20b] Sandra Kiefer. “The Weisfeiler-Leman algorithm: an exploration of its power.” In: *ACM SIGLOG News* 7.3 (2020), pp. 5–27. DOI: 10.1145/3436980.3436982 (cit. on p. 34).

[KL21] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. “The simplicial model of Univalent Foundations (after Voevodsky).” In: *Journal of the European Mathematical Society* 23 (6 2021), pp. 2071–2126. DOI: 10.4171/JEMS/1050 (cit. on p. 19).

[KL80] Gregory M. Kelly and Miguel L. Laplaza. “Coherence for compact closed categories.” In: *Journal of pure and applied algebra* 19 (1980), pp. 193–213. DOI: 10.1016/0022-4049(80)90101-2 (cit. on p. 16).

[KNU02] Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. “Higher-Order Pushdown Trees Are Easy.” In: *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Mogens Nielsen and Uffe Engberg. Vol. 2303. Lecture Notes in Computer Science. Springer, 2002, pp. 205–222. DOI: 10.1007/3-540-45931-6_15 (cit. on p. 38).

[Kob13] Naoki Kobayashi. “Model Checking Higher-Order Programs.” en. In: *Journal of the ACM* 60.3 (June 2013), pp. 1–62. ISSN: 00045411. DOI: 10.1145/2487241.2487246 (cit. on p. 38).

[Kob19] Naoki Kobayashi. “10 Years of the Higher-Order Model Checking Project (Extended Abstract).” In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 2:1–2:2. DOI: 10.1145/3354166.3354167 (cit. on p. 38).

[Kob85] Kojiro Kobayashi. “On the structure of one-tape nondeterministic Turing machine time hierarchy.” In: *Theoretical Computer Science* 40 (1985), pp. 175–193. DOI: 10.1016/0304-3975(85)90165-3 (cit. on p. 10).

[Koł+19] Leszek Aleksander Kołodziejczyk, Henryk Michalewski, Pierre Pradic, and Michał Skrzypczak. “The logical strength of Büchi’s decidability theorem.” In: *Logical Methods in Computer Science* 15.2 (2019). DOI: 10.23638/LMCS-15(2:16)2019 (cit. on pp. 12, 37).

[KP21] Marie Kerjean and Pierre-Marie Pédrot. “∂ is for Dialectica: Typing Differentiable Programming.” working paper or preprint. Feb. 2021. URL: https://hal.archives-ouvertes.fr/hal-03123968 (cit. on p. 22).

[KPP19] Denis Kuperberg, Laureline Pinault, and Damien Pous. "Cyclic Proofs and Jumping Automata." In: *39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019)*. Ed. by Arkadev Chattopadhyay and Paul Gastin. Vol. 150. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 45:1–45:14. ISBN: 978-3-95977-131-3. DOI: 10.4230/LIPIcs.FSTTCS.2019.45 (cit. on p. 36).

[KR65] Kenneth Krohn and John Rhodes. "Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines." en. In: *Transactions of the American Mathematical Society* 116 (1965), pp. 450–464. ISSN: 0002-9947, 1088-6850. DOI: 10.1090/S0002-9947-1965-0188316-1 (cit. on pp. 12, 53).

[Kre15] Robbert Krebbers. "The C standard formalized in Coq." en. PhD thesis. Radboud Universiteit Nijmegen, Dec. 2015. ISBN: 9789462599031. URL: http://hdl.handle.net/2066/147182 (cit. on p. 19).

[Kri09] Jean-Louis Krivine. "Realizability in classical logic." In: *Interactive models of computation and program behaviour*. Vol. 27. Panoramas et Synthèses. Société Mathématique de France, 2009, pp. 197–229. URL: https://hal.archives-ouvertes.fr/hal-00154500 (cit. on p. 22).

[Kri12] Lars Kristiansen. "Higher Types, Finite Domains and Resource-bounded Turing Machines." en. In: *Journal of Logic and Computation* 22.2 (Apr. 2012), pp. 281–304. ISSN: 0955-792X, 1465-363X. DOI: 10.1093/logcom/exq009 (cit. on pp. 18, 20).

[Kri20] Jean-Louis Krivine. "A program for the full axiom of choice." In: *CoRR* abs/2006.05433 (2020). arXiv: 2006.05433 (cit. on p. 22).

[KSK08] Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. "Translation of tree-processing programs into stream-processing programs based on ordered linear type." In: *Journal of Functional Programming* 18.3 (2008), pp. 333–371. DOI: 10.1017/S0956796807006570 (cit. on p. 16).

[Kum+14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: a verified implementation of ML." In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 179–192. DOI: 10.1145/2535838.2535841 (cit. on p. 19).

[Kup21] Denis Kuperberg. "Positive First-order Logic on Words." In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470602 (cit. on p. 34).

[Lai13] James Laird. "Game semantics for a polymorphic programming language." In: *Journal of the ACM* 60.4 (2013), 29:1–29:27. DOI: 10.1145/2505986 (cit. on p. 48).

[Lam58] Joachim Lambek. "The Mathematics of Sentence Structure." In: *The American Mathematical Monthly* 65.3 (1958), pp. 154–170. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2310058 (cit. on p. 16).

[Lam72] Joachim Lambek. "Deductive systems and categories III. Cartesian closed categories, intuitionist propositional calculus, and combinatory logic." In: *Toposes, Algebraic Geometry and Logic*. Ed. by F. W. Lawvere. Berlin, Heidelberg: Springer, 1972, pp. 57–82. ISBN: 978-3-540-37609-5. DOI: 10.1007/BFb0073965 (cit. on p. 21).

[Lau02] Olivier Laurent. "Étude de la polarisation en logique." PhD thesis. Université de la Méditerranée – Aix-Marseille II, Mar. 2002. URL: https://tel.archives-ouvertes.fr/tel-00007884 (cit. on p. 37).

[Lau20] Olivier Laurent. "Polynomial time in untyped elementary linear logic." en. In: *Theoretical Computer Science* 813 (Apr. 2020), pp. 117–142. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.10.002 (cit. on pp. 19, 26).

[Lei93] Daniel Leivant. "Functions over free algebras definable in the simply typed lambda calculus." In: *Theoretical Computer Science* 121.1 (Dec. 1993), pp. 309–321. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90092-8 (cit. on p. 39).

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler." In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814 (cit. on p. 19).

[Lho20] Nathan Lhote. "Pebble Minimization of Polyregular Functions." In: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*. Ed. by Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller. ACM, 2020, pp. 703–712. DOI: 10.1145/3373718.3394804 (cit. on pp. 71, 74–76, 81–83, 85).

[Lin07] Sam Lindley. "Extensional Rewriting with Sums." In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 255–271. DOI: 10.1007/978-3-540-73228-0_19 (cit. on p. 140).

[Lin68] Aristid Lindenmayer. "Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs." en. In: *Journal of Theoretical Biology* 18.3 (Mar. 1968), pp. 300–315. ISSN: 00225193. DOI: 10.1016/0022-5193(68)90080-5 (cit. on p. 61).

[LM93] Daniel Leivant and Jean-Yves Marion. "Lambda Calculus Characterizations of Poly-time." In: *Fundamenta Informaticae* 19.1-2 (Sept. 1993), pp. 167–184. ISSN: 0169-2968. DOI: 10.3233/FI-1993-191-207 (cit. on p. 33).

[LN15] John Longley and Dag Normann. *Higher-Order Computability*. Theory and Applications of Computability. Springer, 2015. DOI: 10.1007/978-3-662-47992-6 (cit. on p. 35).

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician*. Second edition. Graduate Texts in Mathematics. Springer, 1998. ISBN: 9780387984032. DOI: 10.1007/978-1-4757-4721-8 (cit. on pp. 30, 101, 123, 124, 177).

[Mai02] Harry G. Mairson. "From Hilbert Spaces to Dilbert Spaces: Context Semantics Made Simple." In: *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference Kanpur, India, December 12-14, 2002, Proceedings*. Ed. by Manindra Agrawal and Anil Seth. Vol. 2556. Lecture Notes in Computer Science. Springer, 2002, pp. 2–17. DOI: 10.1007/3-540-36206-1_2 (cit. on p. 44).

[Mar18] Sonia Marin. "Modal proof theory through a focused telescope." PhD thesis. Université Paris Saclay, Jan. 2018. URL: https://hal.archives-ouvertes.fr/tel-01951291 (cit. on p. 30).

[Mat02] A. R. D. Mathias. "A Term of Length 4 523 659 424 929." In: *Synthese* 133 (2002), pp. 75–86. ISSN: 00397857, 15730964. DOI: 10.1023/A:1020827725055 (cit. on p. 8).

[Mat15] Satoshi Matsuoka. "A New Proof of P-time Completeness of Linear Lambda Calculus." In: *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*. Ed. by Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov. Vol. 35. EPiC Series in Computing. EasyChair, 2015, pp. 119–130. DOI: 10.29007/svwc (cit. on p. 192).

[Maz15] Damiano Mazza. "Simple Parsimonious Types and Logarithmic Space." In: *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. 2015, pp. 24–40. ISBN: 978-3-939897-90-3. DOI: 10.4230/LIPIcs.CSL.2015.24 (cit. on pp. 29, 45, 46, 50).

[Maz17] Damiano Mazza. "Polyadic Approximations in Logic and Computation." Habilitation à diriger des recherches. Université Paris 13, Nov. 2017. URL: https://lipn.fr/~mazza/papers/Habilitation.pdf (cit. on pp. 25, 30, 35, 36).

[Mel09] Paul-André Melliès. "Categorical semantics of linear logic." In: *Interactive models of computation and program behaviour*. Vol. 27. Panoramas et Synthèses. Société Mathématique de France, 2009, pp. 1–196. URL: https://www.irif.fr/~mellies/papers/panorama.pdf (cit. on pp. 27, 49, 99–103, 121, 177).

[Mel17a] Paul-André Melliès. "Higher-order parity automata." en. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Reykjavik, Iceland: IEEE, June 2017, pp. 1–12. ISBN: 978-1-5090-3018-7. DOI: 10.1109/LICS.2017.8005077 (cit. on pp. 37, 38).

[Mel17b] Paul-André Melliès. “Une étude micrologique de la négation.” Habilitation à diriger des recherches. Université Paris VII, Nov. 2017. URL: https://www.irif.fr/~mellies//hdr-mellies.pdf (cit. on p. 16).

[Mel18] Paul-André Melliès. “Ribbon Tensorial Logic.” en. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*. Oxford, United Kingdom: ACM Press, 2018, pp. 689–698. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209129 (cit. on p. 16).

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire.” In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. Ed. by John Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, 1991, pp. 124–144. DOI: 10.1007/3540543961_7 (cit. on p. 38).

[MG18] Sean K. Moss and Tamara von Glehn. “Dialectica models of type theory.” en. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. ACM Press, 2018, pp. 739–748. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209207 (cit. on p. 116).

[MHR20] David MacQueen, Robert Harper, and John H. Reppy. “The history of Standard ML.” In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), 86:1–86:100. DOI: 10.1145/3386336 (cit. on p. 19).

[Mil78] Robin Milner. “A Theory of Type Polymorphism in Programming.” In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cit. on p. 15).

[Mir06] Jolie G. de Miranda. “Structures generated by higher-order grammars and the safety constraint.” PhD thesis. University of Oxford, 2006. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.442397 (cit. on p. 38).

[MM02] Peter Møller Neergaard and Harry Mairson. “LAL is square: representation and expressiveness in light affine logic.” Presented at the 2002 Workshop on Implicit Computational Complexity, Copenhagen. 2002. URL: https://www.cs.brandeis.edu/~mairson/Papers/LAL-is-square.pdf (cit. on p. 40).

[Mör21] Anders Mörtberg. *Cubical Methods in Homotopy Type Theory and Univalent Foundations*. Lecture notes for the 2019 Homotopy Type Theory summer school. 2021. URL: https://staff.math.su.se/anders.mortberg/papers/cubicalmethods.pdf (cit. on p. 20).

[MP19] Anca Muscholl and Gabriele Puppis. “The Many Facets of String Transducers.” In: *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*. Ed. by Rolf Niedermeier and Christophe Paul. Vol. 126. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 2:1–2:21. ISBN: 978-3-95977-100-9. DOI: 10.4230/LIPIcs.STACS.2019.2 (cit. on pp. 12, 75).

[MR12] Richard Moot and Christian Retoré. *The Logic of Categorial Grammars. A Deductive Account of Natural Language Syntax and Semantics*. Vol. 6850. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-31554-1. DOI: 10.1007/978-3-642-31555-8 (cit. on p. 16).

[MS95] David E. Muller and Paul E. Schupp. “Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra.” In: *Theoretical Computer Science* 141.1–2 (1995), pp. 69–107. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)00214-4 (cit. on pp. 28, 133).

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. “Typechecking for XML transformers.” In: *Journal of Computer and System Sciences* 66.1 (2003). Journal version of a PODS 2000 paper, pp. 66–97. DOI: 10.1016/S0022-0000(02)00030-2 (cit. on pp. 28, 74, 75).

[MT03] Harry G. Mairson and Kazushige Terui. “On the Computational Complexity of Cut-Elimination in Linear Logic.” In: *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*. Ed. by Carlo Blundo and Cosimo Laneve. Vol. 2841. Lecture Notes in Computer Science. Springer, 2003, pp. 23–36. DOI: 10.1007/978-3-540-45208-9_4 (cit. on pp. 26, 27).

[MT15] Damiano Mazza and Kazushige Terui. “Parsimonious Types and Non-uniform Computation.” en. In: *Automata, Languages, and Programming*. Lecture Notes in Computer Science. July 2015, pp. 350–361. DOI: 10.1007/978-3-662-47666-6_28 (cit. on pp. 29, 47).

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7 (cit. on p. 19).

[Mun13] Guillaume Munch-Maccagnoni. “Syntax and Models of a non-Associative Composition of Programs and Proofs.” PhD thesis. Université Paris-Diderot - Paris VII, 2013. URL: https://tel.archives-ouvertes.fr/tel-00918642 (cit. on p. 22).

[Mun18] Guillaume Munch-Maccagnoni. *Resource Polymorphism*. 2018. arXiv: 1803.02796 [cs.PL] (cit. on p. 16).

[Ngu19] Lê Thành Dũng Nguyễn. “On the Elementary Affine Lambda-Calculus with and Without Fixed Points.” In: *Electronic Proceedings in Theoretical Computer Science* 298 (Aug. 2019). In Proceedings DICE-FOPARA 2019, pp. 15–29. ISSN: 2075-2180. DOI: 10.4204/EPTCS.298.2 (cit. on pp. 48, 49).

[Ngu20] Lê Thành Dũng Nguyễn. “Unique perfect matchings, forbidden transitions and proof nets for linear logic with Mix.” In: *Logical Methods in Computer Science* 16.27 (Feb. 2020). DOI: 10.23638/LMCS-16(1:27)2020 (cit. on p. 23).

[nLa20] nLab authors. *Semicartesian monoidal category*. http://ncatlab.org/nlab/show/semicartesian%20monoidal%20category. Revision 24. Mar. 2020 (cit. on p. 102).

[NNP21] Lê Thành Dũng Nguyễn, Camille Noûs, and Pierre Pradic. “Comparison-Free Polyregular Functions.” In: *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Ed. by Nikhil Bansal, Emanuela Merelli, and James Worrell. Vol. 198. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 139:1–139:20. ISBN: 978-3-95977-195-5. DOI: 10.4230/LIPIcs.ICALP.2021.139 (cit. on pp. 23, 29, 43, 50, 71).

[NP19] Lê Thành Dũng Nguyễn and Pierre Pradic. “From normal functors to logarithmic space queries.” In: *46th International Colloquium on Automata, Languages and Programming (ICALP 2019)*. Ed. by Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 123:1–123:15. ISBN: 978-3-95977-109-2. DOI: 10.4230/LIPIcs.ICALP.2019.123 (cit. on pp. 49, 50).

[NP20] Lê Thành Dũng Nguyễn and Pierre Pradic. “Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic.” In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by Artur Czumaj, Anuj Dawar, and Emanuela Merelli. Vol. 168. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 135:1–135:20. DOI: 10.4230/LIPIcs.ICALP.2020.135 (cit. on pp. 23, 25, 43, 44, 50, 190–196).

[NP21] Lê Thành Dũng Nguyễn and Pierre Pradic. “The planar geometry of first-order transductions.” In preparation. Slides available at https://nguyentito.eu/2021-01-links.pdf. 2021 (cit. on p. 44).

[NS22] Lê Thành Dũng Nguyễn and Lutz Straßburger. “BV and Pomset Logic are not the same.” In: *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*. Ed. by Florin Manea and Alex Simpson. Vol. 216. Leibniz International Proceedings in Informatics (LIPIcs). Further results are forthcoming in a journal version, see the slides https://nguyentito.eu/2021-01-reseaux.pdf. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 3:1–3:16. DOI: 10.4230/LIPIcs.CSL.2022.3 (cit. on pp. 16, 23).

[ONe09] Melissa E. O’Neill. “The Genuine Sieve of Eratosthenes.” In: *Journal of Functional Programming* 19.1 (2009), pp. 95–106. DOI: 10.1017/S0956796808007004 (cit. on p. 38).

[Ong06] C.-H. Luke Ong. “On Model-Checking Trees Generated by Higher-Order Recursion Schemes.” en. In: *21st Annual IEEE Symposium on Logic in Computer Science (LICS’06)*. Seattle, WA, USA: IEEE, 2006, pp. 81–90. ISBN: 978-0-7695-2631-7. DOI: 10.1109/LICS.2006.38 (cit. on p. 38).

[OP99] Peter W. O'Hearn and David J. Pym. "The logic of bunched implications." In: *Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244. DOI: 10.2307/421090 (cit. on p. 16).

[Pag21] Benedikt Pago. "Choiceless Computation and Symmetry: Limitations of Definability." In: *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*. Ed. by Christel Baier and Jean Goubault-Larrecq. Vol. 183. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 33:1–33:21. DOI: 10.4230/LIPIcs.CSL.2021.33 (cit. on p. 34).

[Pai89] Valeria C. V. de Paiva. "The Dialectica categories." In: *Categories in Computer Science and Logic*. Ed. by John W. Gray and Andre Scedrov. Vol. 92. Contemporary Mathematics. Proceedings of a Summer Research Conference held June 14–20, 1987. Providence, Rhode Island: American Mathematical Society, 1989, pp. 47–62. DOI: 10.1090/conm/092/1003194 (cit. on p. 22).

[Par18] Paweł Parys. "Homogeneity Without Loss of Generality." In: *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. Ed. by Hélène Kirchner. Vol. 108. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 27:1–27:15. DOI: 10.4230/LIPIcs.FSCD.2018.27 (cit. on p. 42).

[Par21] Paweł Parys. "Higher-Order Model Checking Step by Step." In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. Ed. by Nikhil Bansal, Emanuela Merelli, and James Worrell. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 140:1–140:16. DOI: 10.4230/LIPIcs.ICALP.2021.140 (cit. on p. 38).

[Par92] Michel Parigot. "Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction." In: *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*. Ed. by Andrei Voronkov. Vol. 624. Lecture Notes in Computer Science. Springer, 1992, pp. 190–201. DOI: 10.1007/BFb0013061 (cit. on p. 22).

[PB19] Pierre Pradic and Chad E. Brown. *Cantor-Bernstein implies Excluded Middle*. 2019. arXiv: 1904.09193 [math.LO] (cit. on p. 35).

[Péc20] Romain Péchoux. "Implicit Computational Complexity: past and future (Complexité implicite : bilan et perspectives)." Habilitation à diriger des recherches. Université de Lorraine, Oct. 2020. URL: https://hal.univ-lorraine.fr/tel-02978986 (cit. on pp. 15, 33, 35, 36).

[Péd15] Pierre-Marie Pédrot. "A Materialist Dialectica." PhD thesis. Université Paris Diderot - Paris VII, 2015. URL: https://tel.archives-ouvertes.fr/tel-01247085 (cit. on p. 22).

[Pel17] Luc Pellissier. "Reductions and linear approximations." PhD thesis. Université Sorbonne Paris Cité, Dec. 2017. URL: https://tel.archives-ouvertes.fr/tel-02465653 (cit. on p. 21).

[Pet15] Tomas Petricek. "Against a universal definition of 'Type'." In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Gail C. Murphy and Guy L. Steele Jr. Paper and talk recording available at http://tomasp.net/academic/papers/against-types/. ACM, 2015, pp. 254–266. DOI: 10.1145/2814228.2814249 (cit. on p. 14).

[Pet18] Tomas Petricek. "What we talk about when we talk about monads." In: *The Art, Science, and Engineering of Programming* 2.3 (2018), p. 12. DOI: 10.22152/programming-journal.org/2018/2/12 (cit. on p. 21).

[Pin21a] Jean-Éric Pin, ed. *Handbook of Automata Theory*. European Mathematical Society Publishing House, 2021. ISBN: 978-3-98547-006-8. DOI: 10.4171/Automata (cit. on pp. 10, 12, 13, 31, 38, 67).

[Pin21b] Laureline Pinault. "From automata to cyclic proofs: equivalence algorithms and descriptive complexity." PhD thesis. École normale supérieure de Lyon, July 2021. URL: https://tel.archives-ouvertes.fr/tel-03412556 (cit. on pp. 36, 37).

[Pit16] Andrew M. Pitts. "Nominal techniques." In: *ACM SIGLOG News* 3.1 (2016), pp. 57–72. DOI: 10.1145/2893582.2893594 (cit. on pp. 45, 46).

[Pol01] Jeff Polakow. "Ordered linear logic and applications." PhD thesis. Pittsburgh, PA: Carnegie Mellon University, 2001. URL: http://reports-archive.adm.cs.cmu.edu/anon/2001/CMU-CS-01-152.pdf (cit. on p. 25).

[PP99a] Jeff Polakow and Frank Pfenning. "Natural Deduction for Intuitionistic Non-communicative Linear Logic." In: *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*. Ed. by Jean-Yves Girard. Vol. 1581. Lecture Notes in Computer Science. Springer, 1999, pp. 295–309. DOI: 10.1007/3-540-48959-2_21 (cit. on p. 191).

[PP99b] Jeff Polakow and Frank Pfenning. "Relating Natural Deduction and Sequent Calculus for Intuitionistic Non-Commutative Linear Logic." en. In: *Electronic Notes in Theoretical Computer Science*. MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference 20 (Jan. 1999), pp. 449–466. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80088-4 (cit. on p. 191).

[PR19] Pierre Pradic and Colin Riba. "A Dialectica-Like Interpretation of a Linear MSO on Infinite Words." In: *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019*. Ed. by Mikołaj Bojańczyk and Alex Simpson. Vol. 11425. Lecture Notes in Computer Science. Springer, 2019, pp. 470–487. ISBN: 978-3-030-17126-1. DOI: 10.1007/978-3-030-17127-8_27 (cit. on p. 22).

[PS20] Bart Penders and David M. Shaw. "Civil disobedience in scientific authorship: Resistance and insubordination in science." In: *Accountability in Research* 27.6 (2020). PMID: 32299255, pp. 347–371. DOI: 10.1080/08989621.2020.1756787 (cit. on p. 23).

[PS21] Luc Pellissier and Thomas Seiller. "PRAMs over integers do not compute maxflow efficiently." working paper or preprint. Jan. 2021. URL: https://hal.archives-ouvertes.fr/hal-01921942v2 (cit. on p. 35).

[Ran18] Noé de Rancourt. "Ramsey theory without pigeonhole principle and applications to the proof of Banach-space dichotomies." PhD thesis. Université Sorbonne Paris Cité, June 2018. URL: https://tel.archives-ouvertes.fr/tel-02464512 (cit. on p. 12).

[Ret97] Christian Retoré. "Pomset Logic: A Non-commutative Extension of Classical Linear Logic." In: *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*. Ed. by Philippe de Groote. Vol. 1210. Lecture Notes in Computer Science. Springer, 1997, pp. 300–318. DOI: 10.1007/3-540-62688-3_43 (cit. on p. 16).

[Reu79] Christophe Reutenauer. "Sur les séries associées à certains systèmes de Lindenmayer." In: *Theoretical Computer Science* 9 (1979), pp. 363–375. DOI: 10.1016/0304-3975(79)90036-7 (cit. on p. 94).

[Rey93] John C. Reynolds. "The discoveries of continuations." In: *LISP and Symbolic Computation* 6.3 (Nov. 1993), pp. 233–247. ISSN: 1573-0557. DOI: 10.1007/BF01019459 (cit. on p. 192).

[Rib20] Colin Riba. "Monoidal-closed categories of tree automata." en. In: *Mathematical Structures in Computer Science* 30.1 (Jan. 2020), pp. 62–117. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129519000173 (cit. on pp. 22, 49).

[Rin21] Talia Ringer. "Proof Repair." PhD thesis. University of Washington, 2021. URL: https://dependenttyp.es/pdf/thesis.pdf (cit. on p. 23).

[Rog03] James Rogers. "Syntactic Structures as Multi-dimensional Trees." en. In: *Research on Language and Computation* 1.3 (Sept. 2003), pp. 265–305. ISSN: 1572-8706. DOI: 10.1023/A:1024695608419 (cit. on p. 37).

[Roz86] Brigitte Rozoy. "Outils et résultats pour les transducteurs boustrophédons." In: *RAIRO – Theoretical Informatics and Applications* 20.3 (1986), pp. 221–249. DOI: 10.1051/ita/1986200302211 (cit. on p. 91).

[RR97] Simona Ronchi Della Rocca and Luca Roversi. "Lambda Calculus and Intuitionistic Linear Logic." In: *Studia Logica* 59.3 (1997), pp. 417–448. DOI: 10.1023/A:1005092630115 (cit. on p. 154).

[Rub17] Thomas Rubiano. “Implicit Computational Complexity and Compilers.” PhD thesis. Université Sorbonne Paris Cité, Dec. 2017. URL: https://tel.archives-ouvertes.fr/tel-02362912 (cit. on p. 33).

[SA21] Jonathan Sterling and Carlo Angiuli. “Normalization for Cubical Type Theory.” In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–15. DOI: 10.1109/LICS52264.2021.9470719 (cit. on p. 20).

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Translated by Reuben Thomas. Cambridge University Press, 2009. DOI: 10.1017/CBO9781139195218 (cit. on pp. 52, 59).

[Sal06] Sylvain Salvati. “Encoding second order string ACG with Deterministic Tree Walking Transducers.” In: *The 11th conference on Formal Grammar*. Ed. by Shuly Wintner. FG Online Proceedings. Paola Monachesi; Gerald Penn; Giorgio Satta; Shuly Wintner. Malaga, Spain: CSLI Publications, 2006, pp. 143–156. URL: https://web.stanford.edu/group/cslipublications/cslipublications/FG/2006/salvati.pdf (cit. on p. 43).

[Sal09] Sylvain Salvati. “Recognizability in the Simply Typed Lambda-Calculus.” In: *Logic, Language, Information and Computation, 16th International Workshop, WoLLIC 2009, Tokyo, Japan, June 21-24, 2009. Proceedings*. Ed. by Hiroakira Ono, Makoto Kanazawa, and Ruy J. G. B. de Queiroz. Vol. 5514. Lecture Notes in Computer Science. Springer, 2009, pp. 48–60. DOI: 10.1007/978-3-642-02261-6_5 (cit. on p. 37).

[Sal15] Sylvain Salvati. “Lambda-calculus and formal language theory.” Habilitation à diriger des recherches. Université de Bordeaux, Dec. 2015. URL: https://tel.archives-ouvertes.fr/tel-01253426 (cit. on p. 42).

[Sch07] Ulrich Schöpp. “Stratified Bounded Affine Logic for Logarithmic Space.” In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. July 2007, pp. 411–420. DOI: 10.1109/LICS.2007.45 (cit. on pp. 45, 50).

[Sch16] Gabriel Scherer. “Which types have a unique inhabitant? : Focusing on pure program equivalence.” PhD thesis. Paris Diderot University, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01309712 (cit. on pp. 140, 156).

[Sch19] Markus L. Schmid. “Regular Expressions with Backreferences: Polynomial-Time Matching Techniques.” In: *CoRR* abs/1903.05896 (2019). arXiv: 1903.05896 (cit. on p. 13).

[Sch75] Helmut Schwichtenberg. “Definierbare Funktionen im λ-Kalkül mit Typen.” de. In: *Archiv für mathematische Logik und Grundlagenforschung* 17.3 (Sept. 1975), pp. 113–114. ISSN: 1432-0665. DOI: 10.1007/BF02276799 (cit. on p. 39).

[Sco70] Dana Scott. *Outline of a mathematical theory of computation*. Tech. rep. PRG02. Oxford University Computing Laboratory, Nov. 1970, p. 30. URL: https://www.cs.ox.ac.uk/publications/publication3720-abstract.html (cit. on p. 20).

[SE88] Cees F. Slot and Peter van Emde Boas. “The Problem of Space Invariance for Sequential Machines.” In: *Information and Computation* 77.2 (1988), pp. 93–122. DOI: 10.1016/0890-5401(88)90052-1 (cit. on p. 8).

[Sei18] Thomas Seiller. “Interaction Graphs: Non-Deterministic Automata.” In: *ACM Transactions on Computational Logic* 19.3 (Aug. 2018), 21:1–21:24. ISSN: 1529-3785. DOI: 10.1145/3226594 (cit. on pp. 24, 35, 37).

[Sei19] Thomas Seiller. “Interaction Graphs: Exponentials.” In: *Logical Methods in Computer Science* 15.3 (Aug. 2019). DOI: 10.23638/LMCS-15(3:25)2019 (cit. on p. 35).

[Sén07] Géraud Sénizergues. “Sequences of Level 1, 2, 3, ..., k , ..” In: *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*. Ed. by Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov. Vol. 4649. Lecture Notes in Computer Science. Springer, 2007, pp. 24–32. ISBN: 978-3-540-74509-9. DOI: 10.1007/978-3-540-74510-5_6 (cit. on pp. 41, 61).

[Sim90] Imre Simon. “Factorization Forests of Finite Height.” In: *Theoretical Computer Science* 72.1 (1990), pp. 65–94. DOI: 10.1016/0304-3975(90)90047-L (cit. on p. 12).

[SJL65] Richard Edwin Stearns, Hartmanis Juris, and Philip M. Lewis. "Hierarchies of memory limited computations." In: *6th Annual Symposium on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, USA, October 6-8, 1965*. IEEE Computer Society, 1965, pp. 179–190. DOI: 10.1109/FOCS.1965.11 (cit. on p. 10).

[Smi14] Tim Smith. "A Pumping Lemma for Two-Way Finite Transducers." In: *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*. Ed. by Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik. Vol. 8634. Lecture Notes in Computer Science. Springer, 2014, pp. 523–534. DOI: 10.1007/978-3-662-44522-8_44 (cit. on p. 91).

[Sta79] Richard Statman. "The typed λ-calculus is not elementary recursive." In: *Theoretical Computer Science* 9.1 (July 1979), pp. 73–81. ISSN: 0304-3975. DOI: 10.1016/0304-3975(79)90007-0 (cit. on p. 40).

[Ste17] Florian Steinberg. "Computational Complexity Theory for Advanced Function Spaces in Analysis." PhD thesis. Darmstadt: Technische Universität, Feb. 2017. URL: http://tuprints.ulb.tu-darmstadt.de/6096/ (cit. on p. 35).

[Str11] Lutz Straßburger. "Towards a Theory of Proofs of Classical Logic." Habilitation à diriger des recherches. Université Paris-Diderot - Paris VII, 2011. URL: https://tel.archives-ouvertes.fr/tel-00772590 (cit. on p. 22).

[Str18] Howard Straubing. "First-order logic and aperiodic languages: a revisionist history." In: *ACM SIGLOG News* 5.3 (2018), pp. 4–20. DOI: 10.1145/3242953.3242956 (cit. on pp. 11, 31).

[SW16] Sylvain Salvati and Igor Walukiewicz. "Simply typed fixpoint calculus and collapsible pushdown automata." en. In: *Mathematical Structures in Computer Science* 26.7 (Oct. 2016), pp. 1304–1350. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129514000590 (cit. on pp. 38, 41).

[Ter04] Kazushige Terui. "Light Affine Set Theory: A Naive Set Theory of Polynomial Time." In: *Studia Logica* 77.1 (2004), pp. 9–40. DOI: 10.1023/B:STUD.0000034183.33333.6f (cit. on p. 19).

[Ter11] Kazushige Terui. "Computational ludics." In: *Theoretical Computer Science* 412.20 (2011), pp. 2048–2071. DOI: 10.1016/j.tcs.2010.12.026 (cit. on p. 37).

[Ter12] Kazushige Terui. "Semantic Evaluation, Intersection Types and Complexity of Simply Typed Lambda Calculus." In: *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*. 2012, pp. 323–338. ISBN: 978-3-939897-38-5. DOI: 10.4230/LIPIcs.RTA.2012.323 (cit. on pp. 18, 20, 38, 40).

[Til87] Bret Tilson. "Categories as algebra: An essential ingredient in the theory of monoids." en. In: *Journal of Pure and Applied Algebra* 48.1 (Sept. 1987), pp. 83–198. ISSN: 0022-4049. DOI: 10.1016/0022-4049(87)90108-3 (cit. on p. 22).

[Tra61] Boris A. Trakhtenbrot. "Finite automata and the logic of single-place predicates." In: *Doklady Akademii Nauk SSSR* 140 (2 1961), pp. 326–329. URL: http://mi.mathnet.ru/eng/dan25511 (cit. on p. 31).

[Tur37] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem." In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230 (cit. on p. 7).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://homotopytypetheory.org/book (cit. on p. 19).

[Var15] Moshe Y. Vardi. "Why Doesn't ACM Have a SIG for Theoretical Computer Science?" In: *Communications of the ACM* 58.8 (July 2015), p. 5. ISSN: 0001-0782. DOI: 10.1145/2791388 (cit. on p. 8).

[Var82] Moshe Y. Vardi. "The Complexity of Relational Query Languages (Extended Abstract)." In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*. Ed. by Harry R. Lewis, Barbara B. Simons,

Walter A. Burkhard, and Lawrence H. Landweber. ACM, 1982, pp. 137–146. DOI: 10.1145/800070.802186 (cit. on p. 34).

[Vas19] Virginia Vassilevska Williams. "On some fine-grained questions in algorithms and complexity." In: *Proceedings of the International Congress of Mathematicians (ICM 2018)*. World Scientific Publishing, Apr. 2019, pp. 3447–3487. DOI: 10.1142/9789813272880_0188 (cit. on p. 33).

[Wad07] Philip Wadler. "The Girard–Reynolds isomorphism (second edition)." en. In: *Theoretical Computer Science* 375.1-3 (May 2007), pp. 201–226. ISSN: 03043975. DOI: 10.1016/j.tcs.2006.12.042 (cit. on p. 17).

[Wei00] Klaus Weihrauch. *Computable Analysis – An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000. ISBN: 978-3-540-66817-6. DOI: 10.1007/978-3-642-56999-9 (cit. on p. 35).

[Wil19] Gregory Wilsenach. "Symmetric Circuits and Model-Theoretic Logics." PhD thesis. University of Cambridge, 2019. DOI: 10.17863/CAM.44848 (cit. on p. 34).

[Yet90] David N. Yetter. "Quantales and (noncommutative) linear logic." In: *The Journal of Symbolic Logic* 55.1 (Mar. 1990), pp. 41–64. ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2274953 (cit. on pp. 36, 192).

[Zai87] Marek Zaionc. "Word operation definable in the typed λ-calculus." In: *Theoretical Computer Science* 52.1 (Jan. 1987), pp. 1–14. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90077-6 (cit. on pp. 39, 42).

[Zai91] Marek Zaionc. "λ-Definability on free algebras." In: *Annals of Pure and Applied Logic* 51.3 (Mar. 1991), pp. 279–300. ISSN: 0168-0072. DOI: 10.1016/0168-0072(91)90019-I (cit. on p. 39).

[Zei09] Noam Zeilberger. "The Logical Basis of Evaluation Order and Pattern-Matching." PhD thesis. Pittsburgh, PA: Carnegie Mellon University, 2009. URL: http://noamz.org/thesis.pdf (cit. on p. 22).

[ZG15] Noam Zeilberger and Alain Giorgetti. "A correspondence between rooted planar maps and normal planar lambda terms." In: *Logical Methods in Computer Science* 11.3 (Sept. 2015). ISSN: 18605974. DOI: 10.2168/LMCS-11(3:22)2015 (cit. on pp. 16, 25).

[Zhu20] Dmitriy Zhuk. "A Proof of the CSP Dichotomy Conjecture." In: *Journal of the ACM* 67.5 (2020). Journal version of a FOCS'17 paper, 30:1–30:78. DOI: 10.1145/3402029 (cit. on p. 35).

Mes Révérends Pères, mes Lettres n'avaient pas accoutumé de se suivre de si près, ni d'être si étendues. Le peu de temps que j'ai eu a été cause de l'un et de l'autre. Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte.

Blaise Pascal on why this manuscript is awful

Jestem tylko przechodniem [...]
Mam [...] plan albo brak planu

Raz, Dwa, Trzy on doing a PhD