Simply typed convertibility is TOWER-complete even for safe lambda-terms

Lê Thành Dũng Nguyễn

Logical Methods in Computer Science
Volume 20, Issue 3, 2024, pp. 21:1–21:15
https://lmcs.episciences.org/

Submitted May 23, 2023
Published Sep. 04, 2024

SIMPLY TYPED CONVERTIBILITY IS TOWER-COMPLETE EVEN FOR SAFE LAMBDA-TERMS

LÊ THÀNH DŨNG (TITO) NGUYỄN

Laboratoire de l'informatique du parallélisme (LIP), École normale supérieure de Lyon, France
e-mail address: nltd@nguyentito.eu

Abstract. We consider the following decision problem: given two simply typed λ-terms, are they β-convertible? Equivalently, do they have the same normal form? It is famously non-elementary, but the precise complexity – namely Tower-complete – is lesser known. One goal of this short paper is to popularize this fact.

Our original contribution is to show that the problem stays Tower-complete when the two input terms belong to Blum and Ong's safe λ-calculus, a fragment of the simply typed λ-calculus arising from the study of higher-order recursion schemes. Previously, the best known lower bound for this safe β-convertibility problem was PSpace-hardness. Our proof proceeds by reduction from the star-free expression equivalence problem, taking inspiration from the author's work with Pradic on "implicit automata in typed λ-calculi".

These results also hold for $\beta\eta$-convertibility.

1. Introduction

Consider the following *simply typed β-convertibility* problem:

- *Input:* two simply typed λ-terms t and u of the same type – this involves some choices of representation for such terms, that will be discussed later (§1.3).
- *Output:* does $t =_\beta u$ hold? Equivalently – since β-reduction is confluent and terminating in the simply typed λ-calculus – do t and u have the same normal form?

This is, of course, a fundamental decision problem concerning the λ-calculus. For example, the special case when $t : \texttt{Bool} = o \to o \to o$ and $u = \texttt{true} = \lambda x^o.\, \lambda y^o.\, x$ amounts to *evaluating* the result of the program t. (Here, o is the single base type that we work with: our grammar of simple types is $A, B ::= o \mid A \to B$.) Also, proof assistants based on dependent types often need to check whether two terms are *definitionally equal* – which is nothing more than the generalization of the above problem to more sophisticated type theories. For these reasons, there have been many works on deciding convertibility efficiently, such as [Con20].

Key words and phrases: non-elementary complexity, safe λ-calculus.

Thanks to Anupam Das, Damiano Mazza and Noam Zeilberger for many instructive discussions on the topic of complexity of normalization for λ-calculi.

The author was supported by the LambdaComb project (ANR-21-CE48-0017) while working at École Polytechnique and by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "France 2030" operated by the French National Research Agency (ANR).

DOI:10.46298/LMCS-20(3:21)2024

To find out whether $t =_\beta u$, a naive approach is to normalize both t and u and then compare their normal forms. However, it is not always the best course of action: see, for instance, the practical remarks of [Asp17, Section 4] or the use of denotational semantics to decide convertibility in [Ter12].

1.1. **Complexity of β-convertibility.** A well-known result of Statman [Sta79] is that simply typed β-convertibility is *not* in the complexity class Elementary, whose definition we recall now. Let $2^n_0 = n$ and $2^n_{k+1} = 2^{2^n_k}$. A function is in the class k-FExpTime when it can be computed in time bounded by $2^{P(\text{input size})}_k$ for some polynomial P; and the class FElementary of *elementary recursive functions* is the union of all the k-FExpTime classes for $k \in \mathbb{N}$. By Elementary, we refer to the class of decision problems that, when seen as functions returning a boolean, belong to FElementary.

The statement of Statman's theorem can be strengthened as follows. Let $\text{tower}(n) = 2^1_n$. The complexity class Tower consists of the decision problems that can be solved in time $\text{tower}(2^{P(\text{input size})}_k)$ for some polynomial P and some $k \in \mathbb{N}$. A problem is Tower-hard when every problem in Tower reduces to it via many-one FElementary reductions. By the time hierarchy theorem, a Tower-hard problem cannot be in Elementary.

Theorem 1.1. *Simply typed β-convertibility is Tower-complete (that is, it belongs to the class Tower, and is at the same time Tower-hard).*

The hardness part is actually implicit in Statman's proof [Sta79]: it provides a reduction from a problem which is essentially the same as the Tower-complete Higher-Order Quantified Boolean Satisfiability (HOSAT) problem. For a more detailed discussion of this point, see the "related work" paragraph of [CHHM22, Section 3]; let us also mention that Mairson [Mai92] has given a simpler reduction from HOSAT to simply typed β-convertibility. In fact, in his relatively recent article [Sch16] where the complexity class Tower is explictly introduced for the first time, Schmitz observes that many non-elementary lower bounds in the literature can be read as proofs of Tower-hardness – and he mentions the example of simply typed β-convertibility [Sch16, §3.1.2].

As for membership in Tower, it is known to be the "easy part" of Theorem 1.1. In Section 2, we recall some folklore ideas about normalization in the simply typed λ-calculus, and observe that they yield a Tower algorithm for β-convertibility. Section 2 also discusses how the complexity of normalizing a λ-term is controlled by the *degree* and the *order* of the types that it contains – defined respectively as $\deg(o) = \text{ord}(o) = 0$ and

$$\deg(A \to B) = \max(\deg(A), \deg(B)) + 1 \qquad \text{ord}(A \to B) = \max(\text{ord}(A) + 1, \text{ord}(B))$$

In other words, the degree of a type is the height of its syntax tree, while the order is the nesting depth of function arrows to the left.

1.2. **The case of the safe λ-calculus.** Coincidentally, the order of a simple type is also central to the definition of *safe* fragment of the simply typed λ-calculus, even though the motivation that led Blum and Ong [BO09, Blu09] to introduce this fragment were unrelated to computational complexity. (We discuss this background in Section 3.5.2.)

Let us recall briefly the definition of the safe λ-calculus. Let t be a simply typed λ-term in "Church style" – i.e., the variables have type annotations ensuring that the type of each subterm of t is uniquely determined. Then t is *unsafe* if it contains a subterm $u : A$ that

- contains some free variable $x : B$ with $\mathrm{ord}(B) < \mathrm{ord}(A)$
- while being in "unsafe position", that is:
  - either u equals t itself,
  - or u is in argument position: $t = C[v\ u]$ for some term v and one-hole context $C[\cdot]$,
  - or u is applied to some argument ($t = C[u\ v]$) but u itself is not an application.

A safe λ-term is, obviously, a simply typed λ-term that is not unsafe. For example:

$$\lambda f^{(o\to o)\to o}.\ f\ (\lambda x^o.\ f\ (\underbrace{\lambda y^o.\ \overset{\downarrow}{x}}_{\text{argument subterm of type } (o\to o) \rightsquigarrow \text{ order } 1}))\ \text{is unsafe} \qquad \lambda f^{(o\to o)\to o}.\ f\ (\lambda x^o.\ f\ (\lambda y^o.\ y))\ \text{is safe}$$

(the variable x is a free variable of type $o \rightsquigarrow$ order 0)

The precise complexity of deciding β-convertibility on safe λ-terms has been an open problem until now. In their paper introducing the safe λ-calculus, Blum and Ong prove that it is PSpace-hard [BO09, Section 3]. This lower bound is far below Tower, but they argue that both Statman's proof [Sta79] of the hardness part in Theorem 1.1 and its simplification by Mairson [Mai92] fundamentally require unsafe terms. To quote [BO09, §3.1], this "does not rule out the possibility that another non-elementary problem is encodable in the safe lambda calculus". This is precisely what our original contribution here is about:

Theorem 1.2. *β-convertibility in the safe λ-calculus is Tower-complete.*

Membership in Tower follows, of course, from the "easy part" of Theorem 1.1, which is the topic of Section 2. We establish Tower-hardness in Section 3 even for the special case of "homogeneous long-safe" λ-terms of type `Bool` (Theorem 3.1), by reduction from a standard problem in automata theory: *star-free expression equivalence*. As a corollary, this provides an alternative proof of the "hard part" of Theorem 1.1.

1.3. **Some pedantic points.** Traditionally, the simply typed λ-calculus (and its safe fragment) can be presented in two ways (cf. [KSW16]). We already mentioned the intrinsically typed "Church style". By contrast, in "Curry style", a λ-term is given without type annotations, and may satisfy several different typing judgments (e.g. $\vdash \lambda xy.\, x : A \to B \to A$ for all simple types A and B). We might therefore consider two extreme choices when specifying the input for the simply typed β-convertibility problem, that might *a priori* make a difference regarding computational complexity:

(1) All variables in the input terms are fully annotated with their types, and these annotations are counted in the size of the input.
(2) There are no types in the input at all: we are given two untyped terms t and u, with the promise that there exists some simple type A such that $\vdash t : A$ and $\vdash u : A$.

The mathematical equivalence of these two questions already requires some care. It is not the case in general for type theories that type erasure is injective modulo conversion: for example, in the polymorphic λ-calculus (System F), there are Church-style λ-terms of the same type that are not convertible even though the underlying untyped terms are β-convertible.[1] However, this does not happen in the simply typed λ-calculus, because:

Fact 1.3. A Church-style simply typed λ-term *in normal form*[2] can be reconstructed uniquely from its type and its underlying untyped term.

[1] In System F, the latter condition is called "Strachey equivalence" [PA93, §2.2], see also [JT18].

[2] The normal form assumption is required to exclude counter-examples such as $(\lambda xy.\, y)\,(\lambda z.\, z) : A \to A$ where, even when a specific A is fixed, x can be given any type of the form $B \to B$.

Complexity-wise, (1) is easier than (2): simply erasing all types in the input gives a trivial reduction. Conversely, type inference for the simply typed λ-calculus can be performed in linear time, using first-order unification (see e.g. [Wan87]) – this also works for inferring a common simple type for a pair (t, u) of λ-terms, by adding an equation to the unification problem. A solution to this unification problem – of size $O(|t| + |u|)$ – also gives us the type annotations to add to t and u; we can ask a unification algorithm to return a unifier that has linear size using a representation of the syntax trees as directed acyclic graphs (in order to share subtrees), cf. [BN98, Sections 4.8 and 4.9]. Note that this guarantees that $\mathrm{ord}(A) \leqslant \deg(A) = O(|t| + |u|)$ for any type A among the computed annotations.

Thus, there is a linear time reduction from (2) to the variant (1') of (1) where such shared representations are allowed for the types. In turn, (1') reduces to (1) in exponential time by unfolding the syntax trees of the types. Since 1-FExpTime $\subset$ FElementary, this does not make a difference regarding membership in or hardness for Tower.

1.4. **The η rule.** Theorems 1.1 and 1.2 also hold for $\beta\eta$-conversion.

For the hardness part, this is because we prove it for the problem restricted to λ-terms of type `Bool`, and $\beta\eta$-convertibility coincides with β-convertibility at type `Bool`. (Indeed, for every $t : \texttt{Bool}$, either $t =_\beta \texttt{true}$ or $t =_\beta \texttt{false} = \lambda x^o.\, \lambda y^o.\, y$, and $\texttt{true} \neq_{\beta\eta} \texttt{false}$.)

For the complexity upper bound, note that Section 2 actually gives an algorithm for computing the β-normal form t' of a simply typed λ-term t. Once we have t', applying η-reductions takes time $|t'|^{O(1)}$ until a $\beta\eta$-normal form[3] is reached. And testing whether two terms are $\beta\eta$-convertible can be done by comparing their $\beta\eta$-normal forms for equality[4], thanks to the confluence of $\beta\eta$-reduction on simply typed λ-terms of a fixed type – a fact shown in [Geu92] which is a bit more subtle than one could expect. This also means that we can invoke Fact 1.3 again to conclude that the Church-style and Curry-style variants of the $\beta\eta$-convertibility problem are equivalent.

But this does not work anymore when we add sum types endowed with the "strong" η rule: in this setting, "there can be no λ-calculus with sums *à la* Curry" [MS15, §I(a)] because untyped $\beta\eta$-conversion is inconsistent (that is, it equates all λ-terms). In fact, decidability of $\beta\eta$-convertibility for a simply typed λ-calculus with sums and the empty type, using typed conversion rules, has only been proved relatively recently [Sch17]. As future work, it could be interesting to know whether this problem is still in Tower.

2. Simply typed β-convertibility is in Tower

2.1. **A simple Tower normalization algorithm using the degree.** Let us first define the *type of a redex* $\left(\lambda x^A.\, t\right) u$ in a Church-style simply typed λ-term as $A \to B$ where B is the type of t (so $A \to B$ is also the type of $\lambda x^A.\, t$). The *degree of a redex* is the degree of its type. The idea of using redex degrees to get a weak normalization proof for the simply typed λ-calculus goes back at least to Turing, see [BS23, Section 1] for an explanation and historical discussion. Here, we use a variation on this idea based on the notion of *parallel reduction*, which comes from the Tait/Martin-Löf proof of confluence for untyped β-reduction.

[3]Also known as a β-normal η-*short* form. In many other contexts, β-normal η-*long* forms are more useful, but here η-reduction is more convenient than η-expansion because it gives us a terminating rewriting system.

[4]In this paper, we always consider λ-terms modulo α-renaming, so an equality test that takes concrete syntax trees as input has to take α-renaming into account.

Definition 2.1 [Tak95, p. 121]**.** The parallel reduction $t^\star$ of a λ-term t is, inductively:

$$x^\star = x \qquad (\lambda x.\, t)^\star = \lambda x.\, t^\star \qquad (t\, u)^\star = \begin{cases} (t')^\star\{x := u^\star\} & \text{when } t = \lambda x.\, t' \text{ for some } t' \\ t^\star\, u^\star & \text{otherwise} \end{cases}$$

This definition on untyped λ-terms can be extended to Church-style simply typed λ-terms in the obvious way.

It is immediate that $t \longrightarrow_\beta^* t^\star$. Furthermore:

Proposition 2.2 [AL13, Appendix]**.** *Let t be a simply typed λ-term whose redexes have degree at most $d \in \mathbb{N}$. Then:*

- *The redexes of $t^\star$ have degree at most $d - 1$.*
- *As a consequence, the normal form of t can be reached in d parallel reductions. In other words, let $t^{\star 0} = t$ and $t^{\star(n+1)} = (t^{\star n})^\star$; then $t^{\star d}$ is the normal form of t.*

This gives us a simple algorithm to compute the normal form of a simply typed λ-term. In fact, since parallel reduction does not inspect types, we may also apply it to simply *typable* terms, to solve the Curry-style version of the β-convertibility problem (cf. §1.3).

We consider the following measures on an untyped λ-term t to get a complexity bound:

- $\mathrm{height}(t)$ denotes the height of its syntax tree;
- its *size* $|t|$ is the number of nodes of its syntax tree, including the leaves (variables):

$$|x| = 1 \qquad |\lambda x.\, t| = 1 + |t| \qquad |t\, u| = 1 + |t| + |u|$$

Fact 2.3. For any untyped λ-term t, we have $\mathrm{height}(t^\star) \leqslant |t| \leqslant 2^{\mathrm{height}(t)}$.

(We have not managed to find this statement in the literature, but it is probably not new.)

Proof. The first inequality can be proved by structural induction on Definition 2.1, using

$$\mathrm{height}((t')^\star\{x := u^\star\}) \leqslant \mathrm{height}((t')^\star) + \mathrm{height}(u^\star)$$

while the second inequality is due to the fact that each node in the syntax tree of a λ-term has at most two children. □

As a consequence, $|t^\star| \leqslant 2^{|t|}$ for any λ-term t. Putting all this together, we get:

Theorem 2.4. *There is an algorithm that takes as input a simply typable λ-term t and returns its normal form in time at most $2_d^{P(|t|)}$, where d is the maximum redex degree in some typing of t (not given as input) and P is some polynomial independent from t and d.*

Proof. The algorithm iterates parallel reductions until it reaches a normal form. This takes at most d steps; for each step $n \in \{1, \dots, d\}$, computing $t^{\star n}$ can be done by feeding the input $t^{\star(n-1)}$ – which has size at most $2_{n-1}^{|t|}$ – to a naive 1-FExpTime procedure. □

Given two simply typable λ-terms, we can compute their normal forms and compare them to decide whether they are β-convertible. Together with the considerations from §1.3, this establishes the "membership in Tower" part of Theorem 1.1. (Again, we do not make any claim to originality concerning the material in this section; it has been included for expository purposes.)

2.2. **Finer bounds with the order.** Actually, a Tower algorithm for normalization could also be obtained just by applying successive (non-parallel) β-reductions. The complexity then depends on the length of reduction sequences for simply typed λ-terms, for which bounds are known [Sch91, Bec01]. But note that these bounds are stated using the *order* of the types in a λ-term, not their degree!

Remark 2.5. The paper [Bec01] uses different terminology: "level" refers to what we call the order, while "degree" refers to the maximum order of the types appearing in a term. See also [AKST19] for a probabilistic analysis of β-reduction length in the simply typed λ-calculus, and [Sin22, Section 4.4] for something similar in the linear λ-calculus.

For any simple type A, we have $\mathrm{ord}(A) \leqslant \deg(A)$ and the gap can be arbitrarily large:

$$\deg(\underbrace{o \to \cdots \to o}_{n \text{ times}} \to o) = n \qquad \mathrm{ord}(\underbrace{o \to \cdots \to o}_{n \text{ times}} \to o) = 1$$

While using the degree seems to be the easier way to devise a normalization algorithm together with a proof of a Tower complexity bound, it seems that the order is the parameter that truly controls complexity. As further evidence towards this (cf. [Ngu21, §1.3.6] for a longer discussion of these three items):

- the normalization problem for simply typed λ-terms of type `Bool` containing only subterms whose types have *order at most* $2k+2$ is k-ExpTime-complete [Ter12];
- in fact, k-ExpTime can be characterized as the predicates expressed by simply typed λ-terms of *order at most* $2k+4$ using a certain input-output convention [HK96], refining an earlier characterization of Elementary in the simply typed λ-calculus with no constraint on the order [HKM96];
- the "call-by-name translation" ($A \to B = !A \multimap B$) from the simply typed λ-calculus to propositional linear logic (LL) maps the order to the nesting depth of the exponential modality '!', and '!' is generally considered to be the main source of complexity in LL (see [GRV09] for a connection between the exponential depth in propositional LL and the kind of stratification used in LL-based implicit computational complexity).

3. Safe β-convertibility is Tower-hard

As announced in Section 1.2 of the introduction, our goal here is to show:

Theorem 3.1. *Given a homogeneous long-safe λ-term t of type* `Bool`*, it is Tower-hard to decide whether $t =_\beta$* `true`*.*

Recall that the type `Bool` is defined as $o \to o \to o$. It has exactly two inhabitants in normal form: `true` $= \lambda xy.\, x$ and `false` $= \lambda xy.\, y$. They are both safe [BO09, Remark 2.5(ii)], and even homogeneous long-safe (as defined in the next subsection).

3.1. **Homogeneity and long-safety.** Before proceeding with the proof of Theorem 3.1, let us explain the additional restrictions – beyond the safety condition recalled in Section 1.2 – imposed on the input terms. We chose to prove hardness in presence of these restrictions because they are natural in the context of higher-order recursion schemes (cf. §3.5.2).

First, *homogeneity* is a simple syntactic criterion on types.

Definition 3.2. A simple type $A_1 \to \cdots \to A_n \to o$ (for $n \geq 0$) is homogeneous when each of the A_i is itself homogeneous ($i \in \{1, \ldots, n\}$) and $\mathrm{ord}(A_1) \geq \cdots \geq \mathrm{ord}(A_n)$.

The second restriction is *long-safety*. In Church style, it is the same as the notion of safety in §1.2 except more subterms of a simply typed λ-term are considered to be in "unsafe position": if u is not a λ-abstraction, then it is in unsafe position in $t = C[\lambda x.\, u]$. Equivalently, in Curry style:

Definition 3.3 [BO09, Def. 1.21]**.** The typing rules of the long-safe λ-calculus are

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Theta \vdash t : A}{\Theta' \vdash t : A}\ (\Theta \subset \Theta')$$

$$\frac{\Theta \vdash t : A_1 \to \ldots \to A_n \to B \quad \Theta \vdash u_1 : A_1 \ \ldots\ \Theta \vdash u_n : A_n}{\Theta \vdash t\, u_1\, \ldots\, u_n : B} \text{ when } \mathrm{ord}(B) \leqslant \mathrm{inford}(\Theta)$$

$$\frac{\Theta,\, x_1 : A_1,\, \ldots,\, x_n : A_n \vdash t : B}{\Theta \vdash \lambda x_1 \ldots x_n.\, t : A_1 \to \cdots \to A_n \to B} \text{ when } \mathrm{ord}(A_1 \to \cdots \to A_n \to B) \leqslant \mathrm{inford}(\Theta)$$

where $\mathrm{inford}(\Theta) = \inf_{(y:C)\in\Theta} \mathrm{ord}(C)$ with the usual convention $\inf(\varnothing) = +\infty$.

For the rest of Section 3, we only work with Curry-style typed λ-terms. This is because it allows our constructions to be carried out in polynomial time. If we had to build Church-style terms, writing out the type annotations would take exponential time, as explained in §1.3 (though this is harmless in a TOWER-hardness proof; the point is just to have a more precise statement).

Definition 3.4. For a λ-term t, a simple type A and a context Γ, we write $\Gamma \vdash_{\mathsf{hls}} t : A$ when there is a derivation *using only homogeneous types* of $\Gamma \vdash t : A$ in the above type system. In that case we say that t is *homogeneous long-safe*.

3.2. **Star-free expressions.** As we said in the introduction, our TOWER-hardness proof proceeds by a reduction from an automata-theoretic problem, which we recall now.

Definition 3.5. A *star-free expression* over a finite alphabet Σ is a regular expression without the Kleene star, but with complementation, defining a language $[\![E]\!] \subseteq \Sigma^*$:

$$E, E' ::= \varnothing \mid \overbrace{\varepsilon}^{\text{empty string}} \mid \underbrace{a}_{\text{letter } a\in\Sigma} \mid E \cup E' \mid \overbrace{E \cdot E'}^{\text{concatenation}} \mid \underbrace{\neg E}_{\text{complement}}$$

The *star-free equivalence problem* consists in deciding whether two star-free expressions (over the same alphabet) denote the same language: given E and E', does $[\![E]\!] = [\![E']\!]$ hold?

For instance, over the alphabet $\Sigma = \{a, b, c\}$, we have the equivalence

$$[\![a \cdot \neg\varnothing \cup b \cdot \neg\varnothing]\!] = [\![\neg(\varepsilon \cup c \cdot \neg\varnothing)]\!] = \{w \in \Sigma^* \mid w \text{ starts with an } a \text{ or a } b\}$$

Schmitz's paper introducing TOWER presents star-free equivalence as a typical example of a TOWER-complete problem [Sch16, §3.1] (rephrasing a hardness result proved by Meyer and Stockmeyer, cf. [Sto74, §4.2]). Furthermore, equivalence reduces to emptiness: given two expressions E and F, the language denoted by $\neg(\neg E \cup F) \cup \neg(E \cup \neg F)$ is empty if and only if E and F are equivalent. And to test whether $[\![E]\!] = \varnothing$, it is well known that it suffices to examine words of length at most $\mathrm{tower}(|E| - 1)$:

Proposition 3.6 (cf. e.g. [Sto74, proof of Prop. 4.25])**.** *If a star-free expression of size $n+1$ denotes a non-empty language L, then L contains a word of length at most* $\mathrm{tower}(n)$.

(Further remarks on the above proposition and on the complexity of star-free equivalence are given at the end in Section 3.5.1.) Altogether, this discussion leads us to conclude that:

Lemma 3.7. *The following problem is* Tower*-hard:*
- Input: *a star-free expression E and a natural number n given in unary.*
- Output: *whether or not $[\![E]\!]$ contains a word of length at most* $\mathrm{tower}(n)$.

3.3. **From star-free expressions to safe λ-terms.** We shall represent (star-free) languages as functions from strings to booleans. We already have a coding of booleans, that is involved in the statement of Theorem 3.1. For strings, we use the standard *Church encodings*.

Definition 3.8 [BB85, §4]**.** The type $\mathtt{Str}_\Sigma = \overbrace{(o \to o) \to \cdots \to (o \to o)}^{|\Sigma| \text{ times}} \to o \to o$ is the type of so-called *Church encodings* of strings over the finite alphabet Σ. In the case of a unary alphabet, the type of *Church numerals* is $\mathtt{Nat} = \mathtt{Str}_{\{\mathrm{I}\}} = (o \to o) \to o \to o$.

The Church encoding of a word w over an ordered alphabet $\Sigma = \{c_1, \ldots, c_n\}$ (resp. of a number $n \in \mathbb{N}$) is $\overline{w} = \lambda f_{c_1} \ldots f_{c_n} x.\ \underbrace{f_{w[1]}\ (\ldots\ (f_{w[n]}}_{w[i] \text{ refers to the } i\text{-th letter of } w}\ x)) : \mathtt{Str}_\Sigma$ (resp. $\overline{n} = \underbrace{\overline{\mathrm{I} \ldots \mathrm{I}}}_{n \text{ times}} : \mathtt{Nat}$).

For any finite alphabet Σ and string $w \in \Sigma^*$, one can check that the type $\mathtt{Str}_\Sigma$ is homogeneous and that $\vdash_{\mathsf{hls}} \overline{w} : \mathtt{Str}_\Sigma$ (the safety of $\overline{w}$ is already used in [BO09, §2.2]). Next, to encode languages, a fundamental remark is that if A and B are homogeneous simple types and $\vdash_{\mathsf{hls}} t : A$, then $\vdash_{\mathsf{hls}} t : A[B]$, where:

Notation 3.9. $A[B]$ denotes the substitution of all occurrences of the base type o in the simple type A by the simple type B.

This allows us to use a λ-term t such that $\vdash_{\mathsf{hls}} t : \mathtt{Str}_\Sigma[B] \to \mathtt{Bool}$ to represent the language $\{w \in \Sigma^* \mid t\ \overline{w} =_\beta \mathtt{true}\}$ – this makes sense since $\vdash_{\mathsf{hls}} t\ \overline{w} : \mathtt{Bool}$. The key lemma in our proof of Theorem 3.1 is that star-free expressions can be efficiently translated to such representations by (Curry-style) homogeneous safe λ-terms.

Lemma 3.10. *A star-free expression E over a finite alphabet Σ can be turned in polynomial time into an λ-term t_E such that*
- $\vdash_{\mathsf{hls}} t_E : \mathtt{Str}_\Sigma[A_E] \to \mathtt{Bool}$ *for some simple type A_E;*
- *for any $w \in \Sigma^*$, we have $t_E\ \overline{w} =_\beta \mathtt{true}$ if and only if $w \in [\![E]\!]$.*

(The type A_E is not part of the output of the polynomial-time algorithm taking E and Σ as input. In fact, the size of A_E may be exponential in the size of E.)

To prove Lemma 3.10, we first note that boolean operations can be implemented by

$$\mathtt{and} = \lambda b_1 b_2 xy.\ b_1\ (b_2\ x\ y)\ y \qquad \mathtt{or} = \lambda b_1 b_2 xy.\ b_1\ x\ (b_2\ x\ y) \qquad \mathtt{not} = \lambda bxy.\ b\ y\ x$$

These λ-terms satisfy $\vdash_{\mathsf{hls}} \mathtt{and}, \mathtt{or} : \mathtt{Bool} \to \mathtt{Bool} \to \mathtt{Bool}$ and $\vdash_{\mathsf{hls}} \mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$. Furthermore, we can implement concatenation on strings (cf. [BO09, Theorem 2.8]) with

$$\mathtt{cat} = \lambda s_1 s_2 f_1 \ldots f_{|\Sigma|} x.\ s_1\ f_1\ \ldots\ f_{|\Sigma|}\ (s_2\ f_1\ \ldots\ f_{|\Sigma|}\ x)$$

which is a homogeneous long-safe λ-term of type $\mathtt{Str}_\Sigma \to \mathtt{Str}_\Sigma \to \mathtt{Str}_\Sigma$. This generalizes to k-fold concatenation: $\vdash_{\mathsf{hls}} \mathtt{cat}_k : \underbrace{\mathtt{Str}_\Sigma \to \cdots \to \mathtt{Str}_\Sigma}_{k \text{ times}} \to \mathtt{Str}_\Sigma$.

The next step in our proof is the following lemma, which is reused in Section 3.4.

Lemma 3.11. *For every finite alphabet Σ, one can build in time $|\Sigma|^{O(1)}$ a λ-term $\mathtt{any}_\Sigma$ such that, for any homogeneous type A, there is some $F_{\mathtt{any}}(A)$ such that*

$$\vdash_{\mathsf{hls}} \mathtt{any}_\Sigma : \mathtt{Str}_{\Sigma\cup\{\#\}}[F_{\mathtt{any}}(A)] \to (\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}) \to \mathtt{Bool}$$

and for every list of words $w_0, \dots, w_n \in \Sigma^*$ and every λ-term $t : \mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$,

$$\mathtt{any}_\Sigma\ \overline{w_0\#\dots\#w_n}\ t =_\beta \mathtt{true} \quad\iff\quad \exists i \in \{0,\dots,n\}.\ t\,\overline{w_i} =_\beta \mathtt{true}$$

Proof. Let $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$. Take $F_{\mathtt{any}}(A) = \mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$ and define $\mathtt{any}_\Sigma$ as

$$\lambda sp.\ s\ \overbrace{u_1\ \dots\ u_{|\Sigma|}}^{\text{correspond to letters in } \Sigma}\ \underbrace{(\lambda fx.\ \mathtt{or}\ (p\ x)\ (f\ \overline{\varepsilon}))}_{\text{corresponds to } \#}\ p\ \overline{\varepsilon} \quad \text{where } u_i = \lambda fx.\ f\ (\mathtt{cat}\ x\ \overline{c_i})$$

with $x, \overline{\varepsilon} : \mathtt{Str}_\Sigma[A]$, $p, f : \mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$, $u_i : (\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}) \to \mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$ and $s : \mathtt{Str}_{\Sigma\cup\{\#\}}[\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}]$ in the typing derivation for $\mathtt{any}_\Sigma$.

Explanation: computing $\mathtt{any}_\Sigma\ \overline{w_0\#\dots\#w_n}\ t$ performs a "right fold" over the input with an accumulator of type $\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$, representing a language $L(\sigma) \subseteq \Sigma^*$ where σ is the suffix processed up until this point of the computation. The idea is that

$$u \in L(v\#w_m\#\dots\#w_n) \quad\iff\quad t\,\overline{uv} =_\beta \mathtt{true} \vee \exists i \geq m.\ t\,\overline{w_i} =_\beta \mathtt{true}$$

This makes the condition that we want to check equivalent to $\varepsilon \in L(w_0\#\dots\#w_n)$. □

Proof of Lemma 3.10. By induction on the expression, leading to a recursive algorithm. We also want the algorithm to return $\mathrm{ord}(A_E)$ on the input E, because this information on subexpressions is used in two cases to make sure that the term we build admits a homogeneous type. It will be clear that in every case, $\mathrm{ord}(A_E)$ can be recursively computed and is bounded by $O(|E|)$.

- We take $A_\varnothing = o$ and $t_\varnothing = \lambda s.\ \mathtt{false} : \mathtt{Str}_\Sigma \to \mathtt{Bool}$.
- In the simply typed λ-calculus, testing for the empty word can be done with type $\mathtt{Str}_\Sigma \to \mathtt{Bool}$, but this is not possible with safe λ-terms as discussed in [BO09, Section 2]. We use instead $A_\varepsilon = \mathtt{Bool}$ and

  $$t_\varepsilon = \lambda s.\ s\ (\lambda x.\ \mathtt{false})\ \dots\ (\lambda x.\ \mathtt{false})\ \mathtt{true}$$

- To test whether the word consists of a single letter, say, the first one in the alphabet Σ (call it c_1), we use $A_{c_1} = o \to o \to o \to o$ and define t_{c_1} to be a λ-term corresponding to a deterministic finite automaton that recognizes the language $\{c_1\}$, namely:

  $$\lambda sxy.\ s\ \overbrace{(\lambda qz_0z_1z_2.\ q\ z_1\ z_2\ z_2)}^{\text{represents the transition } \delta(c_1,q_0)=q_1,\ \delta(c_1,q_1)=\delta(c_1,q_2)=q_2}\ \underbrace{(\lambda q.\ q_2)\ \dots\ (\lambda q.\ q_2)}_{|\Sigma|-1 \text{ times}}\ q_0\ \overbrace{y\ x\ y}^{\text{accepting states} = \{q_1\}} \quad \text{where } \underbrace{q_i = \lambda z_0z_1z_2.\ z_i}_{\text{the 3 states of the DFA (of type } A_{c_1})}$$

- Complementation is trivially implemented by post-composing with $\mathtt{not}$.
- To handle a union $E \cup F$, we first check whether $\mathrm{ord}(A_E) \geq \mathrm{ord}(A_F)$. If that is the case, we take $A_{E\cup F} = \mathtt{Str}_\Sigma[A_E] \to \mathtt{Str}_\Sigma[A_F] \to \mathtt{Bool}$ and

  $$t_{E\cup F} = \lambda s.\ s\ \underbrace{\dots\ (\lambda kxy.\ k\ (\mathtt{cat}\ x\ \overline{c_i})\ (\mathtt{cat}\ y\ \overline{c_i}))\ \dots}_{\text{for each } c_i \in \Sigma}\ (\lambda xy.\ \mathtt{or}\ (t_E\ x)\ (t_F\ y))\ \overline{\varepsilon}\ \overline{\varepsilon}$$

  with $k : \mathtt{Str}_\Sigma[A_E] \to \mathtt{Str}_\Sigma[A_F] \to \mathtt{Bool}$, $x : \mathtt{Str}_\Sigma[A_E]$ and $y : \mathtt{Str}_\Sigma[A_F]$ in the typing derivation; the two occurrences of $\overline{\varepsilon}$ at the end must be given the different types $\mathtt{Str}_\Sigma[A_E]$ and $\mathtt{Str}_\Sigma[A_F]$ respectively.

This can be seen as a continuation-passing-style transformation of the following procedure using pair types (which we do not have here): perform a "left fold" over the input string with an accumulator of type $\mathtt{Str}_\Sigma[A_E] \times \mathtt{Str}_\Sigma[A_F]$; for each input letter c, apply the function $(w, w') \in \Sigma^* \mapsto (wc, w'c)$, so that the result of the fold is two copies of the input string with different types; finally, check on (w, w') whether $w \in [\![E]\!]$ or $w' \in [\![F]\!]$.

When $\mathrm{ord}(A_E) < \mathrm{ord}(A_F)$, we take $t_{E\cup F} = t_{F\cup E}$. Observe that t_E and t_F appear only once in $t_{E\cup F}$ – this is true in every recursive case, ensuring that t_E has size $O(|E|)$.

- The remaining case, concatenation, is the most delicate.

  First, we introduce a new letter $\Box \notin \Sigma$ and build a λ-term $t'_{E,F}$ that computes the language $[\![(E\Box)^*F]\!]$. When $\mathrm{ord}(A_E) \geq \mathrm{ord}(A_F)$, we get

$$\vdash_{\mathsf{hls}} t'_{E,F} : \mathtt{Str}_{\Sigma\cup\{\Box\}}[\mathtt{Str}_\Sigma[A_E] \to \mathtt{Str}_\Sigma[A_F] \to \mathtt{Bool}] \to \mathtt{Bool}$$

  by taking $t'_{E,F} = \lambda s.\ s\ v_{c_1}\ \dots\ v_{c_{|\Sigma|}}\ v_\Box\ u\ \overline{\varepsilon}\ \overline{\varepsilon}$ (for $\Sigma = \{c_1, \dots, c_{|\Sigma|}\}$) where

$$v_c = \lambda fxy.\ f\ (\mathtt{cat}\ x\ \overline{c})\ (\mathtt{cat}\ y\ \overline{c}) \qquad v_\Box = \lambda fxy.\ \mathtt{and}\ (t_E\ x)\ (f\ \overline{\varepsilon}\ \overline{\varepsilon}) \qquad u = \lambda xy.\ t_F\ y$$

  Similarly to the proof of Lemma 3.11, $t'_{E,F}$ performs a "right fold" on its input string (over the alphabet $\Sigma \cup \{\Box\}$), whose accumulator $f : \mathtt{Str}_\Sigma[A_E] \to \mathtt{Str}_\Sigma[A_F] \to \mathtt{Bool}$ represents a language (over Σ) – except that, analogously to the case $E \cup F$ above, this representation must be fed two copies $x : \mathtt{Str}_\Sigma[A_E]$ and $y : \mathtt{Str}_\Sigma[A_F]$ of the same string. We get

$$\{w \in \Sigma^* \mid w\sigma \in [\![(E\Box)^*F]\!]\}$$

  after reading the input suffix $\sigma \in (\Sigma \cup \{\Box\})^*$; therefore, once the whole input string has been traversed, we just need to check that the final accumulator contains ε.

  If $\mathrm{ord}(A_E) < \mathrm{ord}(A_F)$, to ensure homogeneity, we replace in the definition of $t'_{E,F}$

$$v_\Box \text{ by } \lambda kxy.\ \mathtt{and}\ (t_E\ y)\ (k\ \overline{\varepsilon}\ \overline{\varepsilon}) \quad \text{and} \quad u \text{ by } \lambda xy.\ t_F\ x$$

  leading to $\vdash_{\mathsf{hls}} t'_{E,F} : \mathtt{Str}_{\Sigma\cup\{\Box\}}[\mathtt{Str}_\Sigma[A_F] \to \mathtt{Str}_\Sigma[A_E] \to \mathtt{Bool}] \to \mathtt{Bool}$.

  Then, the λ-term that we want is $t_{E\cdot F} = \lambda s.\ \overbrace{\mathtt{any}_{\Sigma\cup\{\Box\}}}^{\text{Lemma 3.11 above}}\ (\overbrace{\mathtt{split}_\Sigma}^{\text{Lemma 3.12 below}}\ s)\ t'_{E,F}$ where the role of $t'_{E,F}$ is to distinguish, among the strings containing exactly one '$\Box$', those that belong to $[\![E\Box F]\!]$ – indeed, $[\![(E\Box)^*F]\!] \cap \Sigma^*\Box\Sigma^* = [\![E\Box F]\!]$. □

We isolate the following part of the above proof to serve as an example of independent interest (cf. §3.5.3) of a string-to-string function expressed in the safe λ-calculus.

Lemma 3.12. *For every finite alphabet Σ, one can build in time $|\Sigma|^{O(1)}$ a λ-term $\mathtt{split}_\Sigma$ such that, for some type $A_{\mathtt{split}}$,*

$$\vdash_{\mathsf{hls}} \mathtt{split}_\Sigma : \mathtt{Str}_\Sigma[A_{\mathtt{split}}] \to \mathtt{Str}_\Gamma \quad \textit{where } \Gamma = \Sigma \cup \{\Box, \#\}$$

and for every list of letters $a_1, \dots, a_n \in \Sigma$,

$$\mathtt{split}_\Sigma\ \overline{a_1 \dots a_n} \quad =_\beta \quad \overline{\Box a_1 \dots a_n \# a_1 \Box a_2 \dots a_n \# \dots \# a_1 \dots a_{n-1} \Box a_n \# a_1 \dots a_n \Box}$$

Proof. As in the case of unions in the proof of Lemma 3.10, the idea is to start from a left-to-right procedure using a pair type, and then apply a continuation-passing-style transformation. For $p = a_1 \dots a_m \in \Sigma^*$ (where $a_i \in \Sigma$), let us consider the maps

$$\begin{array}{ll} X_p : \mathbb{N} \to \Sigma^* & F_{a_1\dots a_m} : \Gamma^* \to \Gamma^* \\ \quad n \mapsto p & \qquad\qquad y \mapsto \Box a_1 \dots a_m y a_1 \Box a_2 \dots a_m y \dots y a_1 \dots a_{m-1} \Box a_m y \end{array}$$

In particular, $F_\varepsilon(y) = \varepsilon$ and $F_a(y) = \Box ay$ for $a \in \Sigma$. The maps X_p are constant; the role of their dummy argument is to increase the order of an associated variable in the λ-term defined below so that it satisfies the safety condition – $\mathrm{ord}(\mathtt{Nat} \to \mathtt{Str}_\Gamma) = 3 > 2 = \mathrm{ord}(\mathtt{Str}_\Gamma)$.

For $p \in \Sigma^*$ and $c \in \Sigma$, we have the inductive equations

$$X_{pc}(n) = X_p(0) \cdot c \qquad F_{pc}(y) = F_p(cy) \cdot X_p(0) \cdot \Box c \cdot y$$

which we translate into the λ-term

$$v_c = \lambda kxf.\ k\ (\lambda n.\ \mathtt{cat}\ (x\ \overline{0})\ \overline{c})\ \underbrace{(\lambda y.\ \mathtt{cat}_4\ (f\ (\mathtt{cat}\ \overline{c}\ y))\ (x\ \overline{0})\ \overline{\Box c}\ y)}_{x \text{ appears free in this subterm of order 3} \rightsquigarrow \text{ the } \mathtt{Nat} \to \mathtt{Str}_\Gamma \text{ trick ensures (long-)safety}} \quad \text{for } c \in \Sigma$$

with $k : (\mathtt{Nat} \to \mathtt{Str}_\Gamma) \to (\mathtt{Str}_\Gamma \to \mathtt{Str}_\Gamma) \to \mathtt{Str}_\Gamma$, $x : \mathtt{Nat} \to \mathtt{Str}_\Gamma$, $f : \mathtt{Str}_\Gamma \to \mathtt{Str}_\Gamma$, $n : \mathtt{Nat}$ and $y : \mathtt{Str}_\Gamma$. Finally, we take

$$\mathtt{split} = \lambda s.\ s\ v_{c_1}\ \ldots\ v_{c_{|\Sigma|}}\ (\lambda xf.\ \mathtt{cat}_3\ (f\ \overline{\#})\ (x\ \overline{0})\ \overline{\Box})\ (\lambda n.\ \overline{\varepsilon})\ (\lambda y.\ \overline{\varepsilon})$$

– thus, $A_{\mathtt{split}}$ is the aforementioned type of k. □

3.4. **Proof of Theorem 3.1.** We now have all the ingredients to reduce the TOWER-hard decision problem of Lemma 3.7 to safe β-convertibility. This reduction is performed by the following lemma, which therefore immediately entails our main theorem.

Lemma 3.13. *Given a star-free expression E over a finite alphabet Σ, and $n \in \mathbb{N}$ written in unary, one can build in polynomial time a λ-term b_E such that $\vdash_{\mathsf{hls}} b_E : \mathtt{Bool}$ and*

$$b_E =_\beta \mathtt{true} \quad \iff \quad [\![E]\!] \cap \Sigma^{\leqslant \mathrm{tower}(n)} \neq \varnothing$$

Proof. The key is to build a λ-term $\mathtt{enum}_\Sigma$ such that:

- $\vdash_{\mathsf{hls}} \mathtt{enum}_\Sigma : \mathtt{Nat}[A_{\mathtt{enum}}] \to \mathtt{Str}_{\Sigma \cup \{\#\}}$ for some type $A_{\mathtt{enum}}$;
- for any $m \in \mathbb{N}$, the application $\mathtt{enum}\ \overline{m}$ reduces to the Church encoding of a #-separated list containing all words in Σ^* of length at most m.

We define it for $\Sigma = \{c_1, \ldots, c_{|\Sigma|}\}$ in such a way that $\mathtt{enum}_\Sigma\ \overline{N} =_\beta \overline{f_N(\#)}$ for $N \in \mathbb{N}$, where the sequence of functions $f_n \colon (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ is inductively defined by

$$f_0(x) = x \qquad f_{n+1}(x) = x \cdot f_n(c_1 x) \cdot \ldots \cdot f_n(c_{|\Sigma|} x)$$

e.g. for $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ we have $f_1(x) = x\mathtt{0}x\mathtt{1}x$ and $f_2(x) = x\mathtt{0}x\mathtt{00}x\mathtt{10}x\mathtt{1}x\mathtt{01}x\mathtt{11}x$. Observe that the string $f_2(\#)$ is a #-separated representation of the list $[\varepsilon, \mathtt{0}, \mathtt{00}, \mathtt{10}, \mathtt{1}, \mathtt{01}, \mathtt{11}, \varepsilon]$ that indeed contains all words of length at most 2 over Σ (with one redundancy). To implement this, take $A_{\mathtt{enum}} = \mathtt{Str}_\Sigma \to \mathtt{Str}_\Sigma$ and

$$\mathtt{enum}_\Sigma = \lambda n.\ n\ (\lambda fs.\ \mathtt{cat}_{|\Sigma|+1}\ s\ (f\ (\mathtt{cat}\ \overline{c_1}\ s))\ \ldots\ (f\ (\mathtt{cat}\ \overline{c_{|\Sigma|}}\ s)))\ (\lambda y.\ y)\ \overline{\#}$$

Once we have that, we can take $b_E = \overbrace{\mathtt{any}_\Sigma}^{\text{Lemma 3.11}}\ (\mathtt{enum}_\Sigma\ \mathtt{tow}_n)\ \overbrace{t_E}^{\text{Lemma 3.10}}$ where $\mathtt{tow}_n = \overbrace{\overline{2}\ \ldots\ \overline{2}}^{n \text{ times}}$ satisfies $\vdash_{\mathsf{hls}} \mathtt{tow}_n : \mathtt{Nat}$ and is β-convertible to $\overline{\mathrm{tower}(n)}$ (cf. [BO09, Remark 3.5(iii)]). □

3.5. **Final remarks.** To conclude this paper, we discuss some topics related to Theorem 3.1 and its proof.

3.5.1. *On the complexity of star-free expression equivalence.* The equivalence problem for usual regular expressions, with Kleene star but no complementation, is "merely" PSpace-complete [Sto74, Theorem 4.13] – see also [HK11, Theorem 15]. One could therefore think that the main source of complexity in the case of star-free expressions is *complementation*.

This can be witnessed in the proof idea for Proposition 3.6. Translate E to an equivalent nondeterministic finite automaton (NFA), whose number of states bounds the length of a shortest word in the language (since the shortest accepting runs visit each state at most once). This can be done by induction on E, using any standard construction on NFA for union and concatenation; the costliest operation is complementation, which uses determinization, inducing a single exponential state blowup.

But in our translation from star-free expressions to λ-terms (Lemma 3.10), it turns out that complementation is trivial while concatenation increases the order of the types. Relatedly, in *deterministic* finite automata, complementation can be done without changing the set of states, while concatenation may need an exponential blowup of the number of states. So the complexity of the problem is in fact rooted in the *alternation* between complementation and concatenation – which leads to the "dot-depth hierarchy" [Pin17].

3.5.2. *Safety in higher-order recursion schemes.* Blum and Ong's main motivation for introducing the safe λ-calculus [BO09] was unrelated to complexity: they wanted to transpose to the λ-calculus the notion of safety on higher-order recursion schemes (HORS) [KNU02] – and indeed, the correspondence between HORS and $\lambda\mathbf{Y}$-terms (i.e. terms in the simply typed λ-calculus extended with fixed-point combinators $\mathbf{Y}_A : (A \to A) \to A$) relates safe HORS with safe $\lambda\mathbf{Y}$-terms [SW16]. Even without $\mathbf{Y}$, the safe λ-calculus has some interesting properties: for example, when substituting a safe λ-term into another, no spurious variable capture can happen [BO09, Lemma 1.10] – this leads to a way to normalize long-safe λ-terms by rewriting without α-renaming, but it is a bit subtle, see [FMvO23, Section 5.2]. A universal algebra perspective on the safe λ-calculus is also sketched in [Sal15, §2.3].

The homogeneity condition also comes from the study of HORS, cf. [Par18]. In fact, the homogeneous long-safe λ-calculus is equivalent (cf. [Blu09, Remark 3.53]) to an earlier attempt [dM06, Section 2.4.2] to design a safe λ-calculus inspired by HORS.

3.5.3. *Implicit automata in typed λ-calculi and transducer theory.* Lemma 3.10 tells us that every language of the form $[\![E]\!]$ for some star-free expression can be expressed by some homogeneous long-safe λ-term of type $\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$. These *star-free languages* are a fundamental and well-studied (cf. [Str18]) subclass of *regular languages*. Hillebrand and Kanellakis have shown that the languages computed by simply typed λ-terms of type $\mathtt{Str}_\Sigma[A] \to \mathtt{Bool}$ are, in fact, exactly the regular languages [HK96, Theorem 3.4][5]; they have a translation of deterministic finite automata (rather than regular expressions) into the simply typed λ-calculus that actually produces safe λ-terms. In fact, we used this translation of DFA in the proof of Lemma 3.10, for the "single letter" case of the induction.

By using an affine and non-commutative type system, one can get a variant of Hillebrand and Kanellakis's theorem characterizing star-free languages instead [NP20].[6] The proof does not translate star-free expressions; instead, to encode star-free languages, it goes through

[5]This paper [HK96] is the same that was already mentioned in §2.2.

[6]See also [Ngu21, Chapter 7] for a variant of this result (with linear instead of affine types).

an algebraic characterization. Nevertheless, these works were the main inspirations for our proof strategy for Theorem 3.1.

Further works [Ngu21, PP24] in this "implicit automata in typed λ-calculi" research programme have focused on characterizations of string-to-string (or tree-to-tree) functions computed by *transducers*, i.e. automata with output. For reasons that are close to the aforementioned connection with HORS, as discussed in [Ngu21, §1.4.1], definability in the safe λ-calculus characterizes an important class of functions from 1980s transducer theory. The functions defined by the safe λ-terms $\mathtt{split}_\Sigma$ and $\mathtt{enum}_\Sigma$ (from Lemmas 3.12 and 3.13 respectively) therefore belong to this class. The "split" function is also a typical example of the more recently introduced *polyregular functions*, cf. [Boj22, Kie24] – in fact, it is a slight variation on the "squaring with underlining" function of [Boj22, Example 3].

References

[AKST19] Kazuyuki Asada, Naoki Kobayashi, Ryoma Sin'ya, and Takeshi Tsukada. Almost every simply typed lambda-term has a long beta-reduction sequence. *Logical Methods in Computer Science*, 15(1), 2019. doi:10.23638/LMCS-15(1:16)2019.

[AL13] Andrea Asperti and Jean-Jacques Lévy. The cost of usage in the lambda-calculus. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 293–300. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.35.

[Asp17] Andrea Asperti. About the efficient reduction of lambda terms. *CoRR*, abs/1701.04240, 2017. arXiv:1701.04240.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, January 1985. doi:10.1016/0304-3975(85)90135-5.

[Bec01] Arnold Beckmann. Exact bounds for lengths of reductions in typed lambda-calculus. *Journal of Symbolic Logic*, 66(3):1277–1285, 2001. doi:10.2307/2695106.

[Blu09] William Blum. *The safe lambda calculus*. PhD thesis, University of Oxford, 2009. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.504329.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

[BO09] William Blum and C.-H. Luke Ong. The Safe Lambda Calculus. *Logical Methods in Computer Science*, 5(1), February 2009. doi:10.2168/LMCS-5(1:3)2009.

[Boj22] Mikołaj Bojańczyk. Transducers of polynomial growth. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 1:1–1:27. ACM, 2022. doi:10.1145/3531130.3533326.

[BS23] Pablo Barenbaum and Cristian Sottile. Two decreasing measures for simply typed λ-terms. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, July 3-6, 2023, Rome, Italy*, volume 260 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.FSCD.2023.11.

[CHHM22] Dmitry Chistikov, Christoph Haase, Zahra Hadizadeh, and Alessio Mansutti. Higher-Order Quantified Boolean Satisfiability. In Stefan Szeider, Robert Ganian, and Alexandra Silva, editors, *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, volume 241 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2022.33.

[Con20] Andrea Condoluci. *Beta-Conversion, Efficiently*. PhD thesis, Alma Mater Studiorum – Università di Bologna, April 2020. URL: http://amsdottorato.unibo.it/9444/.

[dM06] 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.

[FMvO23] Samuel Frontull, Georg Moser, and Vincent van Oostrom. α-Avoidance. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*, volume 260 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2023.22.

[Geu92] Herman Geuvers. The Church-Rosser property for beta-eta-reduction in typed lambda-calculi. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*, pages 453–460. IEEE Computer Society, 1992. doi:10.1109/LICS.1992.185556.

[GRV09] Marco Gaboardi, Luca Roversi, and Luca Vercelli. A By-Level Analysis of Multiplicative Exponential Linear Logic. In *Mathematical Foundations of Computer Science 2009*, Lecture Notes in Computer Science, pages 344–355. Springer, Berlin, Heidelberg, August 2009. doi:10.1007/978-3-642-03816-7_30.

[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*, pages 253–263. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561337.

[HK11] Markus Holzer and Martin Kutrib. The complexity of regular(-like) expressions. *International Journal of Foundations of Computer Science*, 22(7):1533–1548, 2011. doi:10.1142/S0129054111008866.

[HKM96] Gerd G. Hillebrand, Paris C. Kanellakis, and Harry G. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. *Information and Computation*, 127(2):117–144, June 1996. doi:10.1006/inco.1996.0055.

[JT18] Guilhem Jaber and Nikos Tzevelekos. A trace semantics for system F parametric polymorphism. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2018. doi:10.1007/978-3-319-89366-2_2.

[Kie24] Sandra Kiefer. Polyregular functions – characterisations and refutations. In Joel D. Day and Florin Manea, editors, *Developments in Language Theory - 28th International Conference, DLT 2024, Göttingen, Germany, August 12-16, 2024, Proceedings*, volume 14791 of *Lecture Notes in Computer Science*, pages 13–21. Springer, 2024. doi:10.1007/978-3-031-66159-4_2.

[KNU02] Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. Higher-order pushdown trees are easy. In Mogens Nielsen and Uffe Engberg, editors, *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*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002. doi:10.1007/3-540-45931-6_15.

[KSW16] Fairouz Kamareddine, Jonathan P. Seldin, and J. B. Wells. Bridging Curry and Church's typing style. *Journal of Applied Logic*, 18:42–70, 2016. doi:10.1016/j.jal.2016.05.008.

[Mai92] Harry G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, September 1992. doi:10.1016/0304-3975(92)90020-G.

[MS15] Guillaume Munch-Maccagnoni and Gabriel Scherer. Polarised intermediate representation of lambda calculus with sums. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 127–140. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.22.

[Ngu21] Lê Thành Dũng Nguyễn. *Implicit automata in linear logic and categorical transducer theory*. PhD thesis, Université Paris XIII (Sorbonne Paris Nord), December 2021. URL: https://theses.hal.science/tel-04132636.

[NP20] Lê Thành Dũng Nguyễn and Cécilia Pradic. Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 135:1–135:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.135.

[PA93] Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1993. doi:10.1007/BFb0037118.

[Par18] Paweł Parys. Homogeneity without loss of generality. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FSCD.2018.27.

[Pin17] Jean-Éric Pin. The dot-depth hierarchy, 45 years later. In Stavros Konstantinidis, Nelma Moreira, Rogério Reis, and Jeffrey O. Shallit, editors, *The Role of Theory in Computer Science – Essays Dedicated to Janusz Brzozowski*, pages 177–202. World Scientific, 2017. doi:10.1142/9789813148208_0008.

[PP24] Cécilia Pradic and Ian Price. Implicit automata in λ-calculi III: affine planar string-to-string functions, 2024. arXiv:2404.03985.

[Sal15] Sylvain Salvati. *Lambda-calculus and formal language theory*. Habilitation à diriger des recherches, Université de Bordeaux, December 2015. URL: https://theses.hal.science/tel-01253426.

[Sch91] Helmut Schwichtenberg. An upper bound for reduction sequences in the typed λ-calculus. *Archive for Mathematical Logic*, 30(5-6):405–408, 1991. doi:10.1007/BF01621476.

[Sch16] Sylvain Schmitz. Complexity hierarchies beyond elementary. *ACM Transactions on Computation Theory*, 8(1):3:1–3:36, 2016. doi:10.1145/2858784.

[Sch17] Gabriel Scherer. Deciding equivalence with sums and the empty type. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 374–386. ACM, 2017. doi:10.1145/3009837.3009901.

[Sin22] Alexandros Singh. *A unified approach to the combinatorics of the λ-calculus and maps: bijections and limit properties*. PhD thesis, Université Paris XIII (Sorbonne Paris Nord), 2022. URL: https://theses.hal.science/tel-04069290.

[Sta79] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, July 1979. doi:10.1016/0304-3975(79)90007-0.

[Sto74] Larry J. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, 1974. URL: http://hdl.handle.net/1721.1/15540.

[Str18] Howard Straubing. First-order logic and aperiodic languages: a revisionist history. *ACM SIGLOG News*, 5(3):4–20, 2018. doi:10.1145/3242953.3242956.

[SW16] Sylvain Salvati and Igor Walukiewicz. Simply typed fixpoint calculus and collapsible pushdown automata. *Mathematical Structures in Computer Science*, 26(7):1304–1350, October 2016. doi:10.1017/S0960129514000590.

[Tak95] Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995. doi:10.1006/inco.1995.1057.

[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)*, pages 323–338, 2012. doi:10.4230/LIPIcs.RTA.2012.323.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987. doi:10.3233/FI-1987-10202.