Complexity Hierarchies and Higher-Order Cons-Free Rewriting*

Cynthia Kop[1] and Jakob Grue Simonsen[2]

1 **Department of Computer Science, University of Copenhagen (DIKU), Copenhagen, Denmark**
`kop@di.ku.dk`

2 **Department of Computer Science, University of Copenhagen (DIKU), Copenhagen, Denmark**
`{simonsen@di.ku.dk`

Abstract

Constructor rewriting systems are said to be cons-free if, roughly, constructor terms in the right-hand sides of rules are subterms of constructor terms in the left-hand side; the computational intuition is that rules cannot build new data structures. It is well-known that cons-free programming languages can be used to characterize computational complexity classes, and that cons-free first-order term rewriting can be used to characterize the set of polynomial-time decidable sets.

We investigate cons-free higher-order term rewriting systems, the complexity classes they characterize, and how these depend on the order of the types used in the systems. We prove that, for every $k \geq 1$, left-linear cons-free systems with type order k characterize $\mathrm{E}^k\mathrm{TIME}$ if arbitrary evaluation is used (i.e., the system does not have a fixed reduction strategy).

The main difference with prior work in implicit complexity is that (i) our results hold for non-orthogonal term rewriting systems with possible rule overlaps with no assumptions about reduction strategy, (ii) results for such term rewriting systems have previously only been obtained for $k = 1$, and with additional syntactic restrictions on top of cons-freeness and left-linearity.

Our results are apparently among the first implicit characterizations of the hierarchy $\mathrm{E} = \mathrm{E}^1\mathrm{TIME} \subsetneq \mathrm{E}^2\mathrm{TIME} \subsetneq \cdots$. Our work confirms prior results that having full non-determinism (via overlaps of rules) does not directly allow characterization of non-deterministic complexity classes like NE. We also show that non-determinism makes the classes characterized highly sensitive to minor syntactic changes such as admitting product types or non-left-linear rules.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases higher-order term rewriting, implicit complexity, cons-freeness, ETIME hierarchy

Digital Object Identifier 10.4230/LIPIcs.FSCD.2016.23

1 Introduction

In [14], Jones introduces *cons-free programming*: working with a small functional programming language, cons-free programs are exactly those where function bodies cannot contain use of data constructors (the "cons" operator on lists). Put differently, a cons-free program is

* The authors are supported by the Marie Skłodowska-Curie action "HORIP", program H2020-MSCA-IF-2014, 658162 and by the Danish Council for Independent Research Sapere Aude grant "Complexity via Logic and Algebra" (COLA).

read-only: data structures cannot be created or altered, only read from the input; and any data passed as arguments to recursive function calls must thus be part of the original input.

The interest in such programs lies in their applicability to computational complexity: by imposing cons-freeness, the resulting set of programs can only decide the sets in a proper subclass of the Turing-decidable sets, indeed are said to *characterize* the subclass. Jones goes on to show that adding further restrictions such as type order or enforcing tail recursion lowers the resulting expressiveness to known classes. For example, cons-free programs with data order 0 can decide exactly the sets in PTIME, while tail-recursive cons-free programs with data order 1 can decide exactly the sets in PSPACE. The study of such restrictions and the complexity classes characterized is a research area known as *implicit complexity* and has a long history with many distinct approaches (see, e.g., [4, 6, 5, 7, 8, 12, 17]).

Rather than a toy language, it is tantalizing to consider *term rewriting* instead. Term rewriting systems have no fixed evaluation order (so call-by-name or call-by-value can be introduced as needed, but are not *required*); and term rewriting is natively non-deterministic, allowing distinct rules to be applied ("functions to be invoked") to the same piece of syntax, hence could be useful for extensions towards non-deterministic complexity classes. Implicit complexity using term rewriting has seen significant advances using a plethora of approaches (e.g. [1, 2, 3]). Most of this research has, however, considered fixed evaluation orders (most prominently innermost reduction), and if not, then systems which are either orthogonal, or at least confluent (e.g. [2]). Almost all of the work considers only first-order rewriting.

The authors of [11] provide a first definition of cons-free term rewriting without constraints on evaluation order or confluence requirements, and prove that this class – limited to *first-order* rewriting – characterizes PTIME. However, they impose a rather severe partial linearity restriction on the programs. This paper seeks to answer two questions: (i) what happens if *no* restrictions beyond left-linearity and cons-freeness are imposed? And (ii) what if *higher-order* term rewriting – including bound variables as in the lambda calculus – is allowed? We obtain that k^{th}-order cons-free term rewriting exactly characterizes E^kTIME. This is surprising because in Jones' rewriting-like language, k^{th}-order programs characterize $\text{EXP}^{k-1}\text{TIME}$: surrendering both determinism and evaluation order thus significantly increases expressivity.

An extended version, including appendices with full proofs, is available online [16].

2 Preliminaries

2.1 Computational complexity

We presuppose introductory working knowledge of computability and complexity theory (corresponding to standard textbooks, e.g., [13]). Notation is fixed below.

Turing Machines (TMs) are triples (A, S, T) where A is a finite set of tape symbols such that $A \supseteq I \cup \{\sqcup\}$, where $I \supseteq \{0, 1\}$ is a set of *initial symbols* and $\sqcup \notin I$ is the special *blank* symbol; $S \supseteq \{\texttt{start}, \texttt{accept}, \texttt{reject}\}$ is a finite set of states, and T is a finite set of transitions (i, r, w, d, j) with $i \in S \setminus \{\texttt{accept}, \texttt{reject}\}$ (the *original state*), $r \in A$ (the *read symbol*), $w \in A$ (the *written symbol*), $d \in \{\texttt{L}, \texttt{R}\}$ (the *direction*), and $j \in S$ (the *result state*). We sometimes write this transition as $i \xRightarrow{r/w\ d} j$. All TMs in the paper are deterministic and (which we can assume wlog.) do not get stuck: for every pair (i, r) with $i \in S \setminus \{\texttt{accept}, \texttt{reject}\}$ and $r \in A$ there is exactly one transition (i, r, w, d, j). Every TM has a single, right-infinite tape.

A *valid tape* is a right-infinite sequence of tape symbols with only finitely many not $\sqcup$. A *configuration* of a TM is a triple (t, p, s) with t a valid tape, $p \in \mathbb{N}$ and $s \in S$. The transitions T induce a binary relation $\Rightarrow$ between configurations in the obvious way.

A TM with input alphabet I *decides* $X \subseteq I^+$ if for any string $x \in I^+$, we have $x \in X$ iff $(\llcorner x_1 \dots x_n \llcorner\llcorner \dots, 0, \texttt{start}) \Rightarrow^* (t, i, \texttt{accept})$ for some t, i, and $(\llcorner x_1 \dots x_n \llcorner\llcorner \dots, 0, \texttt{start}) \Rightarrow^* (t, i, \texttt{reject})$ otherwise (i.e., the machine halts on all inputs, ending in `accept` or `reject` depending on whether $x \in X$). If $f : \mathbb{N} \longrightarrow \mathbb{N}$ is a function, a (deterministic) TM *runs in time* $\lambda n.f(n)$ if, for each $n \in \mathbb{N} \setminus \{0\}$ and each $x \in I^n$: $(\llcorner x \llcorner\llcorner \dots, 0, \texttt{start}) \Rightarrow^{\leq f(n)} (t, i, \underline{\texttt{s}})$ for $\underline{\texttt{s}} \in \{\texttt{accept}, \texttt{reject}\}$, where $\Rightarrow^{\leq f(n)}$ denotes a sequence of at most $f(n)$ transitions.

Complexity and the ETIME hierarchy

For $k, n \geq 0$, let $\exp_2^0(n) = n$ and $\exp_2^{k+1}(n) = 2^{\exp_2^k(n)} = \exp_2^k(2^n)$.

▶ **Definition 1.** Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be a function. Then, $\mathrm{TIME}\,(f(n))$ is the set of all $S \subseteq \{0,1\}^+$ such that there exist $a > 0$ and a deterministic TM running in time $\lambda n.a \cdot f(n)$ that decides S (i.e., S is decidable in time $O(f(n))$). For $k \geq 1$ define: $\mathrm{E}^k\mathrm{TIME} \triangleq \bigcup_{a \in \mathbb{N}} \mathrm{TIME}\left(\exp_2^k(an)\right)$

Observe in particular that $\mathrm{E}^1\mathrm{TIME} = \bigcup_{a \in \mathbb{N}} \mathrm{TIME}\left(\exp_2^1(an)\right) = \bigcup_{a \in \mathbb{N}} \mathrm{TIME}\,(2^{an}) = \mathrm{E}$ (where E is the usual complexity class of this name, see e.g., [19, Ch. 20]).

Note that for any $d, k \geq 1$, we have $(\exp_2^k(x))^d = 2^{d \cdot \exp_2^{k-1}(x)} \leq 2^{\exp_2^{k-1}(dx)} = \exp_2^k(dx)$. Hence, if P is a polynomial with non-negative integer coefficients and the set $S \subseteq \{0,1\}^+$ is decided by an algorithm running in time $O(P(\exp_2^k(an)))$ for some $a \in \mathbb{N}$, then $S \in \mathrm{E}^k\mathrm{TIME}$.

Using the Time Hierarchy Theorem [20], it is easy to see that $\mathrm{E} = \mathrm{E}^1\mathrm{TIME} \subsetneq \mathrm{E}^2\mathrm{TIME} \subsetneq \mathrm{E}^3\mathrm{TIME} \subsetneq \cdots$. The union $\bigcup_{k \in \mathbb{N}} \mathrm{E}^k\mathrm{TIME}$ is the set ELEMENTARY of elementary languages.

2.2 Higher-order rewriting

Unlike first-order term rewriting, there is no single, unified approach to higher-order term rewriting, but rather a number of different co-extensive systems with distinct syntax; for an overview of basic issues, see [21]. We will use *Algebraic Functional Systems* (AFSs) [15, 9], in the simplest form (which disallows partial applications). However, our proofs do not use any particular features of AFSs that preclude using different higher-order formalisms.

Types and Terms

We assume a non-empty set $\mathcal{S}$ of *sorts*, and define types and type orders as follows: (i) every $\iota \in \mathcal{S}$ is a type of order *0*; (ii) if σ, τ are types of order n and m respectively, then $\sigma \Rightarrow \tau$ is a type of order $\max(n+1, m)$. Here $\Rightarrow$ is right-associative, so $\sigma \Rightarrow \tau \Rightarrow \pi$ should be read $\sigma \Rightarrow (\tau \Rightarrow \pi)$. A *type declaration* of order $k \geq 0$ is a tuple $[\sigma_1 \times \cdots \times \sigma_n] \Rightarrow \iota$ with all σ_i types of order at most $k-1$, and $\iota \in \mathcal{S}$; if $n = 0$ this declaration may simply be denoted ι.

We additionally assume given disjoint sets $\mathcal{F}$ of *function symbols* and $\mathcal{V}$ of *variables*. Each symbol in $\mathcal{F}$ is associated with a unique type declaration, and each variable in $\mathcal{V}$ with a unique type. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of *terms over* $\mathcal{F}$ *and* $\mathcal{V}$ consists of those expressions s such that $\vdash s : \sigma$ can be derived for some type σ using the following clauses:

`(var)`	$\vdash x : \sigma$	if $x : \sigma \in \mathcal{V}$
`(app)`	$\vdash s \cdot t : \tau$	if $s : \sigma \Rightarrow \tau$ and $t : \sigma$
`(abs)`	$\vdash \lambda x.s : \sigma \Rightarrow \tau$	if $x : \sigma \in \mathcal{V}$ and $s : \tau$
`(fun)`	$\vdash f(s_1, \dots, s_n) : \iota$	if $f : [\sigma_1 \times \dots \times \sigma_n] \Rightarrow \iota \in \mathcal{F}$ and $s_1 : \sigma_1, \dots, s_n : \sigma_n$

Clearly, each term has a *unique* type. Note that a function symbol $f : [\sigma_1 \times \dots \times \sigma_n] \Rightarrow \iota$ takes exactly n arguments, and its output type ι is a sort. The *abstraction* construction $\lambda x.s$ binds occurrences of x in s as in the λ-calculus, and α-conversion is defined for terms *mutatis mutandis*; we identify terms modulo α-conversion, renaming bound variables if necessary.

Application is left-associative. The set of variables of s which are not bound is denoted $FV(s)$. A term s is *closed* if $FV(s) = \emptyset$. We say that a term s *has base type* if $\vdash s : \iota \in \mathcal{S}$.

▶ **Example 2.** We will often use extensions of the signature $\mathcal{F}_{\texttt{string}}$, given by:

$$\begin{array}{rclcrclcrcl} \texttt{true} & : & \texttt{bool} & & \texttt{0} & : & [\texttt{string}] \Rightarrow \texttt{string} & & \rhd & : & \texttt{string} \\ \texttt{false} & : & \texttt{bool} & & \texttt{1} & : & [\texttt{string}] \Rightarrow \texttt{string} & & & & \end{array}$$

Terms are for instance $\texttt{true}$, $\lambda x.\texttt{0}(\texttt{1}(x))$ and $(\lambda x.\texttt{0}(x)) \cdot \texttt{1}(y)$. The first and last of these terms have base type, and the first two are closed; the last one has y as a free variable.

A *substitution* is a type-preserving map from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F},\mathcal{V})$ which is the identity on all but finitely many variables. Substitutions γ are extended to arbitrary terms s, notation $s\gamma$, by using α-conversion to rename all bound variables in s to fresh ones, then replacing each unbound variable x by $\gamma(x)$. A *context* $C[]$ is a term in $\mathcal{T}(\mathcal{F},\mathcal{V})$ in which a single occurrence of a variable is replaced by a symbol $\Box \notin \mathcal{F} \cup \mathcal{V}$. The result of replacing $\Box$ in $C[]$ by a term s (of matching type) is denoted $C[s]$. Free variables may be captured; e.g. $(\lambda x.\Box)[x] = \lambda x.x$. If $s = C[t]$ we say that t is a *subterm* of s, notation $t \trianglelefteq s$, or $t \lhd s$ if $C[] \neq \Box$.

Rules and Rewriting

A *rule* is a pair $\ell \to r$ of terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$ with the same *sort* (i.e. $\vdash \ell : \iota$ and $\vdash r : \iota$ for some $\iota \in \mathcal{S}$), such that ℓ has the form $f(\ell_1,\dots,\ell_n)$ with $f \in \mathcal{F}$ and such that $FV(r) \subseteq FV(\ell)$. A rule $\ell \to r$ is *left-linear* if every variable occurs at most once in ℓ. We assume given a set $\mathcal{R}$ of rules, and define the one-step rewrite relation $\to_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$ as follows:

$$\begin{array}{rcll} C[\ell\gamma] & \to_{\mathcal{R}} & C[r\gamma] & \quad \text{with } \ell \to r \in \mathcal{R},\ C \text{ a context, } \gamma \text{ a substitution} \\ C[(\lambda x.s) \cdot t] & \to_{\mathcal{R}} & C[s[x := t]] & \end{array}$$

We may write $s \to_\beta t$ for a rewrite step using (beta). Let $\to_{\mathcal{R}}^+$ denote the transitive closure of $\to_{\mathcal{R}}$ and $\to_{\mathcal{R}}^*$ the transitive-reflexive closure. We say that s *reduces to* t if $s \to_{\mathcal{R}} t$. A term s is in *normal form* if there is no t such that $s \to_{\mathcal{R}} t$, and t *is a normal form of* s if $s \to_{\mathcal{R}}^* t$ and t is in normal form. An AFS is a pair $(\mathcal{F},\mathcal{R})$, generating a set of terms and a reduction relation. The *order* of an AFS is the maximal order of any type declaration in $\mathcal{F}$.

▶ **Example 3.** Recall the signature $\mathcal{F}_{\texttt{string}}$ from Example 2; let $\mathcal{F}_{\texttt{count}}$ be its extension with $\texttt{succ} : [\texttt{string}] \Rightarrow \texttt{string}$. We consider the AFS $(\mathcal{F}_{\texttt{count}}, \mathcal{R}_{\texttt{count}})$ with the following rules:

$$\begin{array}{lrcllrcl} \text{(A)} & \texttt{succ}(\rhd) & \to & \texttt{1}(\rhd) & \text{(B)} & \texttt{succ}(\texttt{0}(xs)) & \to & \texttt{1}(xs) \\ & & & & \text{(C)} & \texttt{succ}(\texttt{1}(xs)) & \to & \texttt{0}(\texttt{succ}(xs)) \end{array}$$

This is a *first-order* AFS, implementing the successor function on a binary number expressed as a bitstring with the least significant digit first. For example, 5 is represented by $\texttt{1}(\texttt{0}(\texttt{1}(\rhd)))$, and indeed $\texttt{succ}(\texttt{1}(\texttt{0}(\texttt{1}(\rhd)))) \to_{\mathcal{R}} \texttt{0}(\texttt{succ}(\texttt{0}(\texttt{1}(\rhd)))) \to_{\mathcal{R}} \texttt{0}(\texttt{1}(\texttt{1}(\rhd)))$, which represents 6.

▶ **Example 4.** Alternatively, we may define a bit-sequence as a *function*: let $\mathcal{F}_{\texttt{hocount}}$ be the extension of $\mathcal{F}_{\texttt{string}}$ with $\texttt{not} : [\texttt{bool}] \Rightarrow \texttt{bool}$, $\texttt{ite} : [\texttt{bool} \times \texttt{bool} \times \texttt{bool}] \Rightarrow \texttt{bool}$ and $\texttt{all}$, $\texttt{succ} : [(\texttt{bool} \Rightarrow \texttt{bool}) \times \texttt{string}] \Rightarrow \texttt{string}$. Let $\mathcal{R}_{\texttt{hocount}}$ consist of:

$$\begin{array}{lrcl} \text{(A)} & \texttt{ite}(\texttt{true},x,y) & \to & x \quad \text{(C)} \quad \texttt{not}(x) \to \texttt{ite}(x,\texttt{false},\texttt{true}) \\ \text{(B)} & \texttt{ite}(\texttt{false},x,y) & \to & y \quad \text{(D)} \quad \texttt{all}(F,\rhd) \to F \cdot \rhd \\ \text{(E)} & \texttt{all}(F,\underline{\texttt{a}}(xs)) & \to & \texttt{ite}(F \cdot \underline{\texttt{a}}(xs), \texttt{all}(F,xs), \texttt{false}) \qquad [\![\text{for } \underline{\texttt{a}} \in \{\texttt{0},\texttt{1}\}]\!] \\ \text{(F)} & \texttt{succ}(F,\rhd) & \to & \texttt{not}(F \cdot \rhd) \\ \text{(G)} & \texttt{succ}(F,\underline{\texttt{a}}(xs)) & \to & \texttt{ite}(\texttt{all}(F,xs), \texttt{not}(F \cdot \underline{\texttt{a}}(xs)), F \cdot \underline{\texttt{a}}(xs)) \ [\![\text{for } \underline{\texttt{a}} \in \{\texttt{0},\texttt{1}\}]\!] \end{array}$$

Note that (E) and (G) each represent *two* rules: one for each choice of $\underline{\mathtt{a}}$. This AFS is second-order, due to `all` and `succ`. A function F represents a (potentially infinite) binary number, with the i^{th} bit given by $F \cdot t$ for any bitstring t of length i (counting from $i = 0$, so $t = \rhd$). Thus, the number 0 is represented by, e.g., $\lambda x.\mathtt{false}$, and 1 by $\mathtt{ONE} ::= \lambda x.\mathtt{succ}(\lambda y.\mathtt{false}, x)$. Indeed $\mathtt{ONE} \cdot \rhd = (\lambda x.\mathtt{succ}(\lambda y.\mathtt{false}, x)) \cdot \rhd \rightarrow_\beta \mathtt{succ}(\lambda y.\mathtt{false}, \rhd) \rightarrow_\mathcal{R} \mathtt{not}((\lambda y.\mathtt{false}) \cdot \rhd) \rightarrow_\beta \mathtt{not}(\mathtt{false}) \rightarrow_\mathcal{R} \mathtt{true}$, and $\mathtt{ONE} \cdot \mathtt{0}^k(\rhd) \rightarrow^*_\mathcal{R} \mathtt{false}$ for $k > 0$.

We fix a partitioning of $\mathcal{F}$ into two disjoint sets, $\mathcal{D}$ of *defined symbols* and $\mathcal{C}$ of *constructor symbols*, such that $f \in \mathcal{D}$ for all $f(\vec{\ell}) \to r \in \mathcal{R}$. A term s is a *constructor term* if it is in $\mathcal{T}(\mathcal{C}, \mathcal{V})$ and a *proper constructor term* if it also contains no applications or abstractions. A *closed* proper constructor term is also called a *data term*. The set of data terms is denoted $\mathcal{DA}$. Note that data terms are built using only clause (`fun`). A term $f(s_1, \dots, s_n)$ with $f \in \mathcal{D}$ and each $s_i \in \mathcal{DA}$ is called a *basic term*. A *constructor rewriting system* is an AFS where each rule $f(\ell_1, \dots, \ell_n) \to r \in \mathcal{R}$ satisfies that all ℓ_i are proper constructor terms (and $f \in \mathcal{D}$). An AFS is a *left-linear constructor rewriting system* if moreover each rule is left-linear.

In a constructor rewriting system, β-reduction steps can always be done prior to other steps: if s has a normal form q and $s \rightarrow_\beta t$, then also $t \rightarrow^*_\mathcal{R} q$. Therefore we can (and will!) safely assume that the right-hand sides of rules are in normal form with respect to $\rightarrow_\beta$.

▶ **Example 5.** The AFSs from Examples 3 and 4 are left-linear constructor rewriting systems. In Example 3, $\mathcal{C} = \mathcal{F}_{\mathtt{string}}$ and $\mathcal{D} = \{\mathtt{succ}\}$. If a rule $\mathtt{0}(\rhd) \to \rhd$ were added to $\mathcal{R}_{\mathtt{count}}$, it would no longer be a constructor system, as this would force `0` to be in $\mathcal{D}$, conflicting with rule (B). A rule such as $\mathtt{equal}(xs, xs) \to \mathtt{true}$ would break left-linearity.

▶ Remark. Constructor rewriting systems – typically left-linear – are very common both in the literature on term rewriting and in functional programming, where similar restrictions are imposed. Left-linear systems are well-behaved: contraction of non-overlapping redexes cannot destroy redexes that they themselves are arguments of. Constructor systems avoid non-root overlaps, and allow for a clear split between data and intermediate terms.

They are, however, less common in the literature on *higher-order* term rewriting, and the notion of a *proper* constructor term is new for AFSs (although the exclusion of abstractions and applications in the left-hand sides roughly corresponds to *fully extended pattern HRSs* in Nipkow's style of higher-order rewriting [18]).

Deciding problems using rewriting

Like Turing Machines, an AFS can decide a set $X \subseteq I^+$ (where I is a finite set of symbols). Consider AFSs with a signature $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$ where $\mathcal{C}$ contains symbols $\rhd : \mathtt{string}$, $\mathtt{true} : \mathtt{bool}$, $\mathtt{false} : \mathtt{bool}$ and $\mathtt{a} : [\mathtt{string}] \Rightarrow \mathtt{string}$ for all $a \in I$. There is an obvious correspondence between elements of I^+ and data terms of sort `string`; if $x \in I^+$, we write $\overline{\mathtt{x}}$ for the corresponding data term. The AFS *accepts* $D \subseteq I^+$ if there is a designated defined symbol $\mathtt{decide} : [\mathtt{string}] \Rightarrow \mathtt{bool}$ such that, for every $x \in I^+$ we have $\mathtt{decide}(\overline{\mathtt{x}}) \rightarrow^*_\mathcal{R} \mathtt{true}$ iff $x \in D$. More generally, we are interested in the reductions of a given *basic term* to a *data term*.

We use the acceptance criterion above – reminiscent of the acceptance criterion of non-deterministic Turing machines – because term rewriting is inherently non-deterministic unless further constraints (e.g., orthogonality) are imposed. Thus, an input x is "rejected" by a rewriting system iff there is no reduction to `true` from $\mathtt{decide}(\overline{\mathtt{x}})$; and as evaluation is non-deterministic, there may be many distinct reductions starting from $\mathtt{decide}(\overline{\mathtt{x}})$.

3 Cons-free rewriting

Since the purpose of this research is to find groups of programs which can handle *restricted* classes of Turing-computable problems, we will impose certain limitations. In particular, we will limit interest to *cons-free left-linear constructor rewriting systems.*

▶ **Definition 6.** A rule $\ell \to r$, presented using α-conversion in a form where all binders are distinct from $FV(\ell)$, is *cons-free* if for all subterms $s = f(s_1, \dots, s_n) \trianglelefteq r$ with $f \in \mathcal{C}$, we have $s \lhd \ell$ or $s \in \mathcal{DA}$. A left-linear constructor AFS $(\mathcal{F}, \mathcal{R})$ is cons-free if all rules in $\mathcal{R}$ are.

This definition corresponds largely to the definitions of cons-freeness appearing in [11, 14]. In a cons-free AFS, it is not possible to create more data, as we will see in Section 3.1.

▶ **Example 7.** The AFS from Example 3 is not cons-free due to rules (B) and (C). The AFS from Example 4 *is* cons-free (in rules (E) and (G), $\underline{\mathtt{a}}(xs)$ is allowed to occur on the right despite the constructor $\underline{\mathtt{a}}$, because it also occurs on the left). However, there are few interesting basic terms, as we do not consider for instance $\mathtt{succ}(\lambda x.\mathtt{false}, \rhd)$ basic.

▶ Remark. The limitation to left-linear constructor AFSs is standard, but also *necessary*: if either restriction is dropped, our limitation to cons-free AFSs becomes meaningless. In the case of constructor systems, this is obvious: if defined symbols are allowed to occur within a left-hand side, then we could simply let $\mathcal{D} := \mathcal{F}$ and have a "cons-free" system. The case of left-linearity is a bit more sophisticated; this we will study in more detail in Section 6.

As the first two restrictions are necessary to give meaning to the third, we will consider the limitation to left-linear constructor AFSs implicit in the notion "cons-free".

3.1 Properties of Cons-free Term Rewriting

As mentioned, cons-free term rewriting cannot create new data. This means that the set of data terms that might occur during a reduction starting in some basic term s are exactly the data terms occurring in s, or those occurring in the right-hand side of some rule. Formally:

▶ **Definition 8.** Let $(\mathcal{F}, \mathcal{R})$ be a constructor AFS. For a given term s, the set $\mathcal{B}_s$ contains all data terms t such that (i) $s \trianglerighteq t$, or (ii) $r \trianglerighteq t$ for some rule $\ell \to r \in \mathcal{R}$.

$\mathcal{B}_s$ is a set of data terms, is closed under subterms and, since we have assumed $\mathcal{R}$ to be fixed, has a linear number of elements in the size of s. The property that no new data is generated by reducing s is formally expressed by the following result:

▶ **Definition 9** ($\mathcal{B}$-safety)**.** Let $\mathcal{B} \subseteq \mathcal{DA}$ be a set which (i) is closed under taking subterms, and (ii) contains all data terms occurring as a subterm of the right-hand side of a rule in $\mathcal{R}$. A term s is *$\mathcal{B}$-safe* if for all t with $s \trianglerighteq t$: if t has the form $c(t_1, \dots, t_m)$ with $c \in \mathcal{C}$, then $t \in \mathcal{B}$.

▶ **Lemma 10.** *If s is $\mathcal{B}$-safe and $s \to_{\mathcal{R}} t$, then t is $\mathcal{B}$-safe.*

Proof Sketch. By induction on the form of s; the result follows trivially by the induction hypothesis if the reduction does not take place at the root, leaving only the base cases $s = (\lambda x.u) \cdot v \to_{\mathcal{R}} u[x := v] = t$ and $s = \ell\gamma \to_{\mathcal{R}} r\gamma = t$. The first of these is easy by induction on the form of the ($\mathcal{B}$-safe!) term u, the second follows by induction on the form of r (which, as the right-hand side of a cons-free rule, has convenient properties). ◀

Thus, if we start with a basic term $f(\vec{s})$, any data terms occurring in a reduction $f(\vec{s}) \to_{\mathcal{R}}^* t$ (directly or as subterms) are in $\mathcal{B}_{f(\vec{s})}$. This insight will be instrumental in Section 5.

▶ **Example 11.** By Lemma 10, functions in a cons-free AFSs cannot build recursive data. To code around this, we might use subterms of the input as a measure of length. Consider the decision problem whether a given bitstring is a palindrome. We cannot use a rule such as $\mathtt{decide}(cs) \to \mathtt{equal}(cs, \mathtt{reverse}(cs))$ since, by Lemma 10, it is impossible to define `reverse`. Instead, a typical solution uses a string ys of length k to find $\overline{\mathtt{c_k}}$ in $\overline{\mathtt{c_0} \dots \mathtt{c_{n-1}}}$:

$$\begin{array}{rcll}
\mathtt{decide}(cs) & \to & \mathtt{palindrome}(cs, cs) & \\
\mathtt{palindrome}(cs, \rhd) & \to & \mathtt{true} & \\
\mathtt{palindrome}(cs, \underline{\mathtt{a}}(ys)) & \to & \mathtt{and}(\mathtt{palindrome}(cs, ys), \mathtt{chk}_{\underline{\mathtt{a}}}(cs, ys)) & [\![\mathtt{a} \in \{\mathtt{0},\mathtt{1}\}]\!]
\end{array}$$

$$\begin{array}{rcl@{\qquad}rcll}
\mathtt{and}(\mathtt{true}, x) & \to & x & \mathtt{chk}_{\underline{\mathtt{a}}}(\underline{\mathtt{a}}(xs), \rhd) & \to & \mathtt{true} & [\![\mathtt{a} \in \{\mathtt{0},\mathtt{1}\}]\!] \\
\mathtt{and}(\mathtt{false}, x) & \to & \mathtt{false} & \mathtt{chk}_{\underline{\mathtt{a}}}(\underline{\mathtt{b}}(xs), \rhd) & \to & \mathtt{false} & [\![\mathtt{a},\mathtt{b} \in \{\mathtt{0},\mathtt{1}\} \wedge \mathtt{a} \neq \mathtt{b}]\!] \\
 & & & \mathtt{chk}_{\underline{\mathtt{a}}}(\underline{\mathtt{b}}(xs), \underline{\mathtt{c}}(ys)) & \to & \mathtt{chk}_{\underline{\mathtt{a}}}(xs, ys) & [\![\mathtt{a},\mathtt{b},\mathtt{c} \in \{\mathtt{0},\mathtt{1}\}]\!]
\end{array}$$

(The signature extends $\mathcal{F}_{\mathtt{string}}$, but is otherwise omitted as types can easily be derived.)

Through cons-freeness, we obtain another useful property: we do not have to consider constructors which take functional arguments.

▶ **Lemma 12.** *Given a cons-free AFS $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$, let $Y = \{c : [\sigma_1 \times \dots \times \sigma_n] \Rightarrow \iota \in \mathcal{C}$ some σ_i is not a sort$\}$. Define $\mathcal{F}' := \mathcal{F} \setminus Y$, and let $\mathcal{R}'$ consist of those rules in $\mathcal{R}$ not using any element of Y in either left- or right-hand side. Then (a) all data and $\mathcal{B}$-safe terms are in $\mathcal{T}(\mathcal{F}', \emptyset)$, and (b) if s is a basic term and $s \to_{\mathcal{R}}^* t$, then $t \in \mathcal{T}(\mathcal{F}', \mathcal{V})$ and $s \to_{\mathcal{R}'}^* t$.*

Proof. Since data terms have base type, and the subterms of data terms are data terms, we have (a). Then, $\mathcal{B}$-safe terms can only be matched by rules in $\mathcal{R}'$, so Lemma 10 gives (b). ◀

Therefore we may safely assume that all elements of $\mathcal{C}$ are at most first-order.

3.2 A larger example

None of our examples so far have taken advantage of the native non-determinism of term rewriting. To demonstrate the possibilities, we consider a first-order cons-free AFS that solves the Boolean satisfiability problem (SAT). This is striking because, in Jones' language in [14], first-order programs cannot solve this problem unless P = NP, even if a non-deterministic `choose` operator is added [10]. The crucial difference is that we, unlike Jones, do not employ a call-by-value evaluation strategy.

Given n boolean variables $x_1, \dots, x_n$ and a boolean formula $\psi ::= \varphi_1 \wedge \dots \wedge \varphi_n$, the satisfiability problem considers whether there is an assignment of each x_i to $\top$ or $\bot$ such that ψ evaluates to $\top$. Here, each clause φ_i has the form $a_{i_1} \vee \dots \vee a_{i_{k_i}}$, where each literal a_{i_j} is either some x_p or $\neg x_p$. We represent this problem as a string over $I := \{0, 1, \#, ?\}$: the formula ψ is represented by $L ::= b_{1,1} \dots b_{1,n} \# b_{2,1} \dots \# b_{m,1} \dots b_{m,n} \#$, where each $b_{i,j}$ is 1 if x_j is a literal in φ_i, is 0 if $\neg x_j$ is a literal in φ_i, and is ? otherwise.

▶ **Example 13.** The satisfiability problem for $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$ is encoded as $\mathtt{10?\#?10\#}$.

Letting $\mathtt{0}, \mathtt{1}, \#, ? : [\mathtt{string}] \Rightarrow \mathtt{string}$, and assuming other declarations clear from context, we claim that the AFS in Figure 1 can reduce $\mathtt{decide}(\overline{\mathtt{L}})$ to `true` iff ψ is satisfiable.

In this AFS, we follow some of the same ideas as in Example 11. In particular, any string of the form $b_i \dots b_n \# \dots$ with each $b_j \in \{0, 1, ?\}$ is considered to represent the number i. The rules for `eq` are defined so that $\mathtt{eq}(s, t)$ tests equality of these *numbers*, not the full strings.

The key idea new to this example is that we use terms not in normal form to represent a *set* of numbers. If we are interested in numbers in $\{1, \dots, n\}$, then a set $X \subseteq \{1, \dots, n\}$ is

$$\left.\begin{array}{rcl@{\quad}rcl}
\mathtt{eq}(\#(xs),\#(ys)) & \to & \mathtt{true} & \mathtt{eq}(\#(xs),\underline{\mathtt{a}}(ys)) & \to & \mathtt{false} \\
\mathtt{eq}(\underline{\mathtt{a}}(xs),\underline{\mathtt{b}}(ys)) & \to & \mathtt{eq}(xs,ys) & \mathtt{eq}(\underline{\mathtt{a}}(xs),\#(ys)) & \to & \mathtt{false}
\end{array}\right\}\ [\![\text{for } \underline{\mathtt{a}},\underline{\mathtt{b}} \in \{\mathtt{0},\mathtt{1},\mathtt{?}\}]\!]$$

$$\begin{array}{rcl}
\mathtt{decide}(cs) & \to & \mathtt{assign}(cs,\rhd,\rhd,cs) \\
\mathtt{assign}(\#(xs),s,t,cs) & \to & \mathtt{main}(s,t,cs)
\end{array}$$

$$\left.\begin{array}{rcl}
\mathtt{assign}(\underline{\mathtt{a}}(xs),s,t,cs) & \to & \mathtt{assign}(xs,\mathtt{either}(\underline{\mathtt{a}}(xs),s),t,cs) \\
\mathtt{assign}(\underline{\mathtt{a}}(xs),s,t,cs) & \to & \mathtt{assign}(xs,s,\mathtt{either}(\underline{\mathtt{a}}(xs),t),cs)
\end{array}\right\}\ [\![\text{for } \underline{\mathtt{a}} \in \{\mathtt{0},\mathtt{1},\mathtt{?}\}]\!]$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{either}(xs,q) & \to & xs & \mathtt{either}(xs,q) & \to & q
\end{array}$$

$$\begin{array}{rcl}
\mathtt{main}(s,t,\mathtt{?}(xs)) & \to & \mathtt{main}(s,t,xs) \\
\mathtt{main}(s,t,\mathtt{0}(xs)) & \to & \mathtt{test}(s,t,xs,\mathtt{eq}(t,\mathtt{0}(xs)),\mathtt{eq}(s,\mathtt{0}(xs))) \\
\mathtt{main}(s,t,\mathtt{1}(xs)) & \to & \mathtt{test}(s,t,xs,\mathtt{eq}(s,\mathtt{1}(xs)),\mathtt{eq}(t,\mathtt{1}(xs)))
\end{array}$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{main}(s,t,\rhd) & \to & \mathtt{true} & \mathtt{test}(s,t,xs,\mathtt{true},z) & \to & \mathtt{main}(s,t,\mathtt{skip}(xs)) \\
\mathtt{main}(s,t,\#(xs)) & \to & \mathtt{false} & \mathtt{test}(s,t,xs,z,\mathtt{true}) & \to & \mathtt{main}(s,t,xs)
\end{array}$$

$$\begin{array}{rcl}
\mathtt{skip}(\#(xs)) & \to & xs \\
\mathtt{skip}(\underline{\mathtt{a}}(xs)) & \to & \mathtt{skip}(xs)\ [\![\text{for } \underline{\mathtt{a}} \in \{\mathtt{0},\mathtt{1},\mathtt{?}\}]\!]
\end{array}$$

Figure 1 A cons-free first-order AFS solving the satisfiability problem.

encoded as a pair (s,t) of terms such that, for $i \in \{1,\dots,n\}$: $s \to_{\mathcal{R}}^{*} q$ for some representation q of i if and only if $i \in X$, and $t \to_{\mathcal{R}}^{*} q$ for some representation q of i if and only if $i \notin X$.

This is possible because we do not use a call-by-value or similar reduction strategy: an evaluation of this AFS is allowed to postpone reducing such terms, and we focus on those reductions. The AFS is constructed in such a way that reductions which evaluate these "sets" too eagerly simply end in an irreducible, non-data state.

Now, an evaluation starting in $\mathtt{decide}(\overline{\mathtt{L}})$ first non-deterministically constructs a "set" X containing those boolean variables assigned $\mathtt{true}$: $\mathtt{decide}(\overline{\mathtt{L}}) \to_{\mathcal{R}}^{*} \mathtt{main}(s,t,\overline{\mathtt{L}})$. Then, the main function goes through $\overline{\mathtt{L}}$, finding for each clause a literal that is satisfied by the assignment. Encountering for instance $b_{i_j} = \mathtt{1}$, we determine if $j \in X$ by comparing both a reduct of s and of t to j. If $s \to_{\mathcal{R}}^{*}$ "j" then $j \in X$, if $t \to_{\mathcal{R}}^{*}$ "j" then $j \notin X$; in either case, we continue accordingly. If the evaluation state is incorrect, or if s or t both non-deterministically reduce to some other term, the evaluation gets stuck in a non-data normal form.

▶ **Example 14.** To solve satisfiability of $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$, we reduce $\mathtt{decide}(\overline{\mathtt{L}})$, where $L = \mathtt{10?\#?10\#}$. First, we build a valuation; the choices made by the $\mathtt{assign}$ rules are non-deterministic, but a possible reduction is $\mathtt{decide}(\overline{\mathtt{L}}) \to_{\mathcal{R}}^{*} \mathtt{main}(s,t,\overline{\mathtt{L}})$, where $s = \mathtt{either}(\overline{\mathtt{10?\#?10\#}},\rhd)$ and $t = \mathtt{either}(\overline{\mathtt{?\#?10\#}},\mathtt{either}(\overline{\mathtt{0?\#?10\#}},\rhd))$. Recall that, since $n = 3$, $\overline{\mathtt{10?\#?10\#}}$ represents 1 while $\overline{\mathtt{?\#?10\#}}$ and $\overline{\mathtt{0?\#?10\#}}$ represent 3 and 2 respectively. Thus, this corresponds to the valuation $[x_1 := \top, x_2 := \bot, x_3 := \bot]$.

Then, the main loop recurses over the problem. Note that s reduces to a term $\overline{\mathtt{10?\#}\dots}$ and t reduces to both $\overline{\mathtt{?\#}\dots}$ and $\overline{\mathtt{0?\#}\dots}$. Therefore, $\mathtt{main}(s,t,\overline{\mathtt{L}}) = \mathtt{main}(s,t,\overline{\mathtt{11?\#?01\#}}) \to_{\mathcal{R}}^{*} \mathtt{main}(s,t,\mathtt{skip}(\overline{\mathtt{1?\#?01\#}})) \to_{\mathcal{R}}^{*} \mathtt{main}(s,t,\overline{\mathtt{?01\#}})$: the first clause is confirmed since x_1 is mapped to $\top$, so the clause is removed and the loop continues with the second clause. Next, the loop passes over those variables whose assignment does not contribute to verifying this clause, until the clause is confirmed by x_3: $\mathtt{main}(s,t,\overline{\mathtt{?01\#}}) \to_{\mathcal{R}} \mathtt{main}(s,t,\overline{\mathtt{01\#}}) \to_{\mathcal{R}}^{*} \mathtt{main}(s,t,\overline{\mathtt{1\#}}) \to_{\mathcal{R}}^{*} \mathtt{main}(s,t,\mathtt{skip}(\overline{\#})) \to_{\mathcal{R}} \mathtt{main}(s,t,\rhd) \to_{\mathcal{R}} \mathtt{true}$.

Using non-determinism, the term in Example 14 could easily have been reduced to $\mathtt{false}$ instead, simply by selecting a different valuation. This is not problematic: by definition, the AFS accepts the set of satisfiable formulas if $\texttt{decide}(\overline{\texttt{L}}) \to_{\mathcal{R}}^* \texttt{true}$ if and only if L is a satisfiable formula: false negatives or reductions which do not end in a data state are allowed.

A longer example derivation is given in Appendix B of the full version of the paper.

4 Simulating $\mathrm{E}^k\mathrm{TIME}$ Turing machines

In order to see that cons-free term rewriting captures certain classes of decidable sets, we will simulate Turing Machines. For this, we use an approach very similar to that by Jones [14]. We introduce constructor symbols $\texttt{a} : [\texttt{string}] \Rightarrow \texttt{string}$ for all $a \in A$ (including the blank symbol, which we shall refer to as $\texttt{B}$) along with $\rhd$ and the booleans, $\texttt{s} : \texttt{state}$ for all $s \in S \cup \{\text{fail}\}$, $\texttt{L}, \texttt{R} : \texttt{direction}$ and $\texttt{action} : [\texttt{string} \times \texttt{direction} \times \texttt{state}] \Rightarrow \texttt{trans}$, $\texttt{end} : [\texttt{state}] \Rightarrow \texttt{trans}$, $\texttt{NA} : \texttt{trans}$. We will introduce defined symbols and rules such that, for any string $c \in (A \setminus \{\sqcup\})^*$ – represented as the term $cs := c_1(c_2(\cdots c_n(\rhd)\cdots))$ – we have:

- $\texttt{decide}(cs) \to_{\mathcal{R}}^* \texttt{true}$ if and only if $(\sqcup c \sqcup\sqcup \dots, 0, \texttt{start}) \Rightarrow^* (t, i, \texttt{accept})$ for some t, i;
- $\texttt{decide}(cs) \to_{\mathcal{R}}^* \texttt{false}$ if and only if $(\sqcup c \sqcup\sqcup \dots, 0, \texttt{start}) \Rightarrow^* (t, i, \texttt{reject})$ for some t, i.

As rules may be overlapping, it is possible that $\texttt{decide}(cs)$ will have additional normal forms, but only one normal form will be a data term.

The rough idea of the simulation is to represent non-negative integers as terms and let $\texttt{tape}(n, p)$ reduce to the symbol at position p on the tape at the start of the n^{th} step, while $\texttt{state}(n, p)$ returns the state of the machine at time n, provided the tape head is at position p. If the tape head of the machine is not at position p at time n, then $\texttt{state}(n, p)$ should return $\texttt{fail}$ instead; this makes it possible to test the position of the tape head at any given time. As the machine is deterministic, we can devise rules to compute these terms from earlier configurations.

Finding a suitable representation of integers and corresponding manipulating functions is the most intricate part of this simulation, where we may need both higher-order functions and non-deterministic rules. Therefore, let us first assume that this can be done. Then, for a Turing machine which is given to run in time bounded above by $\lambda x.P(x)$, we define the AFS in Figure 2. Note that, by construction, any occurrence of cs can only be instantiated by the input string during evaluation.

Counting

The goal, then, is to find a representation of numbers and functionality to do four things:

- calculate $[P(|cs|)]$ or an overestimation (as the machine cannot move from its final state);
- test whether a “number” represents 0;
- given $[n]$, calculate $[n-1]$, *provided* $n > 0$ – so it suffices to determine $[\max(n-1, 0)]$;
- given $[n]$, calculate $[n+1]$, *provided* $n+1 \le P(|cs|)$ as necessarily $\texttt{transition}(cs, [n], [p]) \to_{\mathcal{R}} \texttt{NA}$ when $n < p$ and n never increases – so it suffices to determine $[\min(n+1, P(|cs|))]$.

Moreover, these calculations all occur in the right-hand side of a rule containing the initial input list cs on the left, which they can therefore use (for instance to recompute $P(|cs|)$).

Rather than representing a number by a single term, we will use *tuples* of terms (which are not terms themselves, as a pairing constructor would conflict with cons-freeness). To illustrate this, suppose we represent each number n by a pair (n_1, n_2). Then the predecessor and successor function must also be split, e.g. $\texttt{pred}^1(cs, n_1, n_2) \to_{\mathcal{R}}^* n_1'$ and $\texttt{pred}^2(cs, n_1, n_2) \to_{\mathcal{R}}^* n_2'$ for (n_1', n_2') some tuple representing $n-1$. Thus, for instance the first $\texttt{test}$ rule becomes:

$$\texttt{test}(\texttt{fail}, cs, n_1, n_2, p_1, p_2) \to \texttt{findanswer}(cs, n_1, n_2, \texttt{pred}^1(cs, p_1, p_2), \texttt{pred}^2(cs, p_1, p_2))$$

$$\left.\begin{array}{rcl}\mathtt{ifelse}_{\underline{\iota}}(\mathtt{true},y,z) & \to & y\\ \mathtt{ifelse}_{\underline{\iota}}(\mathtt{false},y,z) & \to & z\end{array}\right\}\ [\![\text{for } \underline{\iota}\in\{\mathtt{string},\mathtt{state}\}]\!]$$

$$\begin{array}{rcl}\mathtt{get}(\rhd,[i],q) & \to & q\\ \mathtt{get}(\underline{\mathtt{a}}(xs),[i],q) & \to & \mathtt{ifelse}_{\mathtt{string}}([i=0],\underline{\mathtt{a}}(\rhd),\mathtt{get}(xs,[i-1],q))\quad [\![\text{for all } \underline{\mathtt{a}}\in I]\!]\\ \mathtt{inputtape}(cs,[p]) & \to & \mathtt{ifelse}_{\mathtt{string}}([p=0],\mathtt{B}(\rhd),\mathtt{get}(cs,[p-1],\mathtt{B}(\rhd)))\end{array}$$

$$\begin{array}{rcl}\mathtt{tape}(cs,[n],[p]) & \to & \mathtt{ifelse}_{\mathtt{string}}([n=0],\mathtt{inputtape}(cs,[p]),\mathtt{tapex}(cs,[n-1],[p]))\\ \mathtt{tapex}(cs,[n],[p]) & \to & \mathtt{tapey}(cs,[n],[p],\mathtt{transition}(cs,[n],[p]))\end{array}$$

$$\begin{array}{rclrcl}\mathtt{tapey}(cs,[n],[p],\mathtt{action}(q,d,s)) & \to & q & \mathtt{tapey}(cs,[n],[p],\mathtt{NA}) & \to & \mathtt{tape}(cs,[n],[p])\\ & & & \mathtt{tapey}(cs,[n],[p],\mathtt{end}(s)) & \to & \mathtt{tape}(cs,[n],[p])\end{array}$$

$$\begin{array}{rcl}\mathtt{state}(cs,[n],[p]) & \to & \mathtt{ifelse}_{\mathtt{state}}([n=0],\mathtt{state0}(cs,[p]),\mathtt{statex}(cs,[n-1],[p]))\\ \mathtt{state0}(cs,[p]) & \to & \mathtt{ifelse}_{\mathtt{state}}([p=0],\mathtt{start},\mathtt{fail})\\ \mathtt{statex}(cs,[n],[p]) & \to & \mathtt{statey}(\mathtt{transition}(cs,[n],[p-1]),\mathtt{transition}(cs,[n],[p]),\\ & & \qquad\mathtt{transition}(cs,[n],[p+1]))\end{array}$$

$$\begin{array}{rclrcl}\mathtt{statey}(\mathtt{action}(q,\mathtt{R},s),y,z) & \to & s & \mathtt{statey}(\mathtt{NA},\mathtt{action}(q,d,s),z) & \to & \mathtt{fail}\\ \mathtt{statey}(\mathtt{action}(q,\mathtt{L},s),y,z) & \to & \mathtt{fail} & \mathtt{statey}(\mathtt{NA},\mathtt{NA},\mathtt{action}(q,\mathtt{L},s)) & \to & s\\ \mathtt{statey}(\mathtt{end}(s),y,z) & \to & \mathtt{fail} & \mathtt{statey}(\mathtt{NA},\mathtt{NA},\mathtt{action}(q,\mathtt{R},s)) & \to & \mathtt{fail}\\ \mathtt{statey}(\mathtt{NA},\mathtt{end}(s),z) & \to & s & \mathtt{statey}(\mathtt{NA},\mathtt{NA},\mathtt{end}(s)) & \to & \mathtt{fail}\end{array}$$

$$\begin{array}{rcl}\mathtt{transition}(cs,[n],[p]) & \to & \mathtt{transitionhelp}(\mathtt{state}(cs,[n],[p]),\mathtt{tape}(cs,[n],[p]))\\ \mathtt{transitionhelp}(\mathtt{fail},q) & \to & \mathtt{NA}\\ \mathtt{transitionhelp}(\underline{\mathtt{s}},\underline{\mathtt{r}}(\rhd)) & \to & \mathtt{action}(\underline{\mathtt{w}}(\rhd),\underline{\mathtt{d}},\underline{\mathtt{t}})\qquad [\![\text{for all } \underline{\mathtt{s}}\overset{\underline{\mathtt{r}}/\underline{\mathtt{w}}\ \underline{\mathtt{d}}}{\Longrightarrow}\underline{\mathtt{t}}\in T]\!]\\ \mathtt{transitionhelp}(\underline{\mathtt{s}},q) & \to & \mathtt{end}(\underline{\mathtt{s}})\qquad [\![\text{for } \underline{\mathtt{s}}\in\{\mathtt{accept},\mathtt{reject}\}]\!]\end{array}$$

$$\begin{array}{rcl}\mathtt{decide}(cs) & \to & \mathtt{findanswer}(cs,[P(|cs|)],[P(|cs|)])\\ \mathtt{findanswer}(cs,[n],[p]) & \to & \mathtt{test}(\mathtt{state}(cs,[n],[p]),cs,[n],[p])\\ \mathtt{test}(\mathtt{fail},cs,[n],[p]) & \to & \mathtt{findanswer}(cs,[n],[p-1])\\ \mathtt{test}(\mathtt{accept},cs,[n],[p]) & \to & \mathtt{true}\\ \mathtt{test}(\mathtt{reject},cs,[n],[p]) & \to & \mathtt{false}\end{array}$$

Figure 2 Simulating a deterministic Turing Machine running in $\lambda x.P(x)$ time.

Following Jones [14], we use the notion of a *counting module* which provides an AFS with a representation of a counting function and a means of computing. Counting modules can be composed, making it possible to count to greater numbers. Due to the laxity of term rewriting, our constructions are technically quite different from those of [14].

▶ **Definition 15** (Counting Module). Write $\mathcal{F}=\mathcal{C}\cup\mathcal{D}$ for the signature in Figure 2. For P a function from $\mathbb{N}$ to $\mathbb{N}$, a P-counting module of *order* k is a tuple $C_\pi ::= (\vec{\sigma},\Sigma,R,A,\langle\cdot\rangle)$ s.t.:

- $\vec{\sigma}$ is a sequence of types $\sigma_1\times\dots\times\sigma_a$ where each σ_i has order at most $k-1$;
- Σ is a k^{th}-order signature disjoint from $\mathcal{F}$, with designated symbols $\mathtt{zero}_\pi : [\mathtt{string}\times\vec{\sigma}]\Rightarrow\mathtt{bool}$ and, for $1\le i\le a$ with $\sigma_i=\tau_1\Rightarrow\dots\Rightarrow\tau_m\Rightarrow\iota$ symbols $\mathtt{pred}^i_\pi,\mathtt{suc}^i_\pi,\mathtt{inv}^i_\pi : [\mathtt{string}\times\vec{\sigma}\times\vec{\tau}]\Rightarrow\iota$ and $\mathtt{seed}^i_\pi : [\mathtt{string}\times\vec{\tau}]\Rightarrow\kappa$;
- R is a set of cons-free rules $f(\vec{\ell})\to r$ with $f\in\Sigma$, each $\ell_i\in\mathcal{T}(\mathcal{C},\mathcal{V})$ and $r\in\mathcal{T}(\mathcal{C}\cup\Sigma,\mathcal{V})$;
- for every string $cs\subseteq I^+$, the set $A_{cs}\subseteq\{(s_1,\dots,s_a)\in\mathcal{T}(\mathcal{C}\cup\Sigma)^a \Vdash s_j:\sigma_j \text{ for } 1\le j\le a\}$;
- for every string cs, $\langle\cdot\rangle_{cs}$ is a surjective mapping from A_{cs} to $\{0,\dots,P(|cs|)-1\}$;
- writing e.g. $\mathtt{pred}^i_\pi[\vec{s}]:\sigma_i$ for the term $\lambda\vec{y}.\mathtt{pred}^i_\pi(\vec{s},\vec{y})$, the following properties are satisfied:
  - $(\mathtt{seed}^1_\pi[cs],\dots,\mathtt{seed}^a_\pi[cs])\in A_{cs}$ and $\langle(\mathtt{seed}^1_\pi[cs],\dots,\mathtt{seed}^a_\pi[cs])\rangle_{cs}=P(|cs|)-1$

  and for all $(s_1,\dots,s_a)\in A_{cs}$ with $\langle(s_1,\dots,s_a)\rangle_{cs}=m$:

- $(\mathtt{pred}^1_\pi[cs,\vec{s}],\dots,\mathtt{pred}^a_\pi[cs,\vec{s}])$ and $(\mathtt{suc}^1_\pi[cs,\vec{s}],\dots,\mathtt{suc}^a_\pi[cs,\vec{s}])$ and $(\mathtt{inv}^1_\pi[cs,\vec{s}],\dots,\mathtt{inv}^a_\pi[cs,\vec{s}])$ are all in A_{cs}
- $\langle(\mathtt{pred}^1_\pi[cs,\vec{s}],\dots,\mathtt{pred}^a_\pi[cs,\vec{s}])\rangle_{cs} = \max(m-1,0)$
- $\langle(\mathtt{suc}^1_\pi[cs,\vec{s}],\dots,\mathtt{suc}^a_\pi[cs,\vec{s}])\rangle_{cs} = \min(m+1,P(|cs|)-1)$
- $\langle(\mathtt{inv}^1_\pi[cs,\vec{s}],\dots,\mathtt{inv}^a_\pi[cs,\vec{s}])\rangle_{cs} = P(|cs|)-1-m$
- $\mathtt{zero}_\pi(cs,\vec{s}) \to^*_R \mathtt{true}$ iff $m=0$ and $\mathtt{zero}_\pi(cs,\vec{s}) \to^*_R \mathtt{false}$ iff $m>0$
- if each $s_i \to^*_R t_i$ and $(t_1,\dots,t_a) \in A_{cs}$, then also $\langle(t_1,\dots,t_a)\rangle_{cs} = m$.

It is not hard to see how we would use a P-counting module in the AFS of Figure 2; this results in a kth-order AFS for a kth-order module. Note that this works even if some number representations $(s_1,\dots,s_a)$ are not in normal form: even if we reduce $\vec{s}$ to some tuple $\vec{t}$, the result of the zero test cannot change from true to false or vice versa. Since the algorithm relies heavily on these tests, we may safely assume that terms representing numbers are reduced in a lazy way – as we did in Section 3.2 for the arguments s and t of main.

▶ **Lemma 16.** *There is a first-order $(\lambda n.2^{n+1})$-counting module.*

Proof. Like in Section 3.2, we will represent a set of numbers – or rather, its encoding as a bit-sequence – by a pair of terms. We let $C_{\mathsf{e}} := (\mathtt{string}\times\mathtt{string},\Sigma,R,A,\langle\cdot\rangle)$, where:

- A_{cs} contains all pairs (s,t) such that (a) all data terms q such that $s \to^*_R q$ or $t \to^*_R q$ are subterms of cs, and (b) for each $q \trianglelefteq cs$ either $s \to^*_R q$ or $t \to^*_R q$, but not both.
- Writing $cs = c_N(\dots(c_1(\rhd))\dots)$, we let $cs_0 = \rhd$, $cs_1 = c_1(\rhd)$ and so on. We let $\langle(s,t)\rangle_{cs} = \sum_{i=0}^{N}\{2^{N-i} \mid s \to^*_R cs_i\}$. That is, $\langle(s,t)\rangle_{cs}$ is the number represented by the bit-sequence $b_0\dots b_N$ where $b_i = 1$ iff $s \to^*_R cs_i$, iff not $t \to^*_R cs_i$ (with b_N the least significant digit).
- Σ consists of the defined symbols introduced in R, which we construct below.

As in Section 3.2, we use non-deterministic selection functions to construct (s,t):

$$\mathtt{either}(x,y) \to x \qquad \mathtt{either}(x,y) \to y \qquad \bot \to \bot$$

The symbol $\bot$ will be used for terms which do not reduce to any data (the $\bot \to \bot$ rule is used to force $\bot \in \mathcal{D}$). For the remaining functions, we consider bitvector arithmetic. First, $2^{N+1}-1$ corresponds to the bit-sequence where each $b_i = 1$:

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{seed}^1_{\mathsf{e}}(cs) & \to & \mathtt{all}(cs,\bot) & \mathtt{all}(\rhd,q) & \to & \mathtt{either}(\rhd,q) \\
\mathtt{seed}^2_{\mathsf{e}}(cs) & \to & \bot & \mathtt{all}(\underline{\mathtt{a}}(xs),q) & \to & \mathtt{all}(xs,\mathtt{either}(\underline{\mathtt{a}}(xs),q))\ [\![\text{for } \underline{\mathtt{a}} \in \mathtt{I}]\!]
\end{array}$$

Here, $\mathtt{I} = \{\mathtt{a} \mid a \in I\}$. The inverse function is obtained by flipping the sequence's bits:

$$\mathtt{inv}^1_{\mathsf{e}}(cs,s,t) \to t \qquad \mathtt{inv}^1_{\mathsf{e}}(cs,s,t) \to s$$

In order to define $\mathtt{zero}_{\mathsf{e}}$, we must test the value of all bits in the sequence. This is done by forcing an evaluation from s or t to some data term. This test is constructed in such a way that both true and false results necessarily reflect the state of s and t; any undesirable non-deterministic choices lead to the evaluation getting stuck.

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{eqLen}(\rhd,\rhd) & \to & \mathtt{true} & \mathtt{eqLen}(\rhd,\underline{\mathtt{a}}(ys)) & \to & \mathtt{false} \\
\mathtt{eqLen}(\underline{\mathtt{a}}(xs),\underline{\mathtt{b}}(ys)) & \to & \mathtt{eqLen}(xs,ys) & \mathtt{eqLen}(\underline{\mathtt{a}}(xs),\rhd) & \to & \mathtt{false}
\end{array} \left.\vphantom{\begin{array}{c}a\\b\end{array}}\right\}\ [\![\text{for } \underline{\mathtt{a}},\underline{\mathtt{b}} \in I]\!]$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{bitset}(xs,s,t) & \to & \mathtt{test}(\mathtt{eqLen}(xs,s),\mathtt{eqLen}(xs,t)) & \mathtt{test}(\mathtt{true},x) & \to & \mathtt{true} \\
 & & & \mathtt{test}(x,\mathtt{true}) & \to & \mathtt{false}
\end{array}$$

Then $\mathtt{zero}_{\mathsf{e}}$ simply tests whether the bit is unset for each sublist.

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{zero}_{\mathsf{e}}(xs,s,t) & \to & \mathtt{zo}(xs,s,t,\mathtt{bitset}(xs,s,t)) & \mathtt{zo}(xs,s,t,\mathtt{true}) & \to & \mathtt{false} \\
\mathtt{zo}(\underline{\mathtt{a}}(xs),s,t,\mathtt{false}) & \to & \mathtt{zero}_{\mathsf{e}}(xs,s,t)\ [\![\text{for } \underline{\mathtt{a}} \in I]\!] & \mathtt{zo}(\rhd,s,t,\mathtt{false}) & \to & \mathtt{true}
\end{array}$$

For the predecessor function, note that the predecessor of a bit-sequence $b_0 \dots b_{i-1} b 1 0 \dots 0$ is $b_0 \dots b_{i-1} 0 1 \dots 1$. We first define a helper function $\mathtt{copy}$ to copy $b_0 \dots b_{i-1}$:

$$\begin{array}{rcl}
\mathtt{copy}(xs, s, t, \mathtt{false}) & \to & \mathtt{maybeadd}(xs, \mathtt{bitset}(xs, s, t), \mathtt{copy}(\mathtt{tl}(xs), s, t, \mathtt{empty}(xs))) \\
\mathtt{copy}(xs, s, t, \mathtt{true}) & \to & \bot \qquad\qquad \mathtt{maybeadd}(xs, \mathtt{true}, q) \ \to\ \mathtt{either}(xs, q) \\
 & & \qquad\qquad\quad \mathtt{maybeadd}(xs, \mathtt{false}, q) \ \to\ q \\
\mathtt{empty}(\rhd) & \to & \mathtt{true} \qquad\qquad\qquad \mathtt{tl}(\rhd) \ \to\ \rhd \\
\mathtt{empty}(\underline{\mathtt{a}}(x)) & \to & \mathtt{false}\ [\![\text{for } \underline{\mathtt{a}} \in \mathtt{I}]\!] \quad \mathtt{tl}(\underline{\mathtt{a}}(x)) \ \to\ x \ [\![\text{for } \underline{\mathtt{a}} \in \mathtt{I}]\!]
\end{array}$$

Then $\mathtt{copy}(xs_{\max(i-1,0)}, s, t, [i = 0])$ reduces to those xs_j with $0 \le j < i$ where $b_j = 1$, and $\mathtt{copy}(xs_{\max(i-1,0)}, t, s, [i = 0])$ to those with $b_j = 0$. This works because s and t are each other's complement. To define $\mathtt{pred}$, we first handle the zero case:

$$\begin{array}{l}
\mathtt{pred}^{\underline{\mathtt{i}}}_{\mathtt{e}}(cs, s, t) \ \to\ \mathtt{pz}^{\underline{\mathtt{i}}}(cs, s, t, \mathtt{zero}_{\mathtt{e}}(cs, s, t)) \ [\![\text{for } \underline{\mathtt{i}} \in \{1, 2\}]\!] \\
\mathtt{pz}^1(cs, s, t, \mathtt{true}) \ \to\ s \qquad \mathtt{pz}^1(cs, s, t, \mathtt{false}) \ \to\ \mathtt{pmain}^1(cs, s, t, \mathtt{bitset}(cs, s, t)) \\
\mathtt{pz}^2(cs, s, t, \mathtt{true}) \ \to\ t \qquad \mathtt{pz}^2(cs, s, t, \mathtt{false}) \ \to\ \mathtt{pmain}^2(cs, s, t, \mathtt{bitset}(cs, s, t))
\end{array}$$

Then, $\mathtt{pmain}(xs_N, s, t, [b_N = 1])$ flips the bits $b_N, b_{N-1}, \dots$ until an index is encountered where $b_i = 1$; this last bit is flipped, and the remaining bits copied. Formally:

$$\begin{array}{rcl}
\mathtt{pmain}^1(xs, s, t, \mathtt{true}) & \to & \mathtt{copy}(\mathtt{tl}(xs), s, t, \mathtt{empty}(xs)) \\
\mathtt{pmain}^2(xs, s, t, \mathtt{true}) & \to & \mathtt{either}(xs, \mathtt{copy}(\mathtt{tl}(xs), t, s, \mathtt{empty}(xs))) \\
\mathtt{pmain}^1(xs, s, t, \mathtt{false}) & \to & \mathtt{either}(xs, \mathtt{pmain}^1(\mathtt{tl}(xs), s, t, \mathtt{bitset}(\mathtt{tl}(xs), s, t))) \\
\mathtt{pmain}^2(xs, s, t, \mathtt{false}) & \to & \mathtt{pmain}^2(\mathtt{tl}(xs), s, t, \mathtt{bitset}(\mathtt{tl}(xs), s, t))
\end{array}$$

Finally, we observe that $x + 1 = N - ((N - x) - 1)$ and for $x = N$ also $\min(x + 1, N) = N - (\max((N - x) - 1, 0))$. Thus, we may define $\mathtt{suc}(b)$ as $\mathtt{inv}(\mathtt{pred}(\mathtt{inv}(x)))$. Taking pairing into account and writing out the definition, this simplifies to:

$$\mathtt{suc}^1(cs, s, t) \ \to\ \mathtt{pred}^2(cs, t, s) \qquad \mathtt{suc}^2(cs, s, t) \ \to\ \mathtt{pred}^1(cs, t, s)$$ ◀

Having Lemma 16 as a basis, we can define composite modules. Here, we give fewer details than for Lemma 16 as the constructions use many of the same ideas.

▶ **Lemma 17.** *If there exist a P-counting module C_π and a Q-counting module C_ρ, both of order at most k, then there is a $(\lambda n.P(n) \cdot Q(n))$-counting module $C_{\pi \cdot \rho}$ of order at most k.*

Proof Sketch. Let $C_\pi ::= ([\sigma_1 \times \dots \times \sigma_a], \Sigma^\pi, R^\pi, A^\pi, \langle\cdot\rangle^\pi)$ and $C_\rho ::= ([\tau_1 \times \dots \times \tau_b], \Sigma^\rho, R^\rho, A^\rho, \langle\cdot\rangle^\rho)$. We will, essentially, represent the numbers $i \in \{0, \dots, P(|cs|) \cdot Q(|cs|) - 1\}$ by a pair (i_1, i_2) with $0 \le i_1 < P(|cs|)$ and $0 \le i_2 < Q(|cs|)$, such that $i = i_1 \cdot Q(|cs|) + i_2$. This is done by defining $A^{\pi \cdot \rho}_{cs} = \{(u_1, \dots, u_a, v_1, \dots, v_b) \mid (u_1, \dots, u_a) \in A^\pi_{cs} \wedge (v_1, \dots, v_b) \in A^\rho_{cs}\}$, and $\langle(\vec{u}, \vec{v})\rangle^{\pi \cdot \rho}_{cs} = \langle(\vec{u})\rangle^\pi_{cs} \cdot Q(|cs|) + \langle(\vec{v})\rangle^\rho_{cs}$. The signature of defined symbols and rules of $C_{\pi \cdot \rho}$ are straightforwardly defined as well, extending those in C_π and C_ρ; for instance:

$$\begin{array}{l}
\mathtt{zero}_{\pi \cdot \rho}(cs, u_1, \dots, u_a, v_1, \dots, v_b) \ \to\ \mathtt{and}(\mathtt{zero}_\pi(cs, u_1, \dots, u_a), \mathtt{zero}_\rho(cs, v_1, \dots, v_b)) \\
\mathtt{and}(\mathtt{true}, x) \ \to\ x \qquad \mathtt{and}(\mathtt{false}, y) \ \to\ \mathtt{false}
\end{array}$$ ◀

▶ **Lemma 18.** *If there is a P-counting module C_π of order k, then there is a $(\lambda n.2^{P(n)})$-counting module $C_{p[\pi]}$ of order $k + 1$.*

Proof Sketch. We represent every bitstring $b_{P(|cs|)-1 \cdots b_0}$ as a function of type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_a \Rightarrow \mathtt{bool}$. The various functions are defined as bitvector operations. For example:

$$\mathtt{seed}_{\mathtt{p}[\pi]}(cs, k_1, \dots, k_a) \ \to\ \mathtt{true} \qquad \mathtt{inv}_{\mathtt{p}[\pi]}(cs, F, k_1, \dots, k_a) \ \to\ \mathtt{not}(F \cdot k_1 \cdots k_a)$$

$$
\begin{array}{rcl}
\mathtt{zero}_{\mathtt{p}[\pi]}(cs, F) & \to & \mathtt{zero'}_{\mathtt{p}[\pi]}(cs, \mathtt{seed}^1_\pi[cs], \dots, \mathtt{seed}^a_\pi[cs], F) \\
\mathtt{zero'}_{\mathtt{p}[\pi]}(cs, k_1, \dots, k_a, F) & \to & \mathtt{ztest}_{\mathtt{p}[\pi]}(F \cdot k_1 \cdots k_a, \mathtt{zero}_\pi(cs, k_1, \dots, k_a), cs, \\
 & & \quad k_1, \dots, k_a, F) \\
\mathtt{ztest}_{\mathtt{p}[\pi]}(\mathtt{true}, z, cs, \vec{k}, F) & \to & \mathtt{false} \\
\mathtt{ztest}_{\mathtt{p}[\pi]}(\mathtt{false}, \mathtt{true}, cs, \vec{k}, F) & \to & \mathtt{true} \\
\mathtt{ztest}_{\mathtt{p}[\pi]}(\mathtt{false}, \mathtt{false}, cs, \vec{k}, F) & \to & \mathtt{zero'}_{\mathtt{p}[\pi]}(cs, \mathtt{pred}^1_\pi[cs, \vec{k}], \dots, \mathtt{pred}^a_\pi[cs, \vec{k}], F) \quad ◀
\end{array}
$$

Note that, for instance, $\mathtt{seed}_{\mathtt{p}[\pi]}[cs]$ is $\lambda k_1 \dots k_a.\mathtt{seed}_{\mathtt{p}[\pi]}(cs, k_1, \dots, k_a)$: the additional parameters k_i should be seen as indexing the result of the function.

We obtain:

▶ **Theorem 19.** *Any decision problem in E^{k}TIME can be accepted by a k^{th}-order AFS.*

Proof. Following the construction in this section, it suffices if we can find a k^{th}-order counting module counting up to $\exp_2^k(a \cdot n)$ where n is the size of the input and a a fixed positive integer. Lemma 16 gives a first-order $\lambda n.2^{n+1}$-counting module, and by iteratively using Lemma 17 we obtain $\lambda n.(2^{n+1})^a = \lambda n.2^{a(n+1)}$ for any a. Iteratively applying Lemma 18 on the result gives a k^{th}-order $\lambda n.\exp_2^k(a \cdot (n+1))$-counting module. ◀

5 Finding normal forms

In the previous section we have seen that every function in E^{k}TIME can be implemented by a cons-free k^{th}-order AFS. Towards a characterization result, we must therefore show the converse: that every function implemented by a cons-free k^{th}-order AFS is in E^{k}TIME.

To achieve this goal, we will now give an algorithm that, on input any basic term in an AFS of order k, will output its set of data normal forms in E^{k}TIME in the size of the term.

A key idea is to associate terms of higher-order type to functions. We define:

$$
\begin{array}{rcl}
[\![\iota]\!] & = & \mathbb{P}(\{s \mid s \in \mathcal{B} \wedge \vdash s : \iota\}) \quad \text{for } \iota \in \mathcal{S} \quad (\text{so a set of subsets of } \mathcal{B}) \\
[\![\sigma \Rightarrow \tau]\!] & = & [\![\tau]\!]^{[\![\sigma]\!]} \quad (\text{so the set of functions from } [\![\sigma]\!] \text{ to } [\![\tau]\!])
\end{array}
$$

Intuitively, an element of $[\![\iota]\!]$ represents a set of possible reducts of a term $s : \iota$, while an element of $[\![\sigma \Rightarrow \tau]\!]$ represents the function defined by some $\lambda x.s : \sigma \Rightarrow \tau$. Since – as induction on the structure of σ shows – each $[\![\sigma]\!]$ is *finite*, we can define the following algorithm to find all normal forms of a given basic term. In the algorithm, we build functions $\mathsf{Confirmed}^0, \mathsf{Confirmed}^1, \dots$, each mapping statements $f(A_1, \dots, A_n) \approx t$ to a value in $\{\top, \bot\}$. Intuitively, $\mathsf{Confirmed}^i[f(\vec{A}) \approx t]$ denotes whether, in step i in the algorithm, we have confirmed that $f(s_1, \dots, s_n) \to^*_{\mathcal{R}} t$, where each A_i represents the corresponding s_i.

▶ **Algorithm 20.**

Input: A basic term $s = g(t_1, \dots, t_m)$.

Output: The set of data normal forms of s. Note that this set may be empty.

Set $\mathcal{B} := \mathcal{B}_s$. For all $f : [\sigma_1 \times \cdots \times \sigma_n] \Rightarrow \iota \in \mathcal{D}$, all $A_1 \in [\![\sigma_1]\!], \dots, A_n \in [\![\sigma_n]\!]$, all $t \in [\![\iota]\!]$, we let $\mathsf{Confirmed}^0[f(A_1, \dots, A_n) \approx t] := \bot$. Now, for all such $f, \vec{A}, t$ and all $i \in \mathbb{N}$:

- if $\mathsf{Confirmed}^i[f(\vec{A}) \approx t] = \top$, then $\mathsf{Confirmed}^{i+1}[f(\vec{A}) \approx t] := \top$;
- otherwise, for all rules $f(\ell_1, \dots, \ell_n) \to r \in \mathcal{R}$, for all substitutions γ on domain $FV(f(\vec{\ell})) \setminus \{\vec{\ell}\}$ (so on those variables occurring below constructors) such that $\ell_j\gamma \in A_j$ for all j with ℓ_j not a variable (A_j is a set of terms since ℓ_j, a non-variable proper constructor term, must have base type), let η be the function such that for each $\ell_j \in \mathcal{V}$, $\eta(\ell_j) = A_j$, and test whether $t \in \mathcal{NF}^i(r\gamma, \eta)$. If there are a rule and substitution where this test succeeds, let $\mathsf{Confirmed}^{i+1}[f(\vec{A}) \approx t] := \top$, otherwise let $\mathsf{Confirmed}^{i+1}[f(\vec{A}) \approx t] := \bot$.

Here, $\mathcal{NF}^i(s, \eta)$ is defined recursively for $\mathcal{B}$-safe terms s and functions η mapping all variables $x : \sigma$ in $FV(s)$ to an element of $[\![\sigma]\!]$, as follows:

- if s is a data term, then $\mathcal{NF}^i(s, \eta) := \{s\}$;
- if s is a variable, then $\mathcal{NF}^i(s, \eta) := \eta(s)$;
- if $s = f(s_1, \dots, s_n)$ with $f \in \mathcal{D}$, then $\mathcal{NF}^i(s, \eta)$ is the set of all $t \in \mathcal{B}$ such that $\mathsf{Confirmed}^i[f(\mathcal{NF}^i(s_1, \eta), \dots, \mathcal{NF}^i(s_n, \eta)) \approx t] = \top$;
- if $s = u \cdot v$, then $\mathcal{NF}^i(s, \eta) = \mathcal{NF}^i(u, \eta)(\mathcal{NF}^i(v, \eta))$;
- if $s =_\alpha \lambda x.t : \sigma \Rightarrow \tau$ where $x \notin \mathtt{domain}(\eta)$, then $\mathcal{NF}^i(s, \eta) :=$ the function mapping $A \in [\![\sigma]\!]$ to $\mathcal{NF}^i(t, \eta \cup [x := A])$.

When $\mathsf{Confirmed}^{i+1}[f(\vec{A}) \approx t] = \mathsf{Confirmed}^i[f(\vec{A}) \approx t]$ for all statements, the algorithm ends; we let $I := i + 1$ and return $\{t \in \mathcal{B} \mid \mathsf{Confirmed}^I[g(\{t_1\}, \dots, \{t_m\}) \approx t] = \top\}$.

As $\mathcal{D}$, $\mathcal{B}$ and all $[\![\sigma_i]\!]$ are all finite, and the number of positions at which $\mathsf{Confirmed}^i$ is $\top$ increases in every step, the algorithm always terminates. The intention is that $\mathsf{Confirmed}^I$ reflects rewriting for basic terms. This result is stated formally in Theorem 22.

▶ **Example 21.** Consider the palindrome AFS in Example 11, with starting term $s = \mathtt{1}(\mathtt{0}(\rhd))$. Then $\mathcal{B}_s = \{\mathtt{1}(\mathtt{0}(\rhd)), \mathtt{0}(\rhd), \rhd, \mathtt{true}, \mathtt{false}\}$. Then we have $[\![\mathtt{bool}]\!] = \{\emptyset, \{\mathtt{true}\}, \{\mathtt{false}\}, \{\mathtt{true}, \mathtt{false}\}\}$ and $[\![\mathtt{string}]\!]$ is the set containing all eight subsets of $\{\mathtt{1}(\mathtt{0}(\rhd)), \mathtt{0}(\rhd), \rhd\}$. Thus, there are $8 \cdot 8 \cdot 2$ statements of the form $\mathtt{palindrome}(A, B) \approx t$, $4 \cdot 4 \cdot 2$ of the form $\mathtt{and}(A, B) \approx t$ and so on, totalling 432 statements to be considered in every step.

We consider one step, determining $\mathsf{Confirmed}^1[\mathtt{chk_1}(\{\mathtt{1}(\mathtt{0}(\rhd))\}, \{\mathtt{0}(\rhd), \rhd\}) \approx \mathtt{true}]$. There are two viable combinations of a rule and a substitution: $\mathtt{chk_1}(\mathtt{1}(xs), \mathtt{0}(ys)) \to \mathtt{chk_1}(xs, ys)$ with substitution $\gamma = [xs := \mathtt{0}(\rhd), ys := \rhd]$ and $\mathtt{chk_1}(\mathtt{1}(xs), \rhd) \to \mathtt{true}$ with $\gamma = [xs := \mathtt{0}(\rhd)]$. Consider the first. As there are no functional variables, η is empty and we need to determine whether $\mathtt{true} \in \mathcal{NF}^1(\mathtt{chk_1}(\mathtt{0}(\rhd), \rhd), \emptyset)$. This fails, because $\mathsf{Confirmed}^0[\xi] = \bot$ for all statements ξ. However, the check for the second rule, $\mathtt{true} \in \mathcal{NF}^1(\mathtt{true}, \emptyset)$, succeeds. Thus, we mark $\mathsf{Confirmed}^1[\mathtt{chk_1}(\{\mathtt{1}(\mathtt{0}(\rhd))\}, \{\mathtt{0}(\rhd), \rhd\}) \approx \mathtt{true}] = \top$.

▶ **Theorem 22.** *Let $f : [\iota_1 \times \dots \times \iota_n] \Rightarrow \kappa \in \mathcal{D}$ and $s_1 : \iota_1, \dots, s_n : \iota_n, t : \kappa$ be data terms. Then $\mathsf{Confirmed}^I[f(\{s_1\}, \dots, \{s_n\}) \approx t] = \top$ if and only if $f(\vec{s}) \to^*_{\mathcal{R}} t$.*

Proof Sketch. Define a labeled variation of $\mathcal{R}$:

$$\mathcal{R}_{\mathtt{lab}} = \{f_{i+1}(\vec{\ell}) \to \mathsf{label}_i(r) \mid f(\vec{\ell}) \to r \in \mathcal{R} \wedge i \in \mathbb{N}\} \cup \{f_{i+1}(\vec{x}) \to f_i(\vec{x}) \mid f \in \mathcal{D} \wedge i \in \mathbb{N}\}$$

Here label_i replaces each defined symbol f by a symbol f_i. Then $\mathcal{R}_{\mathtt{lab}}$ is infinite, and $f(\vec{s}) \to^*_{\mathcal{R}} t$ iff some $f_i(\vec{s}) \to^*_{\mathcal{R}_{\mathtt{lab}}} t$. Furthermore, $\to_{\mathcal{R}_{\mathtt{lab}}}$ is terminating (even if $\to_{\mathcal{R}}$ is not!) as is provable using, e.g., the *Computability Path Ordering* [9]. Thus, $\to_{\mathcal{R}_{\mathtt{lab}}}$ is a well-founded binary relation on the set of labeled terms, and we can hence perform induction.

Consider the arguments passed to $\mathsf{Confirmed}^i$ in the recursive process: $\mathcal{NF}^i$ is defined using tests of the form $\mathsf{Confirmed}^i[f(\mathcal{NF}^i(s_1, \eta), \dots, \mathcal{NF}^i(s_n, \eta))] = \top$, where each $\eta(x)$ itself has the form $\mathcal{NF}^j(t, \eta')$. To formally describe this, let an *$\mathcal{NF}$-substitution* be recursively defined as a mapping from some (possibly empty) set $V \subseteq \mathcal{V}$ such that for each $x : \sigma \in V$ there are an $\mathcal{NF}$-substitution δ and a term s with $\vdash s : \sigma$ such that $\eta(x) = \mathcal{NF}^j(s, \delta)$ for some j. For an $\mathcal{NF}$-substitution η on domain V, we define $\overline{\eta}(x) = x$ for $x \notin V$, and $\overline{\eta}(x) = \mathsf{label}_j(s)\overline{\zeta}$ for $x \in V$ with $\eta(x) = \mathcal{NF}^j(s, \zeta)$. Then the following two claims can be derived by mutual induction on q ordered with $\to_{\mathcal{R}_{\mathtt{lab}}} \cup \rhd$ (all η_j and ζ are $\mathcal{NF}$-substitutions):

- $\mathsf{Confirmed}^i[f(\mathcal{NF}^{j_1}(s_1, \eta_1), \dots, \mathcal{NF}^{j_n}(s_n, \eta_n)) \approx t] = \top$ *if and only if* $q := f_i(\mathsf{label}_{j_1}(s_1)\overline{\eta_1}, \dots, \mathsf{label}_{j_n}(s_n)\overline{\eta_n}) \to^*_{\mathcal{R}_{\mathtt{lab}}} t$;

- *$t \in \mathcal{NF}^i(u,\zeta)(\mathcal{NF}^{j_1}(s_1,\eta_1),\dots,\mathcal{NF}^{j_n}(s_n,\eta_n))$ if and only if $q := (\mathsf{label}_i(u)\overline{\zeta}) \cdot \mathsf{label}_{j_1}(s_1)\overline{\eta_1} \cdots \mathsf{label}_{j_n}(s_n)\overline{\eta_n} \to^*_{\mathcal{R}_{\mathtt{lab}}} t$.*

Since, if we refrain from stopping the process in step I, we have $\mathsf{Confirmed}^I = \mathsf{Confirmed}^{I+1} = \mathsf{Confirmed}^{I+2} = \dots$, the theorem follows because $f(\vec{s}) \to^*_{\mathcal{R}} t$ iff some $f_i(\vec{s}) \to^*_{\mathcal{R}_{\mathtt{lab}}} t$. ◀

It remains to prove that Algorithm 20 runs sufficiently fast.

▶ **Theorem 23.** *If $(\mathcal{F},\mathcal{R})$ has order k, then Algorithm 20 runs in time $O(\exp_2^k(m \cdot n))$ for some m.*

Proof. Write $N := |\mathcal{B}|$. As $\mathcal{R}$ and $\mathcal{F}$ are fixed, N is linear in the size of the only input, s. We claim that if $k, i \in \mathbb{N}$ are such that σ has at most *order* k, and the *longest sequence* $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \iota$ occurring in σ has length $n+1 \le i$, then $\mathtt{card}([\![\sigma]\!]) \le \exp_2^{k+1}(i^k \cdot N)$.

(Proof of claim.) Observe first that $\mathbb{P}(\mathcal{B})$ has cardinality 2^N. Proceed by induction on the form of σ. Note that we can write σ in the form $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \iota$ with $n < i$ and each σ_j having order at most $k-1$ (as $n = 0$ when given a 0th-order type). We have:

$$\begin{aligned}
\mathtt{card}([\![\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \iota]\!]) &= \mathtt{card}((\cdots([\![\iota]\!]^{[\![\sigma_n]\!]})^{[\![\sigma_{n-1}]\!]}\cdots)^{[\![\sigma_1]\!]}) = \mathtt{card}([\![\iota]\!])^{\mathtt{card}([\![\sigma_n]\!])\cdots\mathtt{card}([\![\sigma_1]\!])} \\
&\le 2^{N\cdot\mathtt{card}([\![\sigma_n]\!])\cdots\mathtt{card}([\![\sigma_1]\!])} \le 2^{N\cdot\exp_2^k(i^k\cdot N)\cdots\exp_2^k(i^k\cdot N)} \text{(by IH)} \\
&= 2^{N\cdot\exp_2^k(i^k\cdot N)^n} \le 2^{\exp_2^k(i^k\cdot N\cdot n+N)} \text{(by induction on } k\text{)} \\
&= \exp_2^{k+1}(n\cdot i^k\cdot N+N) \le \exp_2^{k+1}(i\cdot i^k\cdot N) = \exp_2^{k+1}(i^{k+1}\cdot N) \\
&\quad \text{(because } n\cdot i^k+1 \le (n+1)\cdot i^k \le i\cdot i^k\text{)}
\end{aligned}$$

(End of proof of claim.)

Since, in a kth-order AFS, all types occurring in type declarations have order at most $k-1$, there is some i (depending solely on $\mathcal{F}$) such that all sets $[\![\sigma]\!]$ in the algorithm have cardinality $\le \exp_2^k(i^{k-1}\cdot N)$. Writing a for the maximal arity in $\mathcal{F}$, there are at most $|\mathcal{D}| \cdot \exp_2^k(i^{k-1}\cdot N)^a \cdot N \le |\mathcal{D}| \cdot \exp_2^k((i^{k-1}\cdot a+1)\cdot N)$ distinct statements $f(\vec{A}) \approx t$.

Writing $m := i^{k-1}\cdot a + 1$ and $X := |\mathcal{D}| \cdot \exp_2^k(m\cdot N)$, we thus find: the algorithm has at most $I \le X+2$ steps, and in each step we consider at most X statements φ where $\mathsf{Confirmed}^i[\varphi] = \bot$. For every applicable rule, there are at most $(2^N)^a$ different substitutions γ, so we have to test a statement $t \in \mathcal{NF}^i(r\gamma,\eta)$ at most $X\cdot(X+2)\cdot|\mathcal{R}|\cdot 2^{aN}$ times. The exact cost of calculating $\mathcal{NF}^i(r\gamma,\eta)$ is implementation-specific, but is certainly bounded by some polynomial $P(X)$ (which depends on the form of r). This leaves the total time cost of the algorithm at $O(X\cdot(X+1)\cdot 2^{aN}\cdot P(X)) = O(P'(\exp_2^k(m\cdot N)))$ for some polynomial P' and constant m. As $\mathrm{E}^k\mathrm{TIME}$ is robust under taking polynomials, the result follows. ◀

▶ **Theorem 24.** *Let $k \ge 1$. A set $S \subseteq \{0,1\}^+$ is in $\mathrm{E}^k\mathrm{TIME}$ iff there is an AFS of order k that accepts S.*

Proof. If $S \in \mathrm{E}^k\mathrm{TIME}$, Theorem 19 shows that it is accepted by an AFS of order k. Conversely, if there is an AFS of order k that accepts S, Theorem 23 shows that we can find whether any basic term reduces to `true` in time $O(\exp_2^k(m\cdot n))$ for some m, and thus $S \in \mathrm{E}^k\mathrm{TIME}$. ◀

▶ Remark. Observe that Theorem 24 concerns *extensional* rather than *intensional* behavior of cons-free AFSs: a cons-free AFS may take arbitrarily many steps to reduce its input to normal form, even if it accepts a set that a Turing machine may decide in a bounded number of steps. However, Algorithm 20 can often find the possible results of an AFS faster than evaluating the AFS would take, by avoiding duplicate calculations.

$$
\begin{array}{lllllll}
\bot{::}t & \to & t & \mathtt{rnd} & \to & \mathtt{I} \\
\mathtt{rnd} & \to & \mathtt{0} & \mathtt{rnd} & \to & \mathtt{B}
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{translate}(\mathtt{0}(xs)) & \to & \mathtt{0} :: \mathtt{translate}(xs) \\
\mathtt{translate}(\mathtt{1}(xs)) & \to & \mathtt{I} :: \mathtt{translate}(xs) \\
\mathtt{translate}(\rhd) & \to & \mathtt{B} :: \mathtt{translate}(\rhd) \\
\mathtt{translate}(\rhd) & \to & \rhd \\
\mathtt{equal}(xl, xl) & \to & \mathtt{true}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{rndtape}(x) \to \rhd \\
\mathtt{rndtape}(x) \to \mathtt{rnd} :: \mathtt{rndtape}(x)
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{start}(cs) & \to & \mathtt{run}(\mathtt{startstate}, \rhd, \mathtt{B}, \mathtt{translate}(cs)) \\
\mathtt{run}(\underline{\mathtt{s}}, xl, \underline{\mathtt{r}}, yl) & \to & \mathtt{shift}(\underline{\mathtt{t}}, xl, \underline{\mathtt{w}}, yl, \underline{\mathtt{d}}) \quad [\![\text{for every transition } \underline{\mathtt{s}} \xRightarrow{\underline{\mathtt{r}}/\underline{\mathtt{w}}\ \underline{\mathtt{d}}} \underline{\mathtt{t}}]\!] \\
\mathtt{shift}(s, xl, c, yl, d) & \to & \mathtt{shift}_1(s, xl, c, yl, d, \mathtt{rnd}, \mathtt{rndtape}(\mathtt{0}), \mathtt{rndtape}(\mathtt{I})) \\
\mathtt{shift}_1(s, xl, c, yl, d, \underline{\mathtt{b}}, t, t) & \to & \mathtt{shift}_2(s, xl, c, yl, d, \underline{\mathtt{b}}, t) \quad [\![\text{for every } \underline{\mathtt{b}} \in \{\mathtt{0}, \mathtt{I}, \mathtt{B}\}]\!] \\
\mathtt{shift}_2(s, xl, c, yl, \mathtt{R}, z, t) & \to & \mathtt{shift}_3(s, c :: xl, z, t, \mathtt{equal}(yl, z :: t)) \\
\mathtt{shift}_2(s, xl, c, yl, \mathtt{L}, z, t) & \to & \mathtt{shift}_3(s, t, z, c :: yl, \mathtt{equal}(xl, z :: t)) \\
\mathtt{shift}_3(s, xl, c, yl, \mathtt{true}) & \to & \mathtt{run}(s, xl, c, yl)
\end{array}
$$

Figure 3 A first-order non-left-linear AFS that simulates a Turing machine.

6 Changing the restrictions

In the presence of non-determinism, minor syntactical changes can make a large difference in expressivity. We briefly consider two natural changes here.

6.1 Non-left-linearity

Recall that we imposed three restrictions: the rules in $\mathcal{R}$ must be *constructor rules*, *left-linear* and *cons-free*. Dramatically, dropping the restriction on left-linearity allows us to decide every Turing-decidable set using first-order systems. This is demonstrated by the first-order AFS in Figure 3 which simulates an arbitrary Turing Machine on input alphabet $I = \{0, 1\}$. Here, a tape $x_0 \dots x_{n \sqcup \sqcup} \dots$ with the tape head at position i is represented by a triple $(x_{i-1}{::}\cdots{::}x_0,\ x_i,\ x_{i+1}{::}\cdots{::}x_n)$, where the "list constructor" :: is a *defined symbol*, ensured by a rule which never fires. To split such a list into a head and tail, the AFS non-deterministically generates a *new* head and tail, makes sure they are fully evaluated, and uses a non-left-linear rule to test whether their combination corresponds to the original list.

6.2 Product Types

Unlike AFSs, Jones' minimal language in [14] employs a *pairing constructor*, essentially admitting terms $(s, t) : \iota \times \kappa$ if $\vdash s : \iota$ and $\vdash t : \kappa$ are data terms or themselves pairs. This is not in conflict with the cons-freeness requirement due to type restrictions: it does not allow the construction of an arbitrarily large structure of fixed type. In our (non-deterministic) setting, however, pairing is significantly more powerful. Following the ideas of Section 4, one can count up to arbitrarily large numbers: for an input string $x_n(\dots(x_1(\rhd)))$ of length n,

- the counting module C_0 represents $i \in \{0, \dots, n\}$ by a substring $x_i(\dots(x_1(\rhd))) : \mathtt{string}$;
- given a $(\lambda n.\exp_2^k(n+1))$-counting module C_k, we let C_{k+1} represent a number b with bit representation $b_0 \dots b_N$ (for $N < \exp_2^k(n+1)$) as the pair (s, t) – a term! – where s reduces to representations of those bits set to 1, and t to representations of bits set to 0.

Then for instance a number in $\{0, \dots, 2^{2^{n+1}} - 1\}$ is represented by a pair $(s, t) : (\mathtt{string} \times \mathtt{string}) \times (\mathtt{string} \times \mathtt{string})$, where s and t themselves are *not* pairs; rather, they are both terms reducing to a variety of different pairs. A membership test would take the form

$$\texttt{elem}_2(k,(s,t)) \to \texttt{elemtest}(\texttt{equal}_1(k,s),\texttt{equal}_1(k,t))$$
$$\texttt{elemtest}(\texttt{true},x) \to \texttt{true} \qquad\qquad \texttt{elemtest}(x,\texttt{true}) \to \texttt{false}$$

with the rule for $\texttt{equal}_1$ having the form $\texttt{equal}_1((s_1,t_1),(s_2,t_2)) \to r$. That is, the rule *forces a partial evaluation*. This is possible because a "false constructor" (i.e., a syntactic structure that rules can match) is allowed to occur above non-data terms.

7 Future work

In this paper, we have considered the expressive power of cons-free term rewriting, and seen that restricting data order results in characterizations of different classes. A natural direction for future work is to consider further restrictions, either on rule formation, reduction strategy, or both. Following Jones [14], we suspect that restricting to innermost evaluation will give the hierarchy P $\subseteq$ EXPTIME $\subseteq$ EXP^{2}TIME $\subsetneq \cdots$. Furthermore, we conjecture that a combination of higher-order rewriting and restrictions on rule formation, possibly together with additions such as product types, may yield characterizations of a wide range of classes, including non-deterministic classes like NP or very small classes like LOGTIME.

References

1 M. Avanzini, N. Eguchi, and G. Moser. A new order-theoretic characterisation of the polytime computable functions. In *APLAS*, volume 7705 of *LNCS*, pages 280–295, 2012. doi:10.1007/978-3-642-35182-2_20.

2 M. Avanzini and G. Moser. Closing the gap between runtime complexity and polytime computability. In *RTA*, volume 6 of *LIPIcs*, pages 33–48, 2010. doi:10.4230/LIPIcs.RTA.2010.33.

3 M. Avanzini and G. Moser. Polynomial path orders. *LMCS*, 9(4), 2013. doi:10.2168/LMCS-9(4:9)2013.

4 P. Baillot. From proof-nets to linear logic type systems for polynomial time computing. In *TLCA*, volume 4583 of *LNCS*, pages 2–7, 2007. doi:10.1007/978-3-540-73228-0_2.

5 P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP*, volume 6012 of *LNCS*, pages 104–124, 2010. doi:10.1007/978-3-642-11957-6_7.

6 P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *CSL*, volume 16 of *LIPIcs*, pages 62–76, 2012. doi:10.4230/LIPIcs.CSL.2012.62.

7 S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992. doi:10.1007/BF01201998.

8 S. Bellantoni, K. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1–3):17–30, 2000. doi:10.1016/S0168-0072(00)00006-3.

9 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *CSL*, volume 5213 of *LNCS*, pages 1–14, 2008.

10 G. Bonfante. Some programming languages for logspace and ptime. In *AMAST*, volume 4019 of *LNCS*, pages 66–80, 2006. doi:10.1007/11784180_8.

11 D. de Carvalho and J. Simonsen. An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In *RTA-TLCA*, volume 8560 of *LNCS*, pages 179–193, 2014.

12 M. Hofmann. Type systems for polynomial-time computation, 1999. Habilitationsschrift.

13 N. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, 1997.

14 N. Jones. The expressive power of higher-order types or, life without CONS. *JFP*, 11(1):55–94, 2001.

15 J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *LICS*, pages 402–411, 1999.

16 C. Kop and J. Simonsen. Complexity hierarchies and higher-order cons-free rewriting (extended version). Technical report, University of Copenhagen, 2016. Available online at the authors' homepages.

17 L. Kristiansen and K. Niggl. On the computational complexity of imperative programming languages. *TCS*, 318(1–2):139–161, 2004. `doi:10.1016/j.tcs.2003.10.016`.

18 R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *TCS*, 192(1):3–29, 1998.

19 C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

20 M. Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, 2006.

21 F. van Raamsdonk. Higher-order rewriting. In *Term Rewriting Systems*, Chapter 11. Cambridge University Press, 2003.