COMPLEXITY HIERARCHIES AND HIGHER-ORDER CONS-FREE TERM REWRITING

CYNTHIA KOP AND JAKOB GRUE SIMONSEN

Department of Computer Science, Copenhagen University
e-mail address: {kop,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 the left-hand sides; the computational intuition is that rules cannot build new data structures. In programming language research, cons-free languages have been used to characterize hierarchies of computational complexity classes; in term rewriting, cons-free first-order TRSs have been used to characterize P.

We investigate cons-free higher-order term rewriting systems, the complexity classes they characterize, and how these depend on the type order of 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 unrestricted 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 TRSs with no assumptions on reduction strategy, (ii) we consequently obtain much larger classes for each type order ($\mathrm{E}^K\mathrm{TIME}$ versus $\mathrm{EXP}^{K-1}\mathrm{TIME}$), and (iii) results for cons-free term rewriting systems have previously only been obtained for $K = 1$, and with additional syntactic restrictions besides cons-freeness and left-linearity.

Our results are 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 overlapping rules) does not directly allow for characterization of non-deterministic complexity classes like NE. We also show that non-determinism makes the classes characterized highly sensitive to minor syntactic changes like admitting product types or non-left-linear rules.

1. Introduction

In [15], 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 *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 programs can only decide the sets in a proper subclass of the Turing-decidable sets; indeed are said to *characterize* the subclass. Jones shows that adding further restrictions such as type order or enforcing tail recursion lowers the resulting

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".

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, 5, 6, 7, 8, 13, 18]).

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 we consider *higher-order* term rewriting? 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. Our results are comparable to work in descriptive complexity theory (roughly, the study of logics characterizing complexity classes) where the *non*-deterministic classes $\text{NEXP}^{K-1}\text{TIME}$ in the exponential hierarchy are exactly the sets axiomatizable by Σ_K formulas in appropriate query logics [19, 12].

2. Preliminaries

2.1. **Computational Complexity.** We presuppose introductory working knowledge of computability and complexity theory (see, e.g., [14]). Notation is fixed below.

Turing Machines (TMs) are tuples (I, A, S, T) where $I \supseteq \{0, 1\}$ is a finite set of *initial symbols*; $A \supseteq I \cup \{\llcorner\}$ is a finite set of *tape symbols* with $\llcorner \notin I$ 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 also write this transition as $i \overset{r/w\ d}{\Longrightarrow} j$. All machines in this paper are *deterministic*: every pair (i, r) with $i \in S \setminus \{\texttt{accept}, \texttt{reject}\}$ is associated with exactly one transition (i, r, w, d, j). Every Turing Machine in this paper has a single, right-infinite tape.

A *valid tape* is a right-infinite sequence of tape symbols with only finitely many not $\llcorner$. 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.

Definition 1. Let $I \supseteq \{\texttt{0}, \texttt{1}\}$ be a set of *symbols*. A *decision problem* is a set $X \subseteq I^+$.

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 $x \in I^n$, we have $(\llcorner x \lrcorner \lrcorner \ldots, 0, \texttt{start}) \Rightarrow^{\leq f(n)} (t, i, \underline{\texttt{s}})$ for some $\underline{\texttt{s}} \in \{\texttt{accept}, \texttt{reject}\}$, where $\Rightarrow^{\leq f(n)}$ denotes a sequence of at most $f(n)$ transitions.

We categorize decision problems into classes based on the time needed to decide them.

Definition 2. Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be a function. Then, $\mathrm{TIME}\,(f(n))$ is the set of all $S \subseteq I^+$ 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 $\mathcal{O}(f(n))$). Note that by design, $\mathrm{TIME}\,(\cdot)$ is closed under $\mathcal{O}$.

Definition 3. For $K, n \geq 0$, let $\exp_2^0(n) = n$ and $\exp_2^{K+1}(n) = 2^{\exp_2^K(n)} = \exp_2^K(2^n)$.

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., [20, Ch. 20]). Note also 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 $\mathrm{TIME}\left(P(\exp_2^K(an))\right)$ for some $a \in \mathbb{N}$, then $S \in \mathrm{E}^K\mathrm{TIME}$.

By the Time Hierarchy Theorem [21], $\mathrm{E} = \mathrm{E}^1\mathrm{TIME} \subsetneq \mathrm{E}^2\mathrm{TIME} \subsetneq \mathrm{E}^3\mathrm{TIME} \subsetneq \cdots$. The union $\bigcup_{K \in \mathbb{N}} \mathrm{E}^K\mathrm{TIME}$ is the set ELEMENTARY of elementary-time computable languages.

We will also sometimes refer to $\mathrm{EXP}^K\mathrm{TIME} \triangleq \bigcup_{a,b \in \mathbb{N}} \mathrm{TIME}\left(\exp_2^K(an^b)\right)$.

2.2. **Applicative term rewriting systems.** 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 [22]. For the present paper, we have chosen to employ *applicative TRSs with simple types*, as (a) the applicative style and absence of explicitly bound variables allows us to present our examples—in particular the "counting modules" of § 4—in the most intuitive way, and (b) this particular variant of higher-order rewriting is syntactically similar to Jones' original definition using functional programming. However, our proofs do not use any features of ATRS that preclude using different formalisms; for a presentation using simply-typed rewriting with explicit binders, we refer to the conference version of this paper [16].

Definition 4 (Simple types)**.** We assume given a non-empty set $\mathcal{S}$ of *sorts*. Every $\iota \in \mathcal{S}$ is a type of order *0*. 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 $\sigma \Rightarrow (\tau \Rightarrow \pi)$.

We additionally assume given disjoint sets $\mathcal{F}$ of *function symbols* and $\mathcal{V}$ of variables, each equipped with a type. This typing imposes a restriction on the formation of *terms*:

Definition 5 (Terms)**.** The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of terms over $\mathcal{F}$ and $\mathcal{V}$ consists of those expressions s such that $s : \sigma$ can be derived for some type σ using the following clauses: (a) $a : \sigma$ for $(a : \sigma) \in \mathcal{F} \cup \mathcal{V}$, and (b) $s\ t : \tau$ if $s : \sigma \Rightarrow \tau$ and $t : \sigma$.

Clearly, each term has a *unique* type. A term *has base type* if its type is in $\mathcal{S}$, and *has functional type* otherwise. We denote $Var(s)$ for the set of variables occurring in a term s and say s is *ground* if $Var(s) = \emptyset$. Application is left-associative, so every term may be denoted $a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$. We call a the *head* of this term. We will sometimes employ vector notation, denoting $a\ s_1 \cdots s_n$ simply as $a\ \vec{s}$ when no confusion can arise.

Example 6. We will often use extensions of the signature $\mathcal{F}_{\mathtt{list}}$, given by:

$$\mathtt{0} : \mathtt{symb} \qquad \mathtt{1} : \mathtt{symb} \qquad [] : \mathtt{list} \qquad ; : \mathtt{symb} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}$$

Terms are for instance $\mathtt{1} : \mathtt{symb}$ and $; \ \mathtt{0} \ (; \ \mathtt{1} \ []) : \mathtt{list}$, as well as $(; \ \mathtt{0}) : \mathtt{list} \Rightarrow \mathtt{list}$. However, we will always denote ; in a right-associative infix way and only use it fully applied; thus, the second of these terms will be denoted $\mathtt{0};\mathtt{1};[]$ and the third will not occur. Later extensions of the signature will often use additional constants of type $\mathtt{symb}$.

The notion of substitution from first-order rewriting extends in the obvious way to applicative rewriting, but we must take special care when defining subterms.

Definition 7 (Substitution, subterms and contexts)**.** A *substitution* is a type-preserving map from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ that is the identity on all but finitely many variables. Substitutions γ are extended to arbitrary terms s, notation $s\gamma$, by replacing each variable x by $\gamma(x)$. The *domain* of a substitution γ is the set consisting of those variables x such that $\gamma(x) \neq x$.

We say t is a subterm of s, notation $s \trianglerighteq t$, if (a) $s = t$, or (b) $s \rhd t$, where $s_1 \ s_2 \rhd t$ if $s_1 \rhd t$ or $s_2 \trianglerighteq t$. In case (b), we say t is a *strict* subterm of s.

Note that s_1 is *not* considered a subterm of $s_1 \ s_2$; thus, in a term $\mathtt{f} \ x_1 \cdots x_n$ the only strict subterms are $x_1, \ldots, x_n$; the term $\mathtt{f} \ x_1 \cdots x_{n-1}$ (for instance) is *not* a subterm. The reason for this arguably unusual definition is that the restrictions on rules we will employ do not allow us to ever isolate the head of an application. Therefore, such "subterms" would not be used, and are moreover problematic to consider due to their higher type order.

Example 8. Let $\mathtt{succ} : \mathtt{list} \Rightarrow \mathtt{list}$ be added to $\mathcal{F}_{\mathtt{bits}}$ of Example 6. Then $\mathtt{succ} \ (\mathtt{0};\mathtt{1};[]) \rhd \mathtt{1};[]$, but not $\mathtt{succ} \ (\mathtt{0};\mathtt{1};[]) \rhd \mathtt{succ}$. An example substitution is $\gamma := [xs := y;\mathtt{1};zs]$ (which is the identity on all variables but xs), and for $s = \mathtt{succ} \ (\mathtt{0};xs)$ we have $s\gamma = \mathtt{succ} \ (\mathtt{0};y;\mathtt{1};zs)$.

At last we are prepared to define the reduction relation.

Definition 9 (Rules and rewriting)**.** A *rule* is a pair $\ell \to r$ of terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ with the same *type* such that $Var(r) \subseteq Var(\ell)$. A rule $\ell \to r$ is *left-linear* if every variable occurs at most once in ℓ. Given a set $\mathcal{R}$ of rules, the reduction relation $\to_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is given by:

$$\begin{array}{lcll} \ell\gamma & \to_{\mathcal{R}} & r\gamma & \text{for any } \ell \to r \in \mathcal{R} \text{ and substitution } \gamma \\ s \ t & \to_{\mathcal{R}} & s' \ t & \text{if } s \to_{\mathcal{R}} s' \\ s \ t & \to_{\mathcal{R}} & s \ t' & \text{if } t \to_{\mathcal{R}} t' \end{array}$$

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 *applicative term rewriting system*, abbreviated *ATRS* is a pair $(\mathcal{F}, \mathcal{R})$ and its *type order* (or just *order*) is the maximal order of any type declaration in $\mathcal{F}$.

Example 10. Let $\mathcal{F}_{\mathtt{count}} = \mathcal{F}_{\mathtt{list}} \cup \{\mathtt{succ} : \mathtt{list} \Rightarrow \mathtt{list}\}$ be the signature from Example 8. We consider the ATRS $(\mathcal{F}_{\mathtt{count}}, \mathcal{R}_{\mathtt{count}})$ with the following rules:

$$\begin{array}{llcl} \text{(A)} & \mathtt{succ} \ [] & \to & \mathtt{1};[] \\ \text{(B)} & \mathtt{succ} \ (\mathtt{0};xs) & \to & \mathtt{1};xs \\ \text{(C)} & \mathtt{succ} \ (\mathtt{1};xs) & \to & \mathtt{0};(\mathtt{succ} \ xs) \end{array}$$

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

Example 11. We may also define counting as an operation on *functions*. We let $\mathcal{F}_{\texttt{hocount}}$ contain a number of typed symbols, including $\texttt{0}, \texttt{1} : \texttt{symb}$, $\texttt{o} : \texttt{nat}$ and $\texttt{s} : \texttt{nat} \Rightarrow \texttt{nat}$ as well as $\texttt{set} : (\texttt{nat} \Rightarrow \texttt{symb}) \Rightarrow \texttt{nat} \Rightarrow \texttt{symb} \Rightarrow \texttt{nat} \Rightarrow \texttt{symb}$. This is a second-order signature with *unary* numbers $\texttt{o}, \texttt{s}\ \texttt{o}, \texttt{s}\ (\texttt{s}\ \texttt{o}), \ldots$, which allows us to represent the bit strings from before as functions in $\texttt{nat} \Rightarrow \texttt{symb}$: a bit string $b_0 \ldots b_{n-1}$ corresponds to a function which reduces $\texttt{s}^i\ \texttt{o}$ to b_i for $0 \leq i < n$ and to $\texttt{0}$ for $i \geq n$. Let $\mathcal{R}_{\texttt{hocount}}$ consist of the rules below; types can be derived from context. The successor of a "bit string" F is given by $\texttt{fsucc}\ F\ \texttt{o}$.

(D)	$\texttt{ifeq o o}\ x\ y$	$\to$	x	(M)	$\texttt{neg}$	$\to$	$\texttt{1}$
(E)	$\texttt{ifeq (s}\ n\texttt{) o}\ x\ y$	$\to$	y	(N)	$\texttt{neg 1}$	$\to$	$\texttt{0}$
(F)	$\texttt{ifeq o (s}\ m\texttt{)}\ x\ y$	$\to$	y	(O)	$\texttt{nul}\ n$	$\to$	$\texttt{0}$
(G)	$\texttt{ifeq (s}\ n\texttt{) (s}\ m\texttt{)}\ x\ y$	$\to$	$\texttt{ifeq}\ n\ m\ x\ y$				
(H)	$\texttt{set}\ F\ n\ x\ m$	$\to$	$\texttt{ifeq}\ n\ m\ x\ (F\ m)$				
(I)	$\texttt{flip}\ F\ n$	$\to$	$\texttt{set}\ F\ n\ (\texttt{neg}\ (F\ n))$				
(J)	$\texttt{fsucc}\ F\ n$	$\to$	$\texttt{fsucchelp}\ (F\ n)\ (\texttt{flip}\ F\ n)\ n$				
(K)	$\texttt{fsucchelp 0}\ F\ n$	$\to$	F				
(L)	$\texttt{fsucchelp 1}\ F\ n$	$\to$	$\texttt{fsucc}\ F\ (\texttt{s}\ n)$				

Rules (I)–(L) have a functional type $\texttt{nat} \Rightarrow \texttt{symb}$. The function $\texttt{nul}$ represents bit strings $0 \ldots 0$, and if F represents $b_0 \ldots b_{n-1}$ then $\texttt{set}\ F\ (\texttt{s}^i\ \texttt{o})\ x$ represents $b_0 \ldots b_{i-1} x b_{i+1} \ldots b_{n-1}$. The number 5 is for instance represented by $t := \texttt{set}\ (\texttt{set nul o 1})\ (\texttt{s}^2\ \texttt{o})\ \texttt{1}$. We easily see that (**) $t\ \texttt{o} \to_{\mathcal{R}}^* \texttt{1}$ and $t\ (\texttt{s o}) \to_{\mathcal{R}}^* \texttt{0}$. Intuitively, $\texttt{fsucc}$ operates on $\texttt{1} \ldots \texttt{1} 0 b_{i+1} \ldots b_{n-1}$ by flipping bits until some $\texttt{0}$ is encountered, giving $\texttt{0} \ldots \texttt{0} 1 b_{i+1} \ldots b_{n-1}$. Using (**), $\texttt{fsucc}\ t\ \texttt{o} \to_{\mathcal{R}} \texttt{fsucchelp}\ (t\ \texttt{o})\ (\texttt{flip}\ t\ \texttt{o})\ \texttt{o} \to_{\mathcal{R}}^* \texttt{fsucchelp 1}\ (\texttt{set}\ t\ \texttt{o}\ (\texttt{neg 1}))\ \texttt{o} \to_{\mathcal{R}}^* \texttt{fsucc}\ (\texttt{set}\ t\ \texttt{o 0})\ (\texttt{s o}) \to_{\mathcal{R}}^* \texttt{fsucchelp 0}\ (\texttt{set}\ (\texttt{set}\ t\ \texttt{o 0})\ (\texttt{s o})\ \texttt{1})\ (\texttt{s o}) \to_{\mathcal{R}} \texttt{set}\ (\texttt{set}\ t\ \texttt{o 0})\ (\texttt{s o})\ \texttt{1}$; writing u for this term, we can confirm that $u\ (\texttt{s}^i\ \texttt{o}) \to_{\mathcal{R}}^* \texttt{1}$ if only if $i = 1$ or $i = 2$: u represents 6.

For the problems we will consider, a key notion is that of *data terms*.

Definition 12. We fix a partitioning of $\mathcal{F}$ into two disjoint sets, $\mathcal{D}$ of *defined symbols* and $\mathcal{C}$ of *constructor symbols*, such that $\texttt{f} \in \mathcal{D}$ for all $\texttt{f}\ \vec{\ell} \to r \in \mathcal{R}$. A term ℓ is a *pattern* if (a) ℓ is a variable, or (b) $\ell = \texttt{c}\ \ell_1 \cdots \ell_m$ with $\texttt{c} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{C}$ for $\iota \in \mathcal{S}$ and all ℓ_i patterns. A *data term* is a pattern without variables, and the set of all data terms is denoted $\mathcal{DA}$. A term $\texttt{f}\ \ell_1 \cdots \ell_n$ of base type, with $\texttt{f} \in \mathcal{D}$ and all $\ell_i \in \mathcal{DA}$ data terms is called a *basic term*. Note that all non-variable patterns—which includes all data terms—also have base type.

We will particularly consider *left-linear constructor rewriting systems*.

Definition 13. A *constructor rewriting system* is an ATRS such that all rules have the form $\texttt{f}\ \ell_1 \cdots \ell_k \to r$ with $\texttt{f} \in \mathcal{D}$ and all ℓ_i patterns. It is *left-linear* if all rules are left-linear.

Left-linear constructor rewriting systems are very common in the literature on term rewriting. The higher-order extension of *patterns* where the first-order definition merely requires constructor terms corresponds to the typical restrictions in functional programming languages, where constructors must be fully applied. However, unlike functional programming languages, we allow for overlapping rules, and do not impose an evaluation strategy.

Example 14. The ATRSs from Examples 10 and 11 are left-linear constructor rewriting systems. In Example 10, $\mathcal{C} = \mathcal{F}_{\texttt{list}}$ and $\mathcal{D} = \{\texttt{succ}\}$. If a rule $\texttt{0};[] \to []$ were added to $\mathcal{R}_{\texttt{count}}$, it would no longer be a constructor rewriting system as this would force ; to be in $\mathcal{D}$, conflicting with rules (B) and (C). A rule such as $\texttt{equal}\ n\ n \to \texttt{1}$ would break left-linearity.

2.3. **Deciding problems using rewriting.** Like Turing Machines, an ATRS can decide a set $S \subseteq I^+$ (where I is a finite set of symbols). Consider ATRSs with a signature $\mathcal{F} = \mathcal{C}_I \cup \mathcal{D}$ where $\mathcal{C}_I = \{[] : \mathtt{list},\ ; : \mathtt{symb} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}, \mathtt{true} : \mathtt{bool}, \mathtt{false} : \mathtt{bool}\} \cup \{\mathtt{a} : \mathtt{symb} \mid a \in I\}$. There is an obvious correspondence between elements of I^+ and data terms of sort `list`; if $x \in I^+$, we write $\overline{\mathtt{x}}$ for the corresponding data term.

Definition 15. An ATRS *accepts* $S \subseteq I^+$ if there is a designated defined symbol $\mathtt{decide} : \mathtt{list} \Rightarrow \mathtt{bool}$ such that, for every $x \in I^+$ we have $\mathtt{decide}\ \overline{\mathtt{x}} \to^*_{\mathcal{R}} \mathtt{true}$ iff $x \in S$. The ATRS *decides* S if moreover $\mathtt{decide}\ \overline{\mathtt{x}} \to^*_{\mathcal{R}} \mathtt{false}$ iff $x \notin S$.

While Jones considered programs *deciding* decision problems, in this paper we will consider *acceptance*—a property 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 if there is no reduction to `true` from $\mathtt{decide}\ \overline{\mathtt{x}}$. As evaluation is non-deterministic, there may be many distinct reductions starting from $\mathtt{decide}\ \overline{\mathtt{x}}$.

With an eye on future extensions in *functional* complexity—where the computational complexity of functions, rather than sets, is considered—our definitions and lemmas will more generally consider programs which reduce an arbitrary *basic term* to a *data term*. However, our main theorems consider only programs with main symbol $\mathtt{decide} : \mathtt{list} \Rightarrow \mathtt{bool}$.

3. Cons-free rewriting

As we aim to find groups of programs which can handle *restricted* classes of Turing-computable problems, we will impose certain limitations. We limit interest to the left-linear constructor TRSs from § 2.2, but impose the additional restriction that they must be *cons-free*.

Definition 16. A rule $\ell \to r$ is cons-free if for all $r \trianglerighteq s$: if s has the form $\mathtt{c}\ s_1 \cdots s_n$ with $\mathtt{c} \in \mathcal{C}$, then $s \in \mathcal{DA}$ or $\ell \rhd s$. A left-linear constructor ATRS is cons-free if all its rules are.

Definition 16 corresponds largely to the definitions of cons-freeness in [11, 15]. In a cons-free system, it is not possible to build new non-constant data, as we will see in § 3.1.

Example 17. The ATRSs from Examples 10 and 11 are not cons-free; in the first case due to rules (B) and (C), in the second due to rule (F). To some extent, we can repair the second case, however: by counting *down* rather than *up*. To be exact, we let n be a *fixed* number, assume that $\mathtt{s}^n\ \mathtt{0}$ is given as input to the ATRS, and represent a number as a finite bitstring $b_0 \dots b_{n-1}$ with the most significant digit first—in contrast to Example 11, where we used essentially infinite bitstrings $b_0 \dots b_{n-1}\mathtt{000}\dots$ with the *least* significant digit first.

We can reuse most of the previous rules, but replace the (non-cons-free) rule (L) by:

$$(\mathsf{L}.1)\quad \mathtt{fsucchelp\ 1}\ F\ \mathtt{o} \ \to\ F \qquad (\mathsf{L}.2)\quad \mathtt{fsucchelp\ 1}\ F\ (\mathtt{s}\ n) \ \to\ \mathtt{fsucc}\ F\ n$$

Now a function F represents $b_0 \dots b_{n-1}$ if F reduces $\mathtt{s}^i\ \mathtt{o}$ to b_i for $0 \le i < n$; since we only consider n bits, F may reduce to anything given data not of this form. Then $\mathtt{fsucc}\ F\ (\mathtt{s}^n\ \mathtt{o})$ reduces to a function representing the successor of F, modulo 2^n ($1 \dots 1$ is reduced to $0 \dots 0$).

Remark 18. The limitation to left-linear constructor systems is standard, but also *necessary*: if either restriction is dropped, our limitation to cons-free systems becomes meaningless, and we retain a Turing-complete language. This will be discussed in detail in § 7.2.

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

3.1. **Properties of Cons-free Term Rewriting.** As mentioned, cons-free term rewriting cannot create new non-constant data terms. 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 19. Let $(\mathcal{F}, \mathcal{R})$ be a fixed constructor ATRS. 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 20 ($\mathcal{B}$-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 $\mathsf{c}\ t_1 \cdots t_m$ with $\mathsf{c} \in \mathcal{C}$, then $t \in \mathcal{B}$.

Lemma 21. *If s is $\mathcal{B}$-safe and $s \to_{\mathcal{R}} t$, then t is $\mathcal{B}$-safe.*

Proof. By induction on the form of s; the result follows trivially by the induction hypothesis if the reduction does not take place at the head of s, leaving only the base case $s = \mathsf{f}\ (\ell_1\gamma) \cdots (\ell_k\gamma)\ s_1 \cdots s_n \to_{\mathcal{R}} r\gamma\ s_1 \cdots s_n = t$ for some rule $\mathsf{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R}$, substitution γ and $n \geq 0$. All subterms u of t are (a) subterms of some s_i, (b) subterms of $r\gamma$ or (c) the term t itself, so suppose $u = \mathsf{c}\ t_1 \cdots t_m$ with $\mathsf{c} \in \mathcal{C}$ and consider the three possible situations.

In case (a), $u \in \mathcal{B}$ by $\mathcal{B}$-safety of s.

In case (b), either $\gamma(x) \trianglerighteq u$ for some x, or $u = r'\gamma$ for some $r \trianglerighteq r' \notin \mathcal{V}$. In the first case, $x \in \mathit{Var}(\ell_i)$ for some i and—since ℓ_i is a pattern—a trivial induction on the form of ℓ_i shows that $\ell\gamma \trianglerighteq \gamma(x) \trianglerighteq u$, so again $u \in \mathcal{B}$ by $\mathcal{B}$-safety of $s = \ell\gamma$. In the second case, if $r' = x\ r_1 \cdots r_{n'}$ with $x \in \mathcal{V}$ and $n > 0$ then $s \trianglerighteq \gamma(x)$ as before, so $\gamma(x) \in \mathcal{DA}$ (because $\gamma(x)$ must have a constructor as its head), which imposes $n = 0$; contradiction. Otherwise $r' = \mathsf{c}\ r_1 \cdots r_n$, so by definition of cons-freeness, either $u = r' \in \mathcal{B}$ or $s \rhd \ell_i\gamma \trianglerighteq r'\gamma = u$.

In case (c), $n = 0$ because, following the analysis above, $r\gamma \in \mathcal{B}$. □

Thus, if we start with a basic term $\mathsf{f}\ s_1 \cdots s_n$, any data terms occurring in a reduction $\mathsf{f}\ \vec{s} \to^*_{\mathcal{R}} t$ (directly or as subterms) are in $\mathcal{B}_{\mathsf{f}\ \vec{s}}$. This insight will be instrumental in § 5.

Example 22. By Lemma 21, functions in a cons-free ATRS cannot build recursive data. Therefore it is often necessary to "code around" a problem. Consider the task of finding the most common bit in a given bit string. A typical solution employs a rule like $\mathtt{majority}\ cs \to \mathtt{cmp}\ (\mathtt{count0}\ cs)\ (\mathtt{count1}\ cs)$. Now, however, we cannot define $\mathtt{count}$ functions which may return arbitrary terms of the form $\mathtt{s}^i\ \mathtt{o}$. Instead we use subterms of the input as a measure of size, representing a number i by a list of length i.

$$
\begin{array}{rcl}
\mathtt{majority}\ cs & \to & \mathtt{count}\ cs\ cs\ cs \\
\mathtt{count}\ (\mathtt{0};xs)\ ys\ (b;zs) & \to & \mathtt{count}\ xs\ ys\ zs \\
\mathtt{count}\ (\mathtt{1};xs)\ (b;ys)\ zs & \to & \mathtt{count}\ xs\ ys\ zs \\
\mathtt{count}\ []\ ys\ zs & \to & \mathtt{cmp}\ ys\ zs
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{cmp}\ []\ zs & \to & \mathtt{1} \\
\mathtt{cmp}\ (y;ys)\ [] & \to & \mathtt{0} \\
\mathtt{cmp}\ (y;ys)\ (z;zs) & \to & \mathtt{cmp}\ ys\ zs
\end{array}
$$

(The signature extends $\mathcal{F}_{\mathtt{list}}$, 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 23. *Given a cons-free ATRS $(\mathcal{F},\mathcal{R})$ with $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$, let $Y = \{\mathtt{c} : \sigma \in \mathcal{C} \mid \mathit{order}(\sigma) > 1\}$. 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 terms 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 $t \in \mathcal{T}(\mathcal{F}',\emptyset)$ 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). Thus $\mathcal{B}$-safe terms can only be matched by rules in $\mathcal{R}'$, so Lemma 21 gives (b). □

3.2. **A larger example.** So far, all our examples have been deterministic. To show the possibilities, we consider a first-order cons-free ATRS that solves the Boolean satisfiability problem (SAT). This is striking because, in Jones' language in [15], first-order programs cannot do this 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 strategy.

Given n boolean variables $x_1,\dots,x_n$ and a boolean formula $\psi ::= \varphi_1 \wedge \dots \wedge \varphi_m$, 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 decision problem as a string over $I := \{\mathtt{0},\mathtt{1},\#,?\}$: the formula ψ is represented by $E ::= b_{1,1}\dots b_{1,n}\#b_{2,1}\cdots_{2,n}\#\dots\#b_{m,1}\dots b_{m,n}\#$, where for each i,j: $b_{i,j}$ is `1` if x_j is a literal in φ_i, $b_{i,j}$ is `0` if $\neg x_j$ is a literal in φ_i, and $b_{i,j}$ is ? otherwise.

Example 24. The satisfiability problem for $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$ is encoded as $E :=$ 10?#?10#. Encoding this string as a data term, we obtain $\overline{\mathtt{E}} = \mathtt{1;0;?;\#;?;1;0;\#;[]}$.

Defining $\mathcal{C}_I$ as done in § 2.3 and assuming other declarations clear from context, we claim that the system in Figure 1 can reduce `decide` $\overline{\mathtt{E}}$ to `true` if and only if ψ is satisfiable.

`//` Rules using $\underline{\mathtt{a}},\underline{\mathtt{b}}$ stand for several rules once: $\underline{\mathtt{a}},\underline{\mathtt{b}}$ range over $\{\mathtt{0},\mathtt{1},?\}$ (but not #).

$$
\begin{array}{rcl@{\qquad}rcl}
\mathtt{equal}\ (\#;xs)\ (\#;ys) & \to & \mathtt{true} & \mathtt{equal}\ (\#;xs)\ (\underline{\mathtt{a}};ys) & \to & \mathtt{false} \\
\mathtt{equal}\ []\ ys & \to & \mathtt{false} & \mathtt{equal}\ (\underline{\mathtt{a}};xs)\ (\#;ys) & \to & \mathtt{false} \\
 & & & \mathtt{equal}\ (\underline{\mathtt{a}};xs)\ (\underline{\mathtt{b}};ys) & \to & \mathtt{equal}\ xs\ ys \\
\mathtt{either}\ xs\ yss & \to & xs & \mathtt{skip}\ (\#;xs) & \to & xs \\
\mathtt{either}\ xs\ yss & \to & yss & \mathtt{skip}\ (\underline{\mathtt{a}};xs) & \to & \mathtt{skip}\ xs
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{decide}\ cs & \to & \mathtt{assign}\ cs\ []\ []\ cs \\
\mathtt{assign}\ (\#;xs)\ yss\ zss\ cs & \to & \mathtt{main}\ yss\ zss\ cs \\
\mathtt{assign}\ (\underline{\mathtt{a}};xs)\ yss\ zss\ cs & \to & \mathtt{assign}\ xs\ (\mathtt{either}\ xs\ yss)\ zss\ cs \\
\mathtt{assign}\ (\underline{\mathtt{a}};xs)\ yss\ zss\ cs & \to & \mathtt{assign}\ xs\ yss\ (\mathtt{either}\ xs\ zss)\ cs \\
\mathtt{main}\ yss\ zss\ (?;xs) & \to & \mathtt{main}\ yss\ zss\ xs \\
\mathtt{main}\ yss\ zss\ (\mathtt{0};xs) & \to & \mathtt{membtest}\ yss\ zss\ xs\ (\mathtt{equal}\ zss\ xs)\ (\mathtt{equal}\ yss\ xs) \\
\mathtt{main}\ yss\ zss\ (\mathtt{1};xs) & \to & \mathtt{membtest}\ yss\ zss\ xs\ (\mathtt{equal}\ yss\ xs)\ (\mathtt{equal}\ zss\ xs) \\
\mathtt{main}\ yss\ zss\ (\#;xs) & \to & \mathtt{false} \\
\mathtt{main}\ yss\ zss\ [] & \to & \mathtt{true} \\
\mathtt{membtest}\ yss\ zss\ xs\ \mathtt{true}\ b & \to & \mathtt{main}\ yss\ zss\ (\mathtt{skip}\ xs) \\
\mathtt{membtest}\ yss\ zss\ xs\ b\ \mathtt{true} & \to & \mathtt{main}\ yss\ zss\ xs
\end{array}
$$

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

In this system, we follow some of the same ideas as in Example 22. In particular, any list of the form $b_{i+1};\dots;b_n;\#\dots$ with each $b_j \in \{\mathtt{0},\mathtt{1},?\}$ is considered to represent the number i (with $\#;\dots$ representing n). The rules for `equal` are defined so that `equal` s t tests equality

of these *numbers*, not the full lists. The key idea new to this example is that we use terms not in normal form to represent a *set* of numbers. Fixing n, a set $X \subseteq \{1,\dots,n\}$ is encoded as a pair (yss, zss) of terms such that, for $i \in \{1,\dots,n\}$: $yss \to^*_{\mathcal{R}} xs$ for a representation xs of i if and only if $i \in X$, and $zss \to^*_{\mathcal{R}} xs$ for a representation xs of i if and only if $i \notin X$.

These pairs (yss,zss) are constructed using the symbol `either`, which is defined by a pair of overlapping rules: `either` s_1 (`either` s_2 (… (`either` s_{n-1} s_n)…)) reduces to each s_i. We can use such terms as we do—copying and passing them around without reducing to normal form—because we do not use call-by-value or similar strategies: the ATRS may be evaluated using, e.g., outermost reduction. While we *can* use other strategies, any evaluation which reduces yss or zss too eagerly just ends in an irreducible, non-data state.

Now, an evaluation starting in `decide` $\overline{\mathtt{E}}$ first non-deterministically constructs a "set" X—represented as (yss, zss)—containing those boolean variables assigned `true`: `decide` $\overline{\mathtt{E}} \to^*_{\mathcal{R}}$ `main` yss zss $\overline{\mathtt{E}}$. Then, the main function goes through $\overline{\mathtt{E}}$, finding for each clause a literal that is satisfied by the assignment. Encountering $b_{i,j} \neq ?$, we determine if $j \in X$ by comparing both a reduct of yss and of zss to j. If $yss \to^*_{\mathcal{R}}$ "j" then $j \in X$, if $zss \to^*_{\mathcal{R}}$ "j" then $j \notin X$; in either case, we continue accordingly. If the evaluation state is incorrect, or if yss or zss are both reduced to some other term, the evaluation gets stuck in a non-data normal form.

Note: variable namings are indicative of their use: in an evaluation starting in `decide` $\overline{\mathtt{E}}$, the variables xs and ys are always instantiated by data term lists, and cs by $\overline{\mathtt{E}}$; variables yss and zss are instantiated by terms of type list which do not need to be in normal form.

Example 25. To determine satisfiability of $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$, we reduce `decide` $\overline{\mathtt{E}}$, where $E = 10?\#?10\#$. First, we build a valuation. The `assign` rules are non-deterministic, but a possible reduction is `decide` $\overline{\mathtt{E}} \to^*_{\mathcal{R}}$ `main` s t $\overline{\mathtt{E}}$, where $s =$ `either` $\overline{0?\#?10\#}$ [] and $t =$ `either` $\overline{\#?10\#}$ (`either` $\overline{?\#?10\#}$ []). Since $n = 3$, $\overline{0?\#?10\#}$ represents 1 while $\overline{\#?10\#}$ and $\overline{?\#?10\#}$ represent 3 and 2 respectively. Thus, we have $[x_1 := \top, x_2 := \bot, x_3 := \bot]$.

Then the main loop recurses over the problem. Since s reduces to a term $\overline{0?\#\dots}$ and t to both $\overline{\#\dots}$ and $\overline{?\#\dots}$ we have `main` s t $\overline{\mathtt{E}} =$ `main` s t $\overline{10?\#?10\#} \to^*_{\mathcal{R}}$ `main` s t (`skip` $\overline{10\#?10\#}$) $\to^*_{\mathcal{R}}$ `main` s t $\overline{?10\#}$: the first clause is confirmed since $x_1 := \top$, so it is removed and the loop continues with the second clause. Next, the loop passes over those variables whose assignment does not contribute to the clause, until the clause is confirmed due to x_3: `main` s t $\overline{?01\#} \to_{\mathcal{R}}$ `main` s t $\overline{01\#} \to^*_{\mathcal{R}}$ `main` s t $\overline{1\#} \to^*_{\mathcal{R}}$ `main` s t (`skip` $\overline{\#}$) $\to_{\mathcal{R}}$ `main` s t [] $\to_{\mathcal{R}}$ `true`.

Due to non-determinism, the term in Example 25 could also have been reduced to `false`, by selecting a different valuation. This is not problematic: by definition, the ATRS accepts the set of satisfiable formulas if: `decide` $\overline{\mathtt{E}} \to^*_{\mathcal{R}}$ `true` if and only if E is a satisfiable formula.

4. Simulating E^{k}TIME Turing machines

We now show how to simulate Turing Machines by cons-free rewriting. For this, we use an approach very similar to that by Jones [15]. Fixing a machine (I, A, S, T), we let $\mathcal{C} := \mathcal{C}_A \cup \{\mathtt{s} : \mathtt{state} \mid s \in S\} \cup \{\mathtt{fail} : \mathtt{state},\ \mathtt{L} : \mathtt{direction},\ \mathtt{R} : \mathtt{direction},\ \mathtt{action} : \mathtt{symb} \Rightarrow \mathtt{direction} \Rightarrow \mathtt{state} \Rightarrow \mathtt{trans}\}$; we denote `B` for the symbol corresponding to $\llcorner \in A$. We will introduce defined symbols and rules such that, for any string $E = c_1 \dots c_n \in I^+$:

- `decide` $\overline{\mathtt{E}} \to^*_{\mathcal{R}}$ `true` iff $(\llcorner c_1 \dots c_n \llcorner\llcorner \dots, 0, \mathtt{start}) \Rightarrow^* (t, i, \mathtt{accept})$ for some t, i;
- `decide` $\overline{\mathtt{E}} \to^*_{\mathcal{R}}$ `false` iff $(\llcorner c_1 \dots c_n \llcorner\llcorner \dots, 0, \mathtt{start}) \Rightarrow^* (t, i, \mathtt{reject})$ for some t, i.

While `decide` $\overline{\mathtt{E}}$ may have other normal forms, only one normal form will be a data term.

4.1. **Core simulation.** The idea of the simulation is to represent non-negative integers as terms and let $\mathtt{tape}\ n\ p$ reduce to the symbol at position p on the tape at the start of the n^{th} step, while $\mathtt{state}\ n\ p$ returns the state of the machine at time n, *provided the tape reading head is at position* p. If the reading head is not at position p at time n, then $\mathtt{state}\ n\ p$ should return $\mathtt{fail}$ instead; this allows us to test the position of the reading head. As the machine is deterministic, we can devise rules to compute these terms from earlier configurations.

Finding a suitable representation of integers is the most intricate part of this simulation, where we may need higher-order functions and non-deterministic rules. Therefore, let us first assume that this can be done. Then, for a Turing machine which is known to run in time bounded above by $\lambda n.P(n)$, we define the ATRS in Figure 2 (further elaboration is given as "comments" in the ATRS). As before, the rules are constructed such that, in an evaluation of $\mathtt{decide}\ \overline{\mathtt{E}}$, the variable cs can always be assumed to be instantiated by $\overline{\mathtt{E}}$.

4.2. **Counting.** The goal, then, is to represent numbers and define rules to do four things:

- calculate $[P(|cs|)]$ or an overestimation (as the TM 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 $[p]$, calculate $[p+1]$, *provided* $p+1 \leq P(|cs|)$ as $\mathtt{transition}\ cs\ [n]\ [p] \to_{\mathcal{R}} \mathtt{NA}$ when $n < p$ and $[n]$ never increases—so it suffices to determine $[\min(p+1, P(|cs|))]$.

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 ATRSs do not admit pair types). 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. $\mathtt{pred}^1\ cs\ n_1\ n_2 \to^*_{\mathcal{R}} n_1'$ and $\mathtt{pred}^2\ cs\ n_1\ n_2 \to^*_{\mathcal{R}} n_2'$ for (n_1', n_2') some tuple representing $n-1$. Thus, for instance the last $\mathtt{get}$ rule becomes:

$$\mathtt{get}\ cs\ (x;xs)\ i_1\ i_2 \to \mathtt{ifelse}_{\mathtt{symb}}\ (\mathtt{zero}\ i_1\ i_2)\ x\ (\mathtt{get}\ xs\ (\mathtt{pred}^1\ cs\ i_1\ i_2)\ (\mathtt{pred}^2\ cs\ i_1\ i_2)$$

Following Jones [15], we use the notion of a *counting module* which provides an ATRS 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 [15].

Definition 26 (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 $C_\pi ::= (\vec{\sigma}, \Sigma, R, \mathcal{A}, \langle\cdot\rangle)$—where π is the name we use to refer to the counting module—such that:

- $\vec{\sigma}$ is a sequence of types $\sigma_1 \otimes \cdots \otimes \sigma_a$ where each σ_i has order at most $K-1$;
- Σ is a K^{th}-order signature disjoint from $\mathcal{F}$, which contains designated symbols $\mathtt{zero}_\pi : \mathtt{list} \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_a \Rightarrow \mathtt{bool}$ and, for $1 \leq i \leq a$, symbols $\mathtt{pred}^i_\pi, \mathtt{succ}^i_\pi : \mathtt{list} \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_a \Rightarrow \sigma_i$ and $\mathtt{seed}^i_\pi : \mathtt{list} \Rightarrow \sigma_i$ (and may contain others);
- R is a set of cons-free (left-linear constructor-)rules $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ with $\mathtt{f} \in \Sigma$, each $\ell_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ and $r \in \mathcal{T}(\mathcal{C} \cup \Sigma, \mathcal{V})$;
- for every string $cs \subseteq I^+$, $\mathcal{A}_{cs} \subseteq \{(s_1, \ldots, s_a) \in \mathcal{T}(\mathcal{C} \cup \Sigma)^a \mid s_j : \sigma_j \text{ for } 1 \leq j \leq a\}$;
- for every string cs, $\langle\cdot\rangle_{cs}$ is a surjective mapping from $\mathcal{A}_{cs}$ to $\{0, \ldots, P(|cs|)-1\}$;
- the following properties on $\mathcal{A}_{cs}$ and $\langle\cdot\rangle_{cs}$ are satisfied:
  - $(\mathtt{seed}^1_\pi\ cs, \ldots, \mathtt{seed}^a_\pi\ cs) \in \mathcal{A}_{cs}$ and $\langle(\mathtt{seed}^1_\pi\ cs, \ldots, \mathtt{seed}^a_\pi\ cs)\rangle_{cs} = P(|cs|) - 1$;

  and for all $(s_1, \ldots, s_a) \in \mathcal{A}_{cs}$ with $\langle(s_1, \ldots, s_a)\rangle_{cs} = m$:

– $(\mathtt{pred}^1_\pi\ cs\ \vec{s}, \ldots, \mathtt{pred}^a_\pi\ cs\ \vec{s})$ and $(\mathtt{succ}^1_\pi\ cs\ \vec{s}, \ldots, \mathtt{succ}^a_\pi\ cs\ \vec{s})$ are in $\mathcal{A}_{cs}$;
– $\langle(\mathtt{pred}^1_\pi\ cs\ \vec{s}, \ldots, \mathtt{pred}^a_\pi\ cs\ \vec{s})\rangle_{cs} = \max(m-1, 0)$;
– $\langle(\mathtt{succ}^1_\pi\ cs\ \vec{s}, \ldots, \mathtt{succ}^a_\pi\ cs\ \vec{s})\rangle_{cs} = \min(m+1, P(|cs|)-1)$;
– $\mathtt{zero}_\pi\ cs\ \vec{s} \to^*_R \mathtt{true}$ 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 $(t_1, \ldots, t_a) \in \mathcal{A}_{cs}$, then also $\langle(t_1, \ldots, t_a)\rangle_{cs} = m$.

// Determine the transition taken at time $[n]$ given input cs, *provided* the tape
// reading head is at position $[p]$ at time $[n]$; if not, reduce to `NA` instead.

$$\begin{aligned}
\mathtt{transition}\ cs\ [n]\ [p] &\to \mathtt{transitionhelp}\ (\mathtt{state}\ cs\ [n]\ [p])\ (\mathtt{tape}\ cs\ [n]\ [p])\\
\mathtt{transitionhelp}\ \mathtt{fail}\ x &\to \mathtt{NA}\\
\mathtt{transitionhelp}\ \underline{\mathtt{s}}\ \underline{\mathtt{r}} &\to \mathtt{action}\ \underline{\mathtt{w}}\ \underline{\mathtt{d}}\ \underline{\mathtt{t}} && [\![\text{for all } \underline{\mathtt{s}} \xRightarrow{\underline{\mathtt{r}}/\underline{\mathtt{w}}\ \underline{\mathtt{d}}} \underline{\mathtt{t}} \in T]\!]\\
\mathtt{transitionhelp}\ \underline{\mathtt{s}}\ x &\to \mathtt{end}\ \underline{\mathtt{s}} && [\![\text{for } \underline{\mathtt{s}} \in \{\mathtt{accept}, \mathtt{reject}\}]\!]
\end{aligned}$$

// Determine the state at time $[n]$ given input cs, *provided* the tape reading head
// is at position $[p]$ at time $[n]$ (which happens if it is at position $[p-1]$, $[p]$ or
// $[p+1]$ at time $[n-1]$ and the right action is taken); if not, reduce to `fail`.

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

$$\begin{aligned}
\mathtt{statey}\ (\mathtt{action}\ x\ \mathtt{R}\ q)\ a\ e &\to q & \qquad \mathtt{statey}\ \mathtt{NA}\ (\mathtt{action}\ x\ d\ q)\ e &\to \mathtt{fail}\\
\mathtt{statey}\ (\mathtt{action}\ x\ \mathtt{L}\ q)\ a\ e &\to \mathtt{fail} & \mathtt{statey}\ \mathtt{NA}\ \mathtt{NA}\ (\mathtt{action}\ x\ \mathtt{L}\ q) &\to q\\
\mathtt{statey}\ (\mathtt{end}\ q)\ a\ e &\to \mathtt{fail} & \mathtt{statey}\ \mathtt{NA}\ \mathtt{NA}\ (\mathtt{action}\ x\ \mathtt{R}\ q) &\to \mathtt{fail}\\
\mathtt{statey}\ \mathtt{NA}\ (\mathtt{end}\ q)\ e &\to q & \mathtt{statey}\ \mathtt{NA}\ \mathtt{NA}\ (\mathtt{end}\ q) &\to \mathtt{fail}
\end{aligned}$$

// Determine the tape symbol at position $[p]$ at time $[n]$ given input cs, which is
// `tape` cs $[n-1]$ $[p]$ unless the transition at time $[n-1]$ occurred at position $[p]$.

$$\begin{aligned}
\mathtt{tape}\ cs\ [n]\ [p] &\to \mathtt{ifelse}_{\mathtt{symb}}\ [n=0]\ (\mathtt{inputtape}\ cs\ [p])\\
&\qquad (\mathtt{tapex}\ cs\ [n-1]\ [p])\\
\mathtt{tapex}\ cs\ [n]\ [p] &\to \mathtt{tapey}\ cs\ [n]\ [p]\ (\mathtt{transition}\ cs\ [n]\ [p])\\
\mathtt{tapey}\ cs\ [n]\ [p]\ (\mathtt{action}\ x\ d\ q) &\to x\\
\mathtt{tapey}\ cs\ [n]\ [p]\ \mathtt{NA} &\to \mathtt{tape}\ cs\ [n]\ [p]\\
\mathtt{tapey}\ cs\ [n]\ [p]\ (\mathtt{end}\ q) &\to \mathtt{tape}\ cs\ [n]\ [p]\\
\mathtt{inputtape}\ cs\ [p] &\to \mathtt{ifelse}_{\mathtt{symb}}\ [p=0]\ \mathtt{B}\ (\mathtt{get}\ cs\ cs\ [p-1])\\
\mathtt{get}\ cs\ []\ [i] &\to \mathtt{B}\\
\mathtt{get}\ cs\ (x;xs)\ [i] &\to \mathtt{ifelse}_{\mathtt{symb}}\ [i=0]\ x\ (\mathtt{get}\ cs\ xs\ [i-1])
\end{aligned}$$

// We simulate the TM's outcome by testing whether the state at time $[P(|cs|)]$ is
// `accept` or `reject`, allowing for any reader head position in $\{[P(|cs|)], \ldots, [0]\}$.

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

// Rules for an if-then-else statement (which is not included by default).

$$\begin{aligned}
\mathtt{ifelse}_{\underline{\iota}}\ \mathtt{true}\ y\ z &\to y && [\![\text{for all } \underline{\iota} \in \{\mathtt{state}, \mathtt{symb}\}]\!]\\
\mathtt{ifelse}_{\underline{\iota}}\ \mathtt{false}\ y\ z &\to z && [\![\text{for all } \underline{\iota} \in \{\mathtt{state}, \mathtt{symb}\}]\!]
\end{aligned}$$

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

It is not hard to see how we would use a P-counting module in the ATRS of Figure 2; this results in a K^{th}-order system for a K^{th}-order module. Note that number representations $(s_1, \dots, s_a)$ are not required to be 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. As 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 § 3.2 for the arguments s and t of `main`.

To simplify the creation of counting modules, we start by observing that $\mathtt{succ}_\pi$ can be expressed in terms of $\mathtt{seed}_\pi$, $\mathtt{pred}_\pi$ and $\mathtt{zero}_\pi$, as demonstrated in Figure 3 (which also introduces an equality test, which will turn out to be useful in Lemma 30). In practice, $\mathtt{succ}_\pi\ cs\ [n]$ counts down from $[P(|cs|) - 1]$ to some $[m]$ with $n = m - 1$.

$$
\begin{aligned}
\mathtt{equal}_\pi\ cs\ n_1 \dots n_a\ m_1 \dots m_a \ &\to\ \mathtt{ifelse}_{\mathtt{bool}}\ (\mathtt{zero}_\pi\ cs\ \vec{n})\ (\mathtt{zero}_\pi\ cs\ \vec{m}) \\
&\qquad (\ \mathtt{ifelse}_{\mathtt{bool}}\ (\mathtt{zero}_\pi\ cs\ \vec{m})\ \mathtt{false} \\
&\qquad\quad (\ \mathtt{equal}_\pi\ cs\ (\mathtt{pred}^1_\pi\ cs\ \vec{n}) \dots (\mathtt{pred}^a_\pi\ cs\ \vec{n}) \\
&\qquad\qquad\quad (\mathtt{pred}^1_\pi\ cs\ \vec{m}) \dots (\mathtt{pred}^a_\pi\ cs\ \vec{m}) \\
&\qquad\quad)\ ) \\
\mathtt{succ}^i_\pi\ cs\ n_1 \dots n_a \ &\to\ \mathtt{succ2}^i_\pi\ cs\ n_1 \dots n_a\ (\mathtt{seed}^1\ cs) \dots (\mathtt{seed}^a\ cs) \\
\mathtt{succ2}^i_\pi\ cs\ n_1 \dots n_a\ m_1 \dots m_a \ &\to\ \mathtt{ifelse}_{\sigma_i}\ (\mathtt{zero}_\pi\ cs\ \vec{m})\ (\mathtt{seed}^i\ cs)\ (\ \mathtt{succ3}^i_\pi \\
&\qquad cs\ \vec{n}\ m_i\ (\mathtt{pred}^1_\pi\ cs\ \vec{m}) \dots (\mathtt{pred}^a_\pi\ cs\ \vec{m})\ ) \\
\mathtt{succ3}^i_\pi\ cs\ n_1 \dots n_a\ m_i\ m'_1 \dots m'_a \ &\to\ \mathtt{ifelse}_{\sigma_i}\ (\mathtt{equal}_\pi\ cs\ n_1 \dots n_a\ m'_1 \dots m'_a)\ m_i \\
&\qquad (\ \mathtt{succ2}^i_\pi\ cs\ n_1 \dots n_a\ m'_1 \dots m'_a\ )
\end{aligned}
$$

$$
\left.
\begin{aligned}
\mathtt{ifelse}_\tau\ \mathtt{true}\ y\ z \ &\to\ y \\
\mathtt{ifelse}_\tau\ \mathtt{false}\ y\ z \ &\to\ z
\end{aligned}
\right\} \ [\![\text{for } \tau \in \{\mathtt{bool}, \sigma_1, \dots, \sigma_a\}]\!]
$$

Figure 3: Expressing $\mathtt{succ}_\pi$ in terms of $\mathtt{seed}_\pi$, $\mathtt{pred}_\pi$ and $\mathtt{zero}_\pi$.

Remark 27. Observant readers may notice that the rule for $\mathtt{equal}_\pi$ is non-terminating: $\mathtt{equal}_\pi\ cs\ [0]\ [0]$ can be reduced to a term containing $\mathtt{equal}_\pi\ cs\ [0]\ [0]$ as a subterm, as the `ifelse` rules are not prioritised over other rules. Following Definition 15, this is unproblematic: it suffices if there *is* a terminating evaluation from $\mathtt{decide}\ \overline{\mathtt{x}}$ to `true` if $x \in S$; it is not necessary for *all* evaluations to terminate.

Example 28. We design a $(\lambda n.n + 1)$-counting module that represents numbers as (terms reducing to) subterms of the input list cs. Formally, we let $C_{\mathtt{lin}} := (\mathtt{list}, \Sigma, R, \mathcal{A}, \langle\cdot\rangle)$ where $\mathcal{A}_{cs} = \{s \in \mathcal{T}(\Sigma \cup \mathcal{C}) \mid s : \mathtt{list} \wedge s \text{ has a unique normal form, which is a subterm of } cs\}$ and $\langle s \rangle_{cs} =$ the number of ; operators in the normal form of s. R consists of the rules below along with the rules in in Figure 3, and Σ consists of the defined symbols in R.

$$
\begin{aligned}
\mathtt{seed}^1_{\mathtt{lin}}\ cs \ &\to\ cs \qquad
\mathtt{pred}^1_{\mathtt{lin}}\ cs\ [] \ \to\ [] \qquad
\mathtt{zero}^1_{\mathtt{lin}}\ cs\ [] \ \to\ \mathtt{true} \\
&\qquad\qquad \mathtt{pred}^1_{\mathtt{lin}}\ cs\ (x;xs) \ \to\ xs \qquad
\mathtt{zero}^1_{\mathtt{lin}}\ cs\ (x;xs) \ \to\ \mathtt{false}
\end{aligned}
$$

The counting module of Example 28 is very simple, but does not count very high: using it with Figure 2, we can simulate only machines operating in $n - 1$ steps or fewer. However, having the linear module as a basis, we can define *composite* modules to count higher:

Lemma 29. *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. Fixing cs and writing $N := P(|cs|)$ and $M := Q(|cs|)$, a number i in $\{0, \dots, N \cdot M - 1\}$ can be seen as a unique pair (n, m) with $0 \le n < N$ and $0 \le m < M$, such that $i = n \cdot M + m$.

Then `seed`, `pred` and `zero` can be expressed using the same functions on n and m. Write $C_\pi ::= (\sigma_1 \otimes \cdots \otimes \sigma_a, \Sigma^\pi, R^\pi, \mathcal{A}^\pi, \langle\cdot\rangle^\pi)$ and $C_\rho ::= (\tau_1 \otimes \cdots \otimes \tau_b, \Sigma^\rho, R^\rho, \mathcal{A}^\rho, \langle\cdot\rangle^\rho)$; we assume Σ^π and Σ^ρ are disjoint (wlog by renaming). Then numbers in $n \in \{0,\dots,N\}$ are represented in C_π by tuples $(u_1,\dots,u_a)$ of length a, and numbers in $m \in \{0,\dots,M\}$ are represented in C_ρ by tuples $(v_1,\dots,v_b)$ of length b. We will represent $n \cdot M + m$ by $(u_1,\dots,u_a,v_1,\dots,v_b)$. Formally, $C_{\pi\cdot\rho} := (\sigma_1 \otimes \cdots \otimes \sigma_a \otimes \tau_1 \otimes \cdots \otimes \tau_b, \Sigma^\pi \cup \Sigma^\rho \cup \Sigma, R^\pi \cup R^\rho \cup R, \mathcal{A}^{\pi\cdot\rho}, \langle\cdot\rangle^{\pi\cdot\rho})$, where:

- $\mathcal{A}^{\pi\cdot\rho} = \{(u_1,\dots,u_a,v_1,\dots,v_b) \mid (u_1,\dots,u_a) \in \mathcal{A}^\pi \wedge (v_1,\dots,v_b) \in \mathcal{A}^\rho\}$,
- $\langle(u_1,\dots,u_a,v_1,\dots,v_b)\rangle^{\pi\cdot\rho}_{cs} = \langle(u_1,\dots,u_a)\rangle^{\pi}_{cs} \cdot Q(|cs|) + \langle(v_1,\dots,v_b)\rangle^{\rho}_{cs}$,
- Σ consists of the defined symbols in $R^\pi \cup R^\rho \cup R$, where R is given by Figure 4. □

// $N \cdot M - 1 = (N-1)\cdot M + (M-1)$, which corresponds to the pair $(N-1, M-1)$;
// that is, the tuple $(\mathtt{seed}^1_\pi\ cs, \dots, \mathtt{seed}^a_\pi\ cs, \mathtt{seed}^1_\rho\ cs, \dots, \mathtt{seed}^b_\rho\ cs)$.

$$\begin{array}{lcll} \mathtt{seed}^i_{\pi\cdot\rho}\ cs & \to & \mathtt{seed}^i_\pi\ cs & [\![\text{for } 1 \le i \le a]\!] \\ \mathtt{seed}^i_{\pi\cdot\rho}\ cs & \to & \mathtt{seed}^{i-a}_\rho\ cs & [\![\text{for } a+1 \le i \le a+b]\!] \end{array}$$

// (n,m) represents 0 iff both n and m are 0.

$$\mathtt{zero}_{\pi\cdot\rho}\ cs\ u_1 \dots u_a\ v_1 \dots v_b \to \mathtt{ifelse}_{\mathtt{bool}}\ (\mathtt{zero}_\pi\ cs\ u_1 \dots u_a)\ (\mathtt{zero}_\rho\ cs\ v_1 \dots v_b)\ \mathtt{false}$$

// $(n,m) - 1$ results in $(n, m-1)$ if $m > 0$, otherwise in $(n-1, M-1)$.

$$\begin{array}{rcll} \mathtt{pred}^i_{\pi\cdot\rho}\ cs\ u_1 \dots u_a\ v_1\ \dots\ v_b & \to & \mathtt{ptest}^i_{\pi\cdot\rho}\ cs\ (\mathtt{zero}_\rho\ v_1 \dots v_b)\ u_1\ \dots\ u_a\ v_1 \dots v_b & \\ & & & [\![\text{for } 1 \le i \le a+b]\!] \\ \mathtt{ptest}^i_{\pi\cdot\rho}\ cs\ \mathtt{false}\ \vec{u}\ \vec{v} & \to & u_i & [\![\text{for } 1 \le i \le a]\!] \\ \mathtt{ptest}^i_{\pi\cdot\rho}\ cs\ \mathtt{false}\ \vec{u}\ \vec{v} & \to & \mathtt{pred}^{i-a}_\rho\ cs\ \vec{v} & [\![\text{for } a+1 \le i \le a+b]\!] \\ \mathtt{ptest}^i_{\pi\cdot\rho}\ cs\ \mathtt{true}\ \vec{u}\ \vec{v} & \to & \mathtt{pred}^i_\pi\ cs\ \vec{u} & [\![\text{for } 1 \le i \le a]\!] \\ \mathtt{ptest}^i_{\pi\cdot\rho}\ cs\ \mathtt{true}\ \vec{u}\ \vec{v} & \to & \mathtt{seed}^{i-a}_\rho\ cs\ \vec{v} & [\![\text{for } a+1 \le i \le a+b]\!] \end{array}$$

Figure 4: Rules for the product counting module $C_{\pi\cdot\rho}$ (Lemma 29)

Lemma 29 is powerful because it can be used iteratively. Starting from the counting module from Example 28, we can thus define a first-order $(\lambda n.(n+1)^a)$-counting module $C_{\mathtt{lin}\cdots\mathtt{lin}}$ for any a. To reach yet higher numbers, we follow the ideas from Example 11 and define counting rules on binary numbers represented as functional terms $F : \vec{\sigma} \Rightarrow \mathtt{bool}$.

Lemma 30. *If there is a P-counting module C_π of order K, then there is a $(\lambda n.2^{P(n)})$-counting module $C_{\mathtt{p}[\pi]}$ of order $K+1$.*

Proof. Write $N := P(|cs|)$ and let $C_\pi = (\sigma_1 \otimes \cdots \otimes \sigma_a, \Sigma, R, \mathcal{A}, \langle\cdot\rangle^\pi)$. We define the 2^P-counting module $C_{\mathtt{p}[\pi]}$ as $(\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_a \Rightarrow \mathtt{bool}, \Sigma^{\mathtt{p}[\pi]}, R^{\mathtt{p}[\pi]}, \mathcal{H}, \langle\cdot\rangle^{\mathtt{p}[\pi]})$, where:

- $\mathcal{H}_{cs}$ contains terms $q : \vec{\sigma} \Rightarrow \mathtt{bool}$ representing a bitstring $b_0 \dots b_{N-1}$ as follows: $q\ s_1 \cdots s_n$ reduces to `true` if $(s_1,\dots,s_n)$ represents a number i in C_π such that $b_i = \mathtt{1}$ and to `false` if it represents i with $b_i = \mathtt{0}$. Formally, $\mathcal{H}_{cs}$ is the set of all $q \in \mathcal{T}(\Sigma^{\mathtt{p}[\pi]} \cup \mathcal{C}, \emptyset)$ of type $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_a \Rightarrow \mathtt{bool}$, where:
  - for all $(s_1,\dots,s_a) \in \mathcal{A}_{cs}$: $q\ s_1 \cdots s_a$ reduces to `true` or `false`, but not both;
  - for all $(s_1,\dots,s_a),(t_1,\dots,t_a) \in \mathcal{A}_{cs}$: if $\langle(\vec{s})\rangle^\pi_{cs} = \langle(\vec{t})\rangle^\pi_{cs}$—so they represent the same number i—then $q\ s_1 \cdots s_a$ and $q\ t_1 \cdots t_a$ reduce to the same boolean value.

  For $q \in \mathcal{H}_{cs}$ and $i < N$, we can thus say either $q\cdot[i] \to^*_{R^{\mathtt{p}[\pi]}} \mathtt{true}$ or $q\cdot[i] \to^*_{R^{\mathtt{p}[\pi]}} \mathtt{false}$.

- Let $\langle q\rangle^{\mathsf{p}[\pi]}_{cs} = \sum_{i=0}^{N-1}\{2^{N-i-1} \mid q\ s_1\cdots s_a \to^*_R \mathtt{true}$ for some $(s_1,\dots,s_a)$ with $\langle(s_1,\dots,s_a)\rangle^{\pi}_{cs} = i\}$. That is, q represents the number given by the bitstring $b_0\dots b_N$ with b_N the least significant digit (where $b_i = 1$ if and only if $q\cdot[i] \to^*_{R^{\mathsf{p}[\pi]}} \mathtt{true}$).
- $\Sigma^{\mathsf{p}[\pi]} = \Sigma \cup \Sigma'$ and $R^{\mathsf{p}[\pi]} = R \cup R'$, where Σ' contains all new symbols in R', and R' contains the rules below along with rules for $\mathtt{equal}_\pi$ and $\mathtt{succ}_{\mathsf{p}[\pi]}$ following Figure 3.

// $\mathtt{seed}$ cs results in a bitstring that is 1 at all bits. We let $\mathtt{seed}_{\mathsf{p}[\pi]}$ cs be a normal form:
// a term of type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_a \Rightarrow \mathtt{bool}$ which maps all $[i]$ to $\mathtt{true}$.

$$\mathtt{seed}_{\mathsf{p}[\pi]}\ cs\ k_1\dots k_a \ \to\ \mathtt{true}$$

// A bitstring represents 0 if all its bits are set to 0. To test this, we count down in C_π and
// evaluate $F\ [N-1],\ F\ [N-2],\ \dots,\ F\ [0]$ to see whether any results in $\mathtt{false}$.

$$\begin{aligned}
\mathtt{zero}_{\mathsf{p}[\pi]}\ cs\ F \ &\to\ \mathtt{zero}'_{\mathsf{p}[\pi]}\ cs\ (\mathtt{seed}^1_\pi\ cs)\dots(\mathtt{seed}^a_\pi\ cs)\ F\\
\mathtt{zero}'_{\mathsf{p}[\pi]}\ cs\ k_1\dots k_a\ F \ &\to\ \mathtt{ifelse}_{\mathtt{bool}}\ (F\ k_1\cdots k_a)\ \mathtt{false}\\
&\qquad(\ \mathtt{ifelse}_{\mathtt{bool}}\ (\mathtt{zero}_\pi\ cs\ k_1\dots k_a)\ \mathtt{true}\\
&\qquad\quad(\ \mathtt{zero}'_{\mathsf{p}[\pi]}\ cs\ (\mathtt{pred}^1_\pi\ cs\ \vec{k})\dots(\mathtt{pred}^a_\pi\ cs\ \vec{k})\ F\ )\\
&\qquad)
\end{aligned}$$

// The predecessor function follows a similar approach to Examples 11 and 17: we flip b_i
// for $i = N-1, N-2, \dots$ until $b_i = \mathtt{1}$ (thus replacing $b_0\dots b_{i-1}\mathtt{10}\dots\mathtt{0}$ by $b_0\dots b_{i-1}\mathtt{01}\dots\mathtt{1}$).

$$\begin{aligned}
\mathtt{pred}_{\mathsf{p}[\pi]}\ cs\ F \ &\to\ \mathtt{predtest}_{\mathsf{p}[\pi]}\ cs\ (\mathtt{zero}_{\mathsf{p}[\pi]}\ F)\ cs\ F\\
\mathtt{predtest}_{\mathsf{p}[\pi]}\ cs\ \mathtt{true}\ F \ &\to\ F\\
\mathtt{predtest}_{\mathsf{p}[\pi]}\ cs\ \mathtt{false}\ F \ &\to\ \mathtt{predhelp}_{\mathsf{p}[\pi]}\ cs\ F\ (\mathtt{seed}^1_\pi\ cs)\dots(\mathtt{seed}^a_\pi\ cs)\\
\mathtt{predhelp}_{\mathsf{p}[\pi]}\ cs\ F\ \vec{k} \ &\to\ \mathtt{checkbit}_{\mathsf{p}[\pi]}\ cs\ (F\ \vec{k})\ (\mathtt{flip}_{\mathsf{p}[\pi]}\ cs\ F\ \vec{k})\ \vec{k}\\
\mathtt{checkbit}_{\mathsf{p}[\pi]}\ cs\ \mathtt{true}\ F\ \vec{k} \ &\to\ F\\
\mathtt{checkbit}_{\mathsf{p}[\pi]}\ cs\ \mathtt{false}\ F\ \vec{k} \ &\to\ \mathtt{predhelp}_{\mathsf{p}[\pi]}\ cs\ F\ (\mathtt{pred}^1_\pi\ cs\ \vec{k})\dots(\mathtt{pred}^a_\pi\ cs\ \vec{k})\\
\mathtt{flip}_{\mathsf{p}[\pi]}\ cs\ F\ \vec{k}\ \vec{n} \ &\to\ \mathtt{ifelse}_{\mathtt{bool}}\ (\mathtt{equal}_\pi\ cs\ \vec{k}\ \vec{n})\ (\mathtt{not}\ (F\ \vec{n}))\ (F\ \vec{n})\\
\mathtt{not}\ \mathtt{true} \ &\to\ \mathtt{false}\\
\mathtt{not}\ \mathtt{false} \ &\to\ \mathtt{true}
\end{aligned}$$

□

Combining Example 28 with Lemmas 29 and 30, we can define a $(\lambda n.\exp_2^{K-1}((n+1)^b))$-counting module $C_{\mathsf{p}[\dots[\mathsf{p}[\mathtt{lin}\cdots\mathtt{lin}]]\dots]}$ of type order K for any $K, b \geq 1$. As the ATRSs of Figure 2 and the modules are all non-overlapping, we thus recover one side of Jones' result: any problem in $\mathrm{EXP}^{K-1}\mathrm{TIME}$ is decided using a deterministic K^{th}-order cons-free ATRS.

Remark 31. The construction used here largely follows the one in [15]. Differences mostly center around the different formalisms: on the one hand Jones' language did not support pattern matching or constructors like $\mathtt{action}$; on the other, we had to code around the lack of pairs. Our notion of a counting module is more complex—restricting the way tuples of terms may be reduced—to support the non-deterministic modules we will consider below.

4.3. **Counting higher.** In ATRSs, we can do better than merely translating Jones' result. By exploiting non-determinism much like we did in § 3.2, we can count up to $2^{n+1}-1$ using only a first-order ATRS, and obtain the jump in expressivity promised in the introduction.

Lemma 32. *There is a first-order $(\lambda n.2^{n+1})$-counting module.*

Proof. Intuitively, we represent a bitstring $b_0 \dots b_N$ by a pair of non-normalized terms (yss, zss), such that $yss \to^*$ [a list of length i] iff $b_i = \mathtt{1}$ and $zss \to^*$ [a list of length i] iff $b_i = \mathtt{0}$. Formally, we let $C_{\mathtt{e}} := (\mathtt{list} \otimes \mathtt{list}, \Sigma, R, \mathcal{A}, \langle\cdot\rangle)$, where:

- $\mathcal{A}_{cs}$ contains all pairs (yss, zss) such that (a) all normal forms of yss or zss are subterms of cs, and (b) for each $u \trianglelefteq cs$ either $yss \to_R^* u$ or $zss \to_R^* u$, but not both.
- Writing $cs = c_N; \dots; c_1; []$, we let $cs_i = c_i; \dots c_1; []$ for $1 \le i \le N$. Let $\langle (yss, zss) \rangle_{cs} = \sum_{i=0}^{N} \{2^{N-i} \mid yss \to_R^* cs_i\}$; then $\langle (yss, zss) \rangle_{cs}$ is the number with bit representation $b_0 \dots b_N$ (most significant digit first) where $b_i = 1$ iff $yss \to_R^* cs_i$, iff $zss \not\to_R^* cs_i$.
- Σ consists of the defined symbols introduced in R, which we construct below.

We include the rules from Figure 3, the rules for $\mathtt{seed}^1_{\mathtt{lin}}$, $\mathtt{pred}^1_{\mathtt{lin}}$ and $\mathtt{zero}^1_{\mathtt{lin}}$ from Example 28—to handle the data lists—and $\mathtt{ifte}_{\mathtt{list}}$ defined similar to other $\mathtt{ifte}$ rules.

As in § 3.2, we use non-deterministic selection functions to construct (yss, zss):

$$\mathtt{either}\ n\ xss \to n \qquad \mathtt{either}\ n\ xss \to xss \qquad \bot \to \bot$$

The symbol $\bot$ will be used for terms which do not reduce to any data (the $\bot \to \bot$ rule serves to force $\bot \in \mathcal{D}$). As discussed in Remark 27, non-termination by itself is not an issue. For the remaining functions, we consider bitstring arithmetic. First, $2^{N+1} - 1$ corresponds to the bitstring where each $b_i = 1$, so yss reduces to all subterms of cs:

$$\begin{aligned}
\mathtt{seed}^1_{\mathtt{e}}\ cs &\to \mathtt{all}\ cs\ (\mathtt{seed}^1_{\mathtt{lin}}\ cs)\ \bot \\
\mathtt{seed}^2_{\mathtt{e}}\ cs &\to \bot \\
\mathtt{all}\ cs\ n\ xss &\to \mathtt{ifte}_{\mathtt{list}}\ (\mathtt{zero}^1_{\mathtt{lin}}\ cs\ n)\ (\mathtt{either}\ n\ xss) \\
&\qquad (\mathtt{all}\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ (\mathtt{either}\ n\ xss))
\end{aligned}$$

(The use of $\mathtt{seed}^1_{\mathtt{lin}}\ cs$ where simply cs would have sufficed may seem overly verbose, but is deliberate because it will make the results of § 6 easier to present.)

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

$$\begin{aligned}
\mathtt{eqLen}\ []\ [] &\to \mathtt{true} & \mathtt{eqLen}\ []\ (y;ys) &\to \mathtt{false} \\
\mathtt{eqLen}\ (x;xs)\ (y;ys) &\to \mathtt{eqLen}\ xs\ ys & \mathtt{eqLen}\ (x;xs)\ [] &\to \mathtt{false}
\end{aligned}$$

$$\begin{aligned}
\mathtt{bitset}\ n\ yss\ zss &\to \mathtt{checkreducts}\ (\mathtt{eqLen}\ n\ yss)\ (\mathtt{eqLen}\ n\ zss) \\
\mathtt{checkreducts}\ \mathtt{true}\ b &\to \mathtt{true} \\
\mathtt{checkreducts}\ b\ \mathtt{true} &\to \mathtt{false}
\end{aligned}$$

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

$$\begin{aligned}
\mathtt{zero}_{\mathtt{e}}\ cs\ yss\ zss &\to \mathtt{zo}\ cs\ (\mathtt{seed}^1_{\mathtt{lin}}\ cs)\ yss\ zss \\
\mathtt{zo}\ cs\ n\ yss\ zss &\to \mathtt{ifte}_{\mathtt{bool}}\ (\mathtt{bitset}\ n\ yss\ zss)\ \mathtt{false} \\
&\qquad (\ \mathtt{ifte}_{\mathtt{bool}}\ (\mathtt{zero}^1_{\mathtt{lin}}\ cs\ n)\ \mathtt{true}\ (\mathtt{zo}\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ yss\ zss)\ )
\end{aligned}$$

For the predecessor function, we again replace $b_0 \dots b_{i-1} b 10 \dots 0$ by $b_0 \dots b_{i-1} 01 \dots 1$. To do so, we fully rebuild yss and zss. We first define a helper function $\mathtt{copy}$ to copy $b_0 \dots b_{i-1}$:

$$\begin{aligned}
\mathtt{copy}\ cs\ n\ yss\ zss\ \mathtt{false} &\to \mathtt{addif}\ (\mathtt{bitset}\ n\ yss\ zss)\ n \\
&\qquad (\ \mathtt{copy}\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ yss\ zss\ (\mathtt{zero}_{\mathtt{lin}}\ cs\ n)\ ) \\
\mathtt{copy}\ cs\ n\ yss\ zss\ \mathtt{true} &\to \bot \\
\mathtt{addif}\ \mathtt{true}\ n\ xss &\to \mathtt{either}\ n\ xss \\
\mathtt{addif}\ \mathtt{false}\ n\ xss &\to xss
\end{aligned}$$

Then, for all i, $\mathtt{copy}\ cs\ cs_{\max(i-1,0)}\ yss\ zss\ [i=0]$ reduces to those cs_j with $0 \leq j < i$ where $b_j = 1$, and $\mathtt{copy}\ cs\ cs_{\max(i-1,0)}\ zss\ yss\ [i=0]$ reduces to those with $b_j = 0$. This works because yss and zss are complements. To define $\mathtt{pred}$, we first handle the zero case:

$$\begin{array}{lcl} \mathtt{pred}^1_{\mathtt{e}}\ cs\ yss\ zss & \to & \mathtt{ifte}_{\mathtt{list}}\ (\mathtt{zero}_{\mathtt{e}}\ cs\ yss\ zss)\ yss\ (\mathtt{pr}^1\ cs\ (\mathtt{seed}^1_{\mathtt{lin}}\ cs)\ yss\ zss) \\ \mathtt{pred}^2_{\mathtt{e}}\ cs\ yss\ zss & \to & \mathtt{ifte}_{\mathtt{list}}\ (\mathtt{zero}_{\mathtt{e}}\ cs\ yss\ zss)\ zss\ (\mathtt{pr}^2\ cs\ (\mathtt{seed}^1_{\mathtt{lin}}\ cs)\ yss\ zss) \end{array}$$

Then $\mathtt{pr}\ cs\ cs_N\ yss\ zss$ flips the bits $b_N, b_{N-1}, \ldots$ until an index is encountered where $b_i = 1$; this last bit is flipped, and the remaining bits are copied:

$$\begin{array}{lcl} \mathtt{pr}^1\ cs\ n\ yss\ zss & \to & \mathtt{ifte}_{\mathtt{list}}\ (\mathtt{bitset}\ n\ yss\ zss) \\ & & \quad (\ \mathtt{copy}\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ yss\ zss\ (\mathtt{zero}_{\mathtt{lin}}\ cs\ n)\ ) \\ & & \quad (\ \mathtt{either}\ n\ (\mathtt{pr}^1\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ yss\ zss)\ ) \\ \mathtt{pr}^2\ cs\ n\ yss\ zss & \to & \mathtt{ifte}_{\mathtt{list}}\ (\mathtt{bitset}\ n\ yss\ zss) \\ & & \quad (\ \mathtt{either}\ n\ (\mathtt{copy}\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ zss\ yss\ (\mathtt{zero}_{\mathtt{lin}}\ cs\ n))\ ) \\ & & \quad (\ \mathtt{pr}^2\ cs\ (\mathtt{pred}^1_{\mathtt{lin}}\ cs\ n)\ yss\ zss\ ) \end{array}$$

□

Note that, unlike Lemma 30, Lemma 32 cannot be used directly to define composite modules: the rules for $\mathtt{eqLen}$ rely on the specific choice of the underlying counting module $C_{\mathtt{lin}}$. They cannot be replaced by an $\mathtt{equals}_{\mathtt{lin}}$ check, because the crucial property is that—like in § 3.2—the $\mathtt{bitset}$ functionality relies on evaluating yss and zss to some normal form. Nevertheless, even without composing we obtain additional power:

Theorem 33. *Any decision problem in E^K TIME is accepted by a K^{th}-order cons-free ATRS.*

Proof. Following the construction in Figure 2, it suffices to 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 32 gives a first-order $\lambda n.2^{n+1}$-counting module, and by iteratively using Lemma 29 we obtain $\lambda n.(2^{n+1})^a = \lambda n.2^{a(n+1)}$ for any a. Iteratively applying Lemma 30 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 $\mathrm{E}^K\mathrm{TIME}$ can be implemented by a cons-free K^{th}-order ATRS. Towards a characterization result, we must therefore show the converse: that every function accepted by a cons-free K^{th}-order ATRS is in $\mathrm{E}^K\mathrm{TIME}$.

To achieve this goal, we will now give an algorithm running in TIME $\left(\exp_2^K(a \cdot n)\right)$ that, on input any basic term in a fixed ATRS of order K, outputs its set of data normal forms.

A key idea is to associate terms of higher-order type to functions. For a given set $\mathcal{B}$ of data terms (a shorthand for a set $\mathcal{B}_s$ following Definition 19), we let:

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

We will refer to the elements of each $[\![\sigma]\!]_{\mathcal{B}}$ as *term representations*. Intuitively, an element of $[\![\iota]\!]_{\mathcal{B}}$ represents a set of possible reducts of a term $s : \iota$, while an element of $[\![\sigma \Rightarrow \tau]\!]_{\mathcal{B}}$ represents the function defined by a functional term $s : \sigma \Rightarrow \tau$. Since each $[\![\sigma]\!]_{\mathcal{B}}$ is *finite*, we can enumerate its elements. In Algorithm 35 below, we build functions $\mathsf{Confirmed}^0, \mathsf{Confirmed}^1, \ldots$, each mapping statements $\mathsf{f}\ A_1 \cdots A_m \rightsquigarrow t$ to a value in $\{\top, \bot\}$. Intuitively, $\mathsf{Confirmed}^i[\mathsf{f}\ A_1 \cdots A_m \rightsquigarrow t]$ denotes whether, in step i in the algorithm, we have confirmed that $\mathsf{f}\ s_1 \cdots s_m$ has normal form t, where each A_j represents the corresponding s_j.

To achieve this, we will use two helper definitions. First:

Definition 34. For a defined symbol $\mathsf{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{D}$, rule $\rho\colon \mathsf{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R}$, variables $x_{k+1} : \sigma_{k+1}, \ldots, x_m : \sigma_m$ not occurring in ρ and $A_1 \in [\![\sigma_1]\!]_{\mathcal{B}}, \ldots, A_m \in [\![\sigma_m]\!]_{\mathcal{B}}$, let the *mapping associated to* ρ, $\vec{x}$ *and* $\mathsf{f}\ \vec{A}$ be the function η on domain $\{\ell_j \mid 1 \leq j \leq k \wedge \ell_j \in \mathcal{V}\} \cup \{x_{k+1}, \ldots, x_m\}$ such that $\eta(\ell_j) = A_j$ for $j \leq k$ with $\ell_j \in \mathcal{V}$, and $\eta(x_j) = A_j$ for $j > k$.

Second, the algorithm employs a function $\mathcal{NF}^i$ for all i, mapping a term $s : \sigma$ and a mapping η as above to an element of $[\![\sigma]\!]_{\mathcal{B}}$ (which depends on $\mathsf{Confirmed}^i$). Intuitively, if δ is a substitution such that each $\eta(x)$ represents $\delta(x)$, then $\mathcal{NF}^i(s, \eta)$ represents the term $s\delta$.

Algorithm 35.

Input: A basic term $s = \mathsf{g}\ s_1 \cdots s_M$.

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

Set $\mathcal{B} := \mathcal{B}_s$. For all $\mathsf{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{D}$ with $\iota \in \mathcal{S}$, all $A_1 \in [\![\sigma_1]\!]_{\mathcal{B}}, \ldots, A_m \in [\![\sigma_m]\!]_{\mathcal{B}}$, all $t \in [\![\iota]\!]_{\mathcal{B}}$, let $\mathsf{Confirmed}^0[\mathsf{f}\ A_1 \cdots A_m \rightsquigarrow t] := \bot$. For all such $\mathsf{f}, \vec{A}, t$ and all $i \in \mathbb{N}$:

- if $\mathsf{Confirmed}^i[\mathsf{f}\ \vec{A} \rightsquigarrow t] = \top$, then $\mathsf{Confirmed}^{i+1}[\mathsf{f}\ \vec{A} \rightsquigarrow t] := \top$;
- otherwise, for all $\rho\colon \mathsf{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R}$ and fresh variables $x_{k+1} : \sigma_{k+1}, \ldots, x_m : \sigma_m$, all substitutions γ on domain $Var(\mathsf{f}\ \vec{\ell}) \setminus \{\vec{\ell}\}$ such that $\ell_j\gamma \in A_j$ whenever $\ell_j \notin \mathcal{V}$, let η be the mapping associated to $\rho, \vec{x}$ and $\mathsf{f}\ \vec{A}$. Test whether $t \in \mathcal{NF}^i((r\ x_{k+1} \cdots x_m)\gamma, \eta)$. Let $\mathsf{Confirmed}^{i+1}[\mathsf{f}\ \vec{A} \rightsquigarrow t]$ be $\top$ if there are ρ, γ where this test succeeds, $\bot$ otherwise.

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

- if t is a data term, then $\mathcal{NF}^i(t, \eta) := \{t\}$;
- if $t = \mathsf{f}\ t_1 \cdots t_m$ with $\mathsf{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{D}$ (for $\iota \in \mathcal{S}$), then $\mathcal{NF}^i(t, \eta)$ is the set of all $u \in \mathcal{B}$ such that $\mathsf{Confirmed}^i[\mathsf{f}\ \mathcal{NF}^i(t_1, \eta) \cdots \mathcal{NF}^i(t_m, \eta) \rightsquigarrow u] = \top$;
- if $t = \mathsf{f}\ t_1 \cdots t_n$ with $\mathsf{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{D}$ (for $\iota \in \mathcal{S}$) and $n < m$, then $\mathcal{NF}^i(t, \eta) :=$ the function mapping $A_{n+1}, \ldots, A_m$ to the set of all $u \in \mathcal{B}$ such that $\mathsf{Confirmed}^i[\mathsf{f}\ \mathcal{NF}^i(t_1, \eta) \cdots \mathcal{NF}^i(t_n, \eta)\ A_{n+1} \cdots A_m \rightsquigarrow u] = \top$;
- if $t = x\ t_1 \cdots t_n$ with $n \geq 0$ and x a variable, then $\mathcal{NF}^i(t, \eta) := \eta(x)(\mathcal{NF}^i(t_1, \eta), \ldots, \mathcal{NF}^i(t_n, \eta))$; so also $\mathcal{NF}^i(t) = \eta(t)$ if t is a variable.

When $\mathsf{Confirmed}^{i+1}[\mathsf{f}\ \vec{A} \rightsquigarrow t] = \mathsf{Confirmed}^i[\mathsf{f}\ \vec{A} \rightsquigarrow t]$ for all statements, the algorithm ends; we let $I := i + 1$ and return $\{t \in \mathcal{B} \mid \mathsf{Confirmed}^I[\mathsf{g}\ \{s_1\} \cdots \{s_M\} \rightsquigarrow t] = \top\}$.

This is well-defined because a non-variable pattern ℓ_j necessarily has base type, which means A_j is a set. As $\mathcal{D}$, $\mathcal{B}$ and all $[\![\sigma_i]\!]_{\mathcal{B}}$ 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 Lemma 38.

Example 36. Consider the *majority* ATRS of Example 22, with starting term $s = \mathtt{majority}\ (\mathtt{1;0;[]})$. Then $\mathcal{B}_s = \{\mathtt{1}, \mathtt{0}, \mathtt{1;0;[]}, \mathtt{0;[]}, []\}$. We have $[\![\mathtt{symb}]\!]_{\mathcal{B}} = \{\emptyset, \{\mathtt{0}\}, \{\mathtt{1}\}, \{\mathtt{0}, \mathtt{1}\}\}$ and $[\![\mathtt{list}]\!]_{\mathcal{B}}$ is the set containing all eight subsets of $\{\mathtt{1;0;[]},\ \mathtt{0;[]},\ []\}$. Thus, there are $8 \cdot 2$ statements of the form $\mathtt{majority}\ A \rightsquigarrow t$, $8^3 \cdot 2$ statements of the form $\mathtt{count}\ A_1\ A_2\ A_3 \rightsquigarrow t$ and $8^2 \cdot 2$ of the form $\mathtt{cmp}\ A_1\ A_2 \rightsquigarrow t$; in total, 1168 statements are considered in each step.

We consider one statement in the first step, determining $\mathsf{Confirmed}^1[\mathtt{cmp}\ \{\mathtt{0;[]}\}\ \{\mathtt{0;[]},\ []\} \rightsquigarrow \mathtt{0}]$. There are two viable combinations of a rule and a substitution: $\mathtt{cmp}\ (y\mathtt{;}ys)\ (z\mathtt{;}zs) \to \mathtt{cmp}\ ys\ zs$ with substitution $\gamma = [y := \mathtt{0}, ys := [], z := \mathtt{0}, zs := []]$ and $\mathtt{cmp}\ (y\mathtt{;}ys)\ [] \to \mathtt{0}$

with substitution $\gamma = [y := \mathtt{0}, ys := []]$. Consider the first. As there are no functional variables, η is empty and we need to determine whether $\mathtt{0} \in \mathcal{NF}^1(\mathtt{cmp}\ []\ [], \emptyset)$. This fails, because $\mathsf{Confirmed}^0[\xi] = \bot$ for all statements ξ. However, the check for the second rule, $\mathtt{0} \in \mathcal{NF}^1(\mathtt{0}, \emptyset)$, succeeds. Thus, we mark $\mathsf{Confirmed}^1[\mathtt{cmp}\ \{\mathtt{0};[]\}\ \{\mathtt{0};[],\ []\} \rightsquigarrow \mathtt{0}] = \top$.

Before showing correctness of Algorithm 35, we see that it has the expected complexity.

Lemma 37. *If $(\mathcal{F}, \mathcal{R})$ has type order K, then Algorithm 35 runs in TIME $\big(\exp_2^K(a \cdot n)\big)$ for some a.*

Proof. Write $N := |\mathcal{B}|$; N is linear in the size of the only input, s ($\mathcal{R}$ and $\mathcal{F}$ are not considered input). We claim: if $K, d \in \mathbb{N}$ are such that σ has at most order K, and the longest sequence $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \iota$ occurring in σ has length $n+1 \leq d$, then $\mathtt{card}([\![\sigma]\!]_{\mathcal{B}}) \leq \exp_2^{K+1}(d^K \cdot N)$.

(Proof of claim.) Proceed by induction on the form of σ. Observe that $\mathbb{P}(\mathcal{B})$ has cardinality 2^N, so for $\iota \in \mathcal{S}$ also $\mathtt{card}([\![\iota]\!]_{\mathcal{B}}) \leq 2^N = \exp_2^1(d^0 \cdot N)$. For the induction step, write $\sigma = \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \iota$ with $n < d$ and each σ_j having order at most $K-1$. We have:

$$\begin{aligned}
\mathtt{card}([\![\sigma]\!]_{\mathcal{B}}) &= \mathtt{card}((\cdots([\![\iota]\!]_{\mathcal{B}}^{[\![\sigma_n]\!]_{\mathcal{B}}})^{[\![\sigma_{n-1}]\!]_{\mathcal{B}}} \ldots)^{[\![\sigma_1]\!]_{\mathcal{B}}}) = \mathtt{card}([\![\iota]\!]_{\mathcal{B}})^{\mathtt{card}([\![\sigma_n]\!]_{\mathcal{B}}) \cdots \mathtt{card}([\![\sigma_1]\!]_{\mathcal{B}})} \\
&\leq 2\hat{}(N \cdot \mathtt{card}([\![\sigma_n]\!]_{\mathcal{B}}) \cdots \mathtt{card}([\![\sigma_1]\!]_{\mathcal{B}})) \leq 2\hat{}(N \cdot \exp_2^K(d^{K-1} \cdot N) \cdots \exp_2^K(d^{K-1} \cdot N)) \\
&= 2\hat{}(N \cdot \exp_2^K(d^{K-1} \cdot N)^n) \leq 2\hat{}(\exp_2^K(d^{K-1} \cdot N \cdot n + N)) \quad \text{(by induction on } K \geq 1) \\
&= \exp_2^{K+1}(n \cdot d^{K-1} \cdot N + N) \leq \exp_2^{K+1}(d \cdot d^{K-1} \cdot N) = \exp_2^{K+1}(d^K \cdot N) \quad (n+1 \leq d)
\end{aligned}$$

(End of proof of claim.)

Since, in a K^{th}-order ATRS, all arguments types have order at most $K-1$, we thus find d (depending solely on $\mathcal{F}$) such that all sets $[\![\sigma]\!]_{\mathcal{B}}$ in the algorithm have cardinality $\leq \exp_2^K(d^{K-1} \cdot N)$. Writing a for the maximal arity in $\mathcal{F}$, there are therefore at most $|\mathcal{D}| \cdot \exp_2^K(d^{K-1} \cdot N)^a \cdot N \leq |\mathcal{D}| \cdot \exp_2^K((d^{K-1} \cdot a + 1) \cdot N)$ distinct statements $\mathsf{f}\ \vec{A} \rightsquigarrow t$.

Writing $m := d^{K-1} \cdot a + 1$ and $X := |\mathcal{D}| \cdot \exp_2^K(m \cdot N)$, we thus find: the algorithm has at most $I \leq X + 2$ steps, and in each step i 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 $t \in \mathcal{NF}^i((r\ \vec{x})\gamma, \eta)$ at most $X \cdot (X+2) \cdot |\mathcal{R}| \cdot 2^{aN}$ times. The exact cost of calculating $\mathcal{NF}^i((r\ \vec{x})\gamma, \eta)$ 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 $\mathcal{O}(X \cdot (X+1) \cdot 2^{aN} \cdot P(X)) = P'(\exp_2^K(m \cdot N))$ for some polynomial P' and constant m. As $\mathrm{E}^K\mathrm{TIME}$ is robust under taking polynomials, the result follows. □

5.1. **Algorithm correctness.** The one remaining question is whether our algorithm accurately simulates rewriting. This is set out in Lemma 38.

Lemma 38. *Let $\mathsf{g} : \iota_1 \Rightarrow \ldots \Rightarrow \iota_M \Rightarrow \iota \in \mathcal{D}$ and $s_1 : \iota_1, \ldots, s_M : \iota_M, t : \iota$ be data terms. Then $\mathsf{Confirmed}^I[\mathsf{g}\ \{s_1\} \cdots \{s_M\} \rightsquigarrow t] = \top$ if and only if $\mathsf{g}\ s_1 \cdots s_M \to_{\mathcal{R}}^* t$. (Here, I is the point at which the algorithm stops progressing, as defined in the last line of Algorithm 35.)*

A key understanding for Lemma 38 is that algorithm 35 traces *semi-outermost* reductions:

Definition 39. A reduction $s \to_{\mathcal{R}}^* t$ is semi-outermost if either $s = t$, or it has the form $s = \mathsf{f}\ s_1 \cdots s_n \to_{\mathcal{R}}^* \mathsf{f}\ (\ell_1\gamma) \cdots (\ell_k\gamma)\ s_{k+1} \cdots s_m \to_{\mathcal{R}} (r\gamma)\ s_{k+1} \cdots s_m \to_{\mathcal{R}}^* t$, the sub-reductions $s_i \to_{\mathcal{R}}^* \ell_i\gamma$ and $(r\gamma)\ s_{k+1} \cdots s_m \to_{\mathcal{R}}^* t$ are semi-outermost, and $s_j = \ell_j\gamma$ whenever $\ell_j \in \mathcal{V}$.

Proof Idea of Lemma 38. By postponing reductions at argument positions until needed, we can safely assume that any reduction in a cons-free ATRS is semi-outermost. Then, writing $s \approx A$ to indicate that s is "represented" by A, we prove by induction:

- if $s_j \approx A_j$ for $1 \leq j \leq m$, then $\mathsf{Confirmed}^I[\mathsf{f}\ A_1 \cdots A_m \rightsquigarrow t]$ iff $\mathsf{f}\ s_1 \cdots s_m \to_{\mathcal{R}}^* t$;
- if δ and η have the same domain, and both $\delta(x) \approx \eta(x)$ for all x and $t_j \approx A_j$ for $1 \leq j \leq n$, then $t \in \mathcal{NF}^I(s,\eta)(A_1,\ldots,A_n)$ iff $(s\delta)\ t_1 \cdots t_n \to_{\mathcal{R}}^* t$.

Lemma 38 is then obtained as an instance of the former statement. □

To translate this intuition to a formal proof we must overcome three difficulties: to translate an arbitrary reduction into a semi-outermost one, to associate terms to term representations, and to find an ordering to do induction on (as, in practice, neither induction on the algorithm nor on reduction lengths works very well with the definition of $\mathcal{NF}^i$). The first challenge would be easily handled by an induction on terms if $\to_{\mathcal{R}}$ were terminating, but that is not guaranteed. To solve this issue, we will define a terminating relation corresponding to $\to_{\mathcal{R}}$. This will also be very useful for the latter two challenges.

Definition 40 (Labeled system)**.** Let $\mathcal{F}_{\mathtt{lab}} := \mathcal{C} \cup \{\mathsf{f}_i : \sigma \mid \mathsf{f} : \sigma \in \mathcal{D} \wedge i \in \mathbb{N}\}$. For $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and $i \in \mathbb{N}$, let $\mathsf{label}_i(s)$ be s with all instances of any defined symbol f replaced by f_i. For $t \in \mathcal{T}(\mathcal{F}_{\mathtt{lab}},\mathcal{V})$, let $\|t\|$ be t with all symbols f_i replaced by f. Then, let

$$\mathcal{R}_{\mathtt{lab}} = \{\mathsf{f}_{i+1} \to \mathsf{f}_i \mid \mathsf{f} \in \mathcal{D} \wedge i \in \mathbb{N}\} \cup \{\mathsf{f}_{i+1}\ \ell_1 \cdots \ell_k \to \mathsf{label}_i(r) \mid \mathsf{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R} \wedge i \in \mathbb{N}\}$$

Note that constructor terms are unaffected by label_i and $\|\cdot\|$. The ATRS $(\mathcal{F}_{\mathtt{lab}},\mathcal{R}_{\mathtt{lab}})$ is both non-deterministic and infinite in its signature and rules, but can be used as a reasoning tool because data normal forms correspond between the labeled and unlabeled system:

Lemma 41. *For all* $\mathsf{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{D}$ *and data terms* $s_1,\ldots,s_m,t$*:*

$$\mathsf{f}\ s_1 \cdots s_m \to_{\mathcal{R}}^* t \text{ if and only if } \mathsf{f}_i\ s_1 \cdots s_m \to_{\mathcal{R}_{\mathtt{lab}}}^* t \text{ for some } i$$

Proof. The *if* direction is trivial, as $u \to_{\mathcal{R}_{\mathtt{lab}}} v$ clearly implies that $\|u\| \to_{\mathcal{R}} \|v\|$ or $\|u\| = \|v\|$. For the *only if* direction, note that $u \to_{\mathcal{R}} v$ implies $\mathsf{label}_{i+1}(u) \to_{\mathcal{R}_{\mathtt{lab}}}^* \mathsf{label}_i(v)$ for any i, by using the labeled rule $\mathsf{f}_{i+1}\ \ell_1 \cdots \ell_k \to \mathsf{label}_i(r)$ if the step $u \to_{\mathcal{R}} v$ uses rule $\mathsf{f}\ \ell_1 \cdots \ell_k \to r$ and using the labeled rules $\mathsf{g}_{i+1} \to \mathsf{g}_i$ to lower the labels of all other symbols in u. □

Despite the label decrease, termination of $\to_{\mathcal{R}_{\mathtt{lab}}}$ is non-obvious due to variable copying. For example, a pair of rules $\mathtt{f}_1\ (\mathtt{c}\ F) \to F$, $\mathtt{g}_2\ x \to \mathtt{f}_1\ x\ x$ with the constructor $\mathtt{c} : (\iota \Rightarrow \iota) \Rightarrow \iota$ is non-terminating through the term $\mathtt{f}_1\ (\mathtt{c}\ \mathtt{g}_2)\ (\mathtt{c}\ \mathtt{g}_2)$. In our setting, such rules can be assumed not to occur by Lemma 23, however. Thus, we indeed obtain:

Lemma 42. *There is no infinite* $\to_{\mathcal{R}_{\mathtt{lab}}}^*$ *reduction.*

Proof. We use a computability argument reminiscent of the one used for the *computability path ordering* [9] (CPO does not apply directly due to our applicative term structure). First, we define *computability* by induction on types: (a) $s : \iota \in \mathcal{S}$ is computable if s is terminating: there is no infinite $\to_{\mathcal{R}_{\mathtt{lab}}}^*$-reduction starting in s; (b) $s : \sigma \Rightarrow \tau$ is computable if $s\ t$ is computable for all computable $t : \sigma$. Note that (I) every computable term is terminating and (II) if s is computable and $s \to_{\mathcal{R}_{\mathtt{lab}}} t$, then t is computable. Also, (III), if $\ell\gamma$ is computable for a pattern ℓ, then $\gamma(x)$ is computable for all $x \in Var(\ell)$: if x has base type then $\gamma(x)$ is a subterm of a terminating term by (I), otherwise (by Lemma 23) $\ell = x$ and $\gamma(x) = \ell\gamma$.

We first observe: every variable, constructor symbol and defined symbol f_0 is computable: let $a : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$ be such a symbol; computability follows if $a\ s_1 \cdots s_m$ is terminating for all computable $s_1 : \sigma_1,\ldots,s_m : \sigma_m$. We use induction on $(s_1,\ldots.s_m)$ (using the product extension of $\to_{\mathcal{R}_{\mathtt{lab}}}$, which is well-founded on computable terms by (I)) and conclude with (II) and the induction hypothesis since $a\ s_1 \cdots s_m$ can only be reduced by reducing some s_i.

Next we see: every defined symbol f_i is computable, by induction on i. For f_0 we are done; for $\mathsf{f}_{i+1} : \sigma_1 \Rightarrow \ldots \Rightarrow_m \Rightarrow \iota$ we must show termination of $\mathsf{f}_{i+1}\ s_1 \cdots s_m$ for computable $\vec{s}$. We are done if every reduct is terminating. By induction on $\vec{s}$ by $\to_{\mathcal{R}_{\mathtt{lab}}}$ as before, we are done for reduction steps inside any s_j. Also $\mathsf{f}_i\ s_1 \cdots s_m$ is computable as $i < i+1$. This leaves only head reductions $\mathsf{f}_{i+1}\ s_1 \cdots s_m \to_{\mathcal{R}_{\mathtt{lab}}} (\mathsf{label}_i(r)\gamma)\ s_{k+1} \cdots s_m$ for some $\mathsf{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R}$ with each $s_j = \ell_j\gamma$. Certainly $(\mathsf{label}_i(r)\gamma)\ s_{k+1} \cdots s_m$ is terminating if $\mathsf{label}_i(r)\gamma$ is computable. We prove this by a third induction on r, observing that each $\gamma(x)$ is computable by (III):

Write $r = a\ r_1 \cdots r_n$ with $x \in \mathcal{V} \cup \mathcal{F}$. Then $\mathsf{label}_i(r)\gamma = u\ (\mathsf{label}_i(r_1)\gamma) \cdots (\mathsf{label}_i(r_n)\gamma)$ with $u = \gamma(a)$ or $u \in \mathcal{C}$ or $u = \mathsf{g}_i$; using the observations above and the first induction hypothesis, u is computable in all cases. By the third induction hypothesis, also each $\mathsf{label}_i(r_j)\gamma$ is computable, so $\mathsf{label}_i(r)\gamma$ is a base-type application of computable terms. □

Thus we obtain (a slight variation of) the first step of the proof intuition:

Lemma 43. *If $s \to^*_{\mathcal{R}_{\mathtt{lab}}} t \in \mathcal{DA}$ and s is $\mathcal{B}$-safe, then $s \to^*_{\mathcal{R}_{\mathtt{lab}}} t$ by a semi-outermost reduction.*

Proof. By induction on s using $\to_{\mathcal{R}_{\mathtt{lab}}} \cup \rhd$. If $s = t$ we are done, otherwise (by $\mathcal{B}$-safety) $s = \mathsf{f}_i\ s_1 \cdots s_n$ with f_i not occurring in t. Thus, a head step must be done: $s = \mathsf{f}_i\ s_1 \cdots s_n \to^*_{\mathcal{R}_{\mathtt{lab}}} \mathsf{f}_i\ (\ell_1\gamma) \cdots (\ell_k\gamma)\ s'_{k+1} \cdots s'_n \to_{\mathcal{R}_{\mathtt{lab}}} (r\gamma)\ s_{k+1'} \cdots s'_n$ for some rule $\mathsf{f}_i\ \ell_1 \cdots \ell_k \in \mathcal{R}$, substitution γ and $s'_{k+1}, \ldots, s'_n$ such that $s_i \to^*_{\mathcal{R}_{\mathtt{lab}}} \ell_i\gamma$ for $1 \le i \le k$ and $s_i \to^*_{\mathcal{R}_{\mathtt{lab}}} s'_i$ for $k < i \le n$.

Now let $\delta := [x := \gamma(x) \mid x \text{ occurs as a strict subterm of some } \ell_j] \cup [\ell_j := s_j \mid 1 \le j \le k \wedge \ell_j \text{ is a variable}]$. Since all variables occurring in a pattern ℓ_j are subterms of ℓ_j, clearly $s \to^*_{\mathcal{R}_{\mathtt{lab}}} \mathsf{f}_i\ (\ell_1\delta) \cdots (\ell_k\delta)\ s_{k+1} \cdots s_n \to_{\mathcal{R}_{\mathtt{lab}}} (r\delta)\ s_{k+1} \cdots s_n \to^*_{\mathcal{R}_{\mathtt{lab}}} \mathsf{f}_i\ (\ell_1\gamma) \cdots (\ell_k\gamma)\ s'_{k+1} \cdots s'_n \to^*_{\mathcal{R}_{\mathtt{lab}}} t$. Then $s_j = \ell_j\delta$ if ℓ_j is a variable, and by Lemma 21 and the induction hypothesis ($\rhd$ part for each s_j and $\to_{\mathcal{R}_{\mathtt{lab}}}$ part otherwise), all relevant sub-reductions are semi-outermost. □

The second difficulty of the proof idea is in the way terms are associated with term representations. Within the algorithm, a single term can have *multiple* representations; for example, a term s which reduces to `true` and `false` is represented both by $\{\mathtt{false}\}$ and $\{\mathtt{true}, \mathtt{false}\}$. This is necessary, because different normal forms are derived at different times, and may depend on each other; for example, in an ATRS $\{\mathtt{or\ true}\ x \to \mathtt{true},\ \mathtt{or\ false}\ x \to x,\ \mathtt{f} \to \mathtt{false},\ \mathtt{f} \to \mathtt{or\ f\ true},\ \mathtt{g} \to \mathtt{h}\}$, we need to use that $\mathcal{NF}^1(\mathtt{f}) = \{\mathtt{false}\}$ to obtain $\mathcal{NF}^2(\mathtt{f}) = \{\mathtt{true}, \mathtt{false}\}$. To reflect these levels, we will continue to use labeled terms:

Definition 44. Let $\approx$ be the smallest relation such that $s \approx A$ if we can write $s = \mathsf{label}_i(t)\delta$ and $A = \mathcal{NF}^i(t, \eta)$ for some i, t, δ, η such that δ and η have the same domain and each $\delta(x) \approx \eta(x)$. Here, $\mathcal{NF}^i := \mathcal{NF}^I$ if $i > I$.

The final challenge of the proof idea, the induction, can be handled in the same way: we will use induction on labeled terms using $\to^*_{\mathcal{R}_{\mathtt{lab}}}$. Thus, we are ready for the formal proof:

Proof of Lemma 38. Writing $\mathsf{Confirmed}^i := \mathsf{Confirmed}^I$ for all $i > I$, we will see, for all relevant $i \in \mathbb{N}, \mathsf{f} \in \mathcal{D}, u, \vec{s} \in \mathcal{T}(\mathcal{F}_{\mathtt{lab}}, \mathcal{V}), t \in \mathcal{B}$, and term representations $\vec{A}, D$:

(A): if $s_j \approx A_j$ for $1 \le j \le m$, then $\mathsf{Confirmed}^i[\mathsf{f}\ A_1 \cdots A_m \rightsquigarrow t]$ if and only if $q := \mathsf{f}_i\ s_1 \cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} t$ by a semi-outermost reduction;

(B): if $s_j \approx A_j$ for $1 \le j \le m$ and $u \approx D$, then $t \in D(A_1, \ldots, A_m)$ if and only if $q := u\ s_1 \cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} t$ by a semi-outermost reduction.

This proves the lemma because, for data terms s_j, a trivial induction on the definition of $\approx$ shows that $s_j \approx A_j$ iff $A_j = \{s_j\}$. Thus: $\mathsf{g}\ s_1 \cdots s_M \to^*_{\mathcal{R}} t$ if and only if $\mathsf{g}_i\ s_1 \cdots s_M \to^*_{\mathcal{R}_{\mathtt{lab}}} t$

for some i (Lemma 41), if and only if the same holds with a semi-outermost reduction (Lemma 43), if and only if $\mathsf{Confirmed}^i[\mathsf{g}\ \{s_1\}\cdots\{s_M\} \rightsquigarrow t] = \top$ for some i (A). Since $\mathsf{Confirmed}^i[\xi]$ implies $\mathsf{Confirmed}^I[\xi]$ for all i, we have the required equivalence.

We prove (A) and (B) together by a mutual induction on q, oriented with $\to_{\mathcal{R}_{\mathtt{lab}}} \cup \rhd$.

(A), only if case. Suppose $\mathsf{Confirmed}^i[\mathsf{f}\ A_1\cdots A_m \rightsquigarrow t] = \top$, and each $s_j \approx A_j$. Then $i > 0$, and if $\mathsf{Confirmed}^{i-1}[\mathsf{f}\ A_1\cdots A_m \rightsquigarrow t] = \top$, then the induction hypothesis yields $q \to_{\mathcal{R}_{\mathtt{lab}}} \mathsf{f}_{i-1}\ s_1\cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} t$ by the rule $\mathsf{f}_i \to \mathsf{f}_{i-1}$, so we are done.

Otherwise, there exist a rule $\mathsf{f}\ \ell_1\cdots\ell_k \to r \in \mathcal{R}$, variables $x_{k+1},\dots,x_m$ and a substitution γ on domain $Var(\mathsf{f}\ \vec{\ell}) \setminus \{\vec{\ell}\}$ such that (a) $\ell_j\gamma \in A_j$ for all non-variable ℓ_j and (b) $t \in \mathcal{NF}^{i-1}((r\ x_{k+1}\cdots x_m)\gamma, \eta)$ where η maps each variable ℓ_j to A_j, and x_j to A_j for $j > k$.

By part (B) of the induction hypothesis—since $q \rhd s_j$—(a) implies that (c) $s_j \to^*_{\mathcal{R}_{\mathtt{lab}}} \ell_j\gamma$ by a semi-outermost reduction for all non-variable ℓ_j. Now, if we let $\delta := [\ell_j := s_j \mid 1 \le j \le k \wedge \ell_j \in \mathcal{V}] \cup [x_j := s_j \mid k < j \le m]$ we have $\delta(x) \approx \eta(x)$ for all x. This gives:

$$\begin{array}{lll} q = \mathsf{f}_i\ s_1\cdots s_m & \to^*_{\mathcal{R}_{\mathtt{lab}}} (\mathsf{f}_i\ \ell_1\cdots\ell_k\ x_{k+1}\cdots x_m)\gamma\delta & \text{(by (c) and definition of } \delta) \\ & \to_{\mathcal{R}_{\mathtt{lab}}} (\mathsf{label}_{i-1}(r)\ x_{k+1}\cdots x_m)\gamma\delta & \text{(by the labeled rule for } \mathsf{f}\ \ell_1\cdots\ell_k \to r) \\ & = \mathsf{label}_{i-1}((r\ x_{k+1}\cdots x_m)\gamma)\delta & \end{array}$$

Since at least one step is done and $\mathsf{label}_{i-1}(v)\delta \approx \mathcal{NF}^{i-1}(v,\eta)$ for the $\mathcal{B}$-safe term $v = (r\ x_{k+1}\cdots x_m)\gamma$, we can use induction hypothesis (B) on observation (b) to derive that $q \to^*_{\mathcal{R}_{\mathtt{lab}}} \mathsf{label}_{i-1}((r\ x_{k+1}\cdots x_m)\gamma)\delta \to^*_{\mathcal{R}_{\mathtt{lab}}} t$. This reduction is semi-outermost.

(A), if case. Suppose $q = \mathsf{f}_i\ s_1\cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} t$ by a semi-outermost reduction. Since t cannot still contain f_i, this is not the empty reduction, so either

$$q = \mathsf{f}_i\ s_1\cdots s_m \to_{\mathcal{R}_{\mathtt{lab}}} \mathsf{f}_{i-1}\ s_1\cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} t$$

in which case induction hypothesis (A) gives $\mathsf{Confirmed}^{i-1}[\mathsf{f}\ A_1\cdots A_m \rightsquigarrow t] = \top$, or

$$q = \mathsf{f}_i\ s_1\cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} (\mathsf{f}_i\ \ell_1\cdots\ell_k\ x_{k+1}\cdots x_m)\gamma \to_{\mathcal{R}_{\mathtt{lab}}} \mathsf{label}_{i-1}(r\ x_{k+1}\cdots x_m)\gamma \to^*_{\mathcal{R}_{\mathtt{lab}}} t$$

for some rule $\mathsf{f}\ \ell_1\cdots\ell_k \to r \in \mathcal{R}$, substitution γ and fresh variables $x_{k+1},\dots,x_m$. Here, $\gamma(x_j) = s_j$ for all $j > k$ and $\gamma(\ell_j) = s_j$ for those ℓ_j which are variables. By induction hypothesis (B), $\ell_j\gamma \in A_j$ whenever ℓ_j is not a variable. Splitting $\gamma := \gamma_1 \uplus \gamma_2$—where γ_1 has domain $\{x \mid x \text{ occurs in some non-variable } \ell_j\}$ and γ_2 has the remainder—and writing $\eta_2 := [\ell_j := A_j \mid 1 \le j \le k \wedge \ell_j \text{ is a variable}] \cup [x_j := A_j \mid k < j \le m]$, we have $\gamma_2(x) \approx \eta_2(x)$ for all x in the shared domain. Therefore $\mathsf{label}_{i-1}(r\ x_{k+1}\cdots x_m)\gamma = (\mathsf{label}_{i-1}(r\ x_{k+1}\cdots x_m)\gamma_1)\gamma_2 \approx \mathcal{NF}^{i-1}((r\ x_{k+1}\cdots x_m)\gamma_1, \eta_2)$, and we obtain $t \in \mathcal{NF}^{i-1}((r\ x_{k+1}\cdots x_m)\gamma_1, \eta_2)$ by IH (B).

Thus, in either case, $\mathsf{Confirmed}^i[\mathsf{f}\ A_1\cdots A_m \rightsquigarrow t] = \top$ follows immediately.

(B), both cases. We prove (B) by an additional induction on the definition of $u \approx D$. Observe that $u \approx D$ implies that $u = \mathsf{label}_i(v)\delta$ and $D = \mathcal{NF}^i(v,\eta)$ for some v, i, δ, η such that each $\delta(x) \approx \eta(x)$. Consider the form of the $\mathcal{B}$-safe term v.

- If $v \in \mathcal{DA}$, then $m = 0$ and $t \in D = \mathcal{NF}^i(v,\eta)$ iff $t = v = \mathsf{label}_i(v) = u$.
- If $v = \mathsf{f}\ v_1\cdots v_n$ with $\mathsf{f} \in \mathcal{D}$, then denote $C_j := \mathcal{NF}^i(v_j,\eta)$ for $1 \le j \le n$; we have $t \in D(A_1,\dots,A_m)$ iff $\mathsf{Confirmed}^i[\mathsf{f}\ C_1\cdots C_n\ A_1\cdots A_m \rightsquigarrow t] = \top$. By case (A), this holds iff $q = (\mathsf{label}_i(v)\delta)\ s_1\cdots s_m = \mathsf{f}_i\ (\mathsf{label}_i(v_1)\delta)\cdots(\mathsf{label}_i(v_n)\delta)\ s_1\cdots s_m \to^*_{\mathcal{R}_{\mathtt{lab}}} t$.
- If $v = x\ v_1\cdots v_n$ with $x \in \mathcal{V}$, then denote $C_j := \mathcal{NF}^i(v_j,\eta)$ for $1 \le j \le n$; then clearly $\mathsf{label}_i(v_j)\delta \approx C_j$. We observe that, on the one hand,

$$\begin{array}{rcll} D(A_1,\dots,A_m) & = & \mathcal{NF}^i(v,\eta)(A_1,\dots,A_m) & = & (\eta(x)(C_1,\dots,C_n))(A_1,\dots,A_m) \\ & & & = & \eta(x)(C_1,\dots,C_n,A_1,\dots,A_m) \end{array}$$

And on the other hand,

$$q = (\mathsf{label}_i(v)\delta)\ s_1 \cdots s_m = \delta(x)\ (\mathsf{label}_i(v_1)\delta) \cdots (\mathsf{label}_i(v_n)\delta)\ s_1 \cdots s_m$$

As $\delta(x) \approx \eta(x)$ is used in the derivation of $u \approx D$, the second induction hypothesis gives the desired equivalence. □

And from Lemmas 37 and 38 together we obtain:

Theorem 45. *Any decision problem accepted by a cons-free K^{th}-order ATRS is in E^K TIME.*

Proof. By Lemma 38, decision problems accepted by a cons-free K^{th}-order ATRS are decided by Algorithm 35; by Lemma 37, this algorithm operates within $\bigcup_{a\in\mathbb{N}} \text{TIME}\left(\exp_2^K(an)\right)$. □

5.2. **Characterization result.** Combining Theorems 33 and 45 we thus find:

Corollary 46. *A decision problem X is in E^K TIME iff there is a K^{th}-order cons-free ATRS which accepts X: the class of cons-free ATRSs with order K* characterizes *E^K TIME.*

Remark 47. There are many similarities between the algorithm and correctness proof presented here and those in Jones' work, most pertinently the use of *memoization*. We have chosen to use a methodology which suits better with the semantics of term rewriting than the derivation trees of [15], for example by enumerating all possible reductions beforehand rather than using caching, but this makes little practical difference. We have also had to make several changes for the non-determinism and different evaluation strategy. For example the step to semi-outermost reductions is unique to this setting, and the term representations are different than they must be in the deterministic (or call-by-value) cases.

6. Pairing

Unlike our applicative term rewriting systems, Jones' minimal language in [15] includes *pairing*. While not standard in term rewriting, some styles of higher-order rewriting also admit pairs. We consider whether this feature affects expressivity of the considered systems.

Definition 48. An *Applicative Pairing Term Rewriting System* (APTRS) is defined following the definitions for ATRSs in § 2.2, with the following changes:

- In Definition 4 (simple types): if σ, τ are types of order n, m, then also $\sigma \times \tau$ is a type of order $\max(n, m)$; the pairing constructor $\times$ is considered right-associative.
- In Definition 5 (terms): terms are expressions typable by clauses (a), (b), (c), where (c) is: $(s, t) : \sigma \times \tau$ if $s : \sigma$ and $t : \tau$. Pairing is right-associative, so $(s, t, u) = (s, (t, u))$.
- In Definition 12 (patterns, data and basic terms): a term ℓ is a *pattern* if (a), (b) or (c) holds, where (c) is $\ell = (\ell_1, \ell_2)$ with ℓ_1 and ℓ_2 both patterns.

The last item is used to define *constructor APTRSs* as before.

Cons-freeness for left-linear constructor APTRSs is unaltered from Definition 16; however, pairing is not a constructor, so may occur freely in both sides of rules. Lemmas 21 and 23 go through unmodified, but constructors *can* have a product type of order 0 as argument type.

In a deterministic setting, pairing makes no difference: a function $\mathsf{f} : (\sigma \times \tau) \Rightarrow \pi$ can be replaced by a function $\mathsf{f} : \sigma \Rightarrow \tau \Rightarrow \pi$ with two arguments, and a function $\mathsf{f} : \pi \Rightarrow (\sigma \times \tau)$ by *two* functions $\mathsf{f}^1 : \pi \Rightarrow \sigma$ and $\mathsf{f}^2 : \pi \Rightarrow \tau$. We exploited this when defining counting modules (in [15], a number is represented by a single term, which may have product type). However, when allowing non-deterministic choice, pairing does increase expressivity—alarmingly so.

Lemma 49. *Suppose counting modules are defined over APTRSs. If there is a first-order P-counting module $C_\pi = (\sigma \otimes \tau, \Sigma^\pi, R^\pi, A^\pi, \langle\cdot\rangle^\pi)$, then there is a first-order $(\lambda n.2^{P(n)-1})$-counting module $C_{(\pi\pi)} = ((\sigma \times \tau) \otimes (\sigma \times \tau), \Sigma^{(\pi\pi)}, R^{(\pi\pi)}, A^{(\pi\pi)}, \langle\cdot\rangle^{(\pi\pi)})$.*

Proof. By using pairing, the ideas of Lemma 32 can be used to create a *composite* module. We will use almost the same rules, but replace the underlying module $C_{\mathtt{lin}}$ by C_π. We say $s \mapsto i$ if there is $(t,u) \in \mathcal{A}^\pi_{cs}$ such that $s \to^*_R (t,u)$ and $\langle (t,u)\rangle^\pi_{cs} = i$. A bitstring $b_0 \dots b_N$ is represented by a pair (yss, zss) such that $yss \mapsto i$ iff $b_i = \mathtt{1}$ and $zss \mapsto i$ iff $b_i = \mathtt{0}$.

- $\mathcal{A}^{(\pi\pi)}_{cs}$ contains all pairs (yss, zss) where
  - for all $0 \le i < P(n)$: either $yss \mapsto i$ or $zss \mapsto i$, but not both;
  - if $yss \to^*_R (u,v)$ then there is $(u',v') \in \mathcal{A}^\pi_{cs}$ such that $yss \to^*_R (u',v') \to^*_R (u,v)$ (thus, any pair which yss reduces to is a number in C_π, or a reduct thereof);
  - if $zss \to^*_R (u,v)$ then there is $(u',v') \in \mathcal{A}^\pi_{cs}$ such that $zss \to^*_R (u',v') \to^*_R (u,v)$.
- $\langle (s,t)\rangle^{(\pi\pi)}_{cs} = \sum_{i=0}^{P(|cs|)-1} \{2^{|cs|-i} \mid s \mapsto i\}$. So $\langle (s,t)\rangle^{(\pi\pi)}_{cs}$ is the number with bitstring $b_0 \dots b_{P(|cs|)-1}$ where $b_i = 1$ iff $s \mapsto i$, iff $t \not\mapsto i$ (with b_0 the most significant digit).
- $\Sigma^{(\pi\pi)}$ consists of the defined symbols introduced in $R^{(\pi\pi)}$, which we construct below.

The rules for the module closely follow those in Lemma 32, except that:

- calls to $\mathtt{seed}^1_{\mathtt{lin}}$, $\mathtt{zero}_{\mathtt{lin}}$ and $\mathtt{pred}^1_{\mathtt{lin}}$ are replaced by $\mathtt{seed}_\pi$, $\mathtt{zero}_\pi$ and $\mathtt{pred}_\pi$ respectively, where these symbols are supported by rules such as $\mathtt{zero}_\pi\ cs\ (s,t) \to \mathtt{zero}_\pi\ cs\ s\ t$ and $\mathtt{pred}_\pi\ cs\ (s,t) \to (\mathtt{pred}^1_\pi\ cs\ s\ t, \mathtt{pred}^2_\pi\ cs\ s\ t)$;
- calls $\mathtt{eqLen}\ n\ q$ are replaced by $\mathtt{eqBase}\ cs\ n\ q$, and the rules for $\mathtt{eqLen}$ replaced by $\mathtt{eqBase}\ cs\ (n_1,n_2)\ (m_1,m_2) \to \mathtt{equal}_\pi\ cs\ n_1\ n_2\ m_1\ m_2$. Just like a call to $\mathtt{eqLen}\ n\ q$ forces a reduction from q to a data term, a call to $\mathtt{eqBase}\ cs\ n\ q$ forces q to be reduced to a pair—but not necessarily to normal form.

With these rules, indeed $\mathtt{seed}^1_{(\pi\pi)}\ cs$ is in $\mathcal{A}^{(\pi\pi)}_{cs}$, as is $\mathtt{pred}^1_{(\pi\pi)}\ cs\ n$ if $n \in \mathcal{A}^\pi_{cs}$. Moreover, we can check that the requirements on reduction are satisfied. □

Thus, by starting with $C_{\mathtt{e}}$ and repeatedly using Lemma 49, we can reach arbitrarily high exponential bounds (since $2^{2^n-1} \ge 2^n$). Following the reasoning of § 4, we thus have:

Corollary 50. *Every set in ELEMENTARY is accepted by a cons-free first-order APTRS.*

The key reason for this explosion in expressivity is that, by matching on a pattern (x,y), a rule forces a *partial evaluation*. Recall that, in a cons-free ATRS (without pairing), we can limit interest to *semi-outermost* reductions, where sub-reductions $\mathtt{f}\ s_1 \cdots s_n \to^*_{\mathcal{R}} \mathtt{f}\ u_1 \cdots u_n = \ell\gamma \to_{\mathcal{R}} r\gamma$ have $s_i = u_i$ or $u_i \in \mathcal{DA}$ for all i: we can postpone an evaluation at an argument position if it is not to a data term. By allowing a wider range of terms than just the elements of $\mathcal{B}$ to carry testable information, expressivity increases accordingly.

We strongly conjecture that it is not possible to accept sets *not* in ELEMENTARY, however. A proof might use a variation of Algorithm 35, where $[\![\sigma \times \tau]\!]_{\mathcal{B}} = \{(A,B) \mid A \in [\![\sigma]\!]_{\mathcal{B}} \wedge B \in [\![\tau]\!]_{\mathcal{B}}\}$: the size of this set is exponential in the sizes of $[\![\sigma]\!]_{\mathcal{B}}$ and $[\![\tau]\!]_{\mathcal{B}}$, leading to a limit of the form $\exp_2^{a \cdot n^b}$ depending on the types used. However, we do not have the space to prove this properly, and the result does not seem interesting enough to warrant the effort.

Yet, product types *are* potentially useful. We can retain them while suitably constraining expressivity, by imposing a new restriction.

Definition 51. An APTRS is *product-cons-free* if it is cons-free and for all rules $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ and subterms $r \trianglerighteq (r_1, r_2)$: each r_i has a form (a) (s,t), (b) $\mathtt{c}\ s_1 \cdots s_n$ with $\mathtt{c} \in \mathcal{C}$, or (c) $x \in \mathcal{V}$ such that $x \neq \ell_j$ for any j (so x occurs below a constructor or pair on the left).

In a product-cons-free APTRS, any pair which is created is necessarily a data term. Lemma 49 does not go through in a product-cons-free APTRS (due to the rules for $\mathtt{pred}_\pi$ and $\mathtt{seed}_\pi$), but we *do* obtain a milder increase in expressivity: from $\mathrm{E}^K\mathrm{TIME}$ to $\mathrm{EXP}^K\mathrm{TIME}$.

Lemma 52. *Suppose counting modules are defined over product-cons-free APTRSs. Then for all $a \geq 0, b > 0$, there is a first-order $(\lambda n.2^{a\cdot(n+1)^b})$-counting module $C_{\mathtt{exp}\langle a,b\rangle}$.*

Proof. As in $C_{\mathtt{e}}$ from Lemma 32, we will represent a number with bitstring $b_0 \dots b_N$ by two terms yss and zss, such that $yss \mapsto i$ iff $b_i = \mathtt{1}$ and $zss \mapsto i$ iff $b_i = \mathtt{0}$. However, where in Lemma 32 we say $s \mapsto i$ if s reduces to a data term list of length i, here we say $s \mapsto i$ if s reduces to a data term of type $\mathtt{list}^{b+1}$ which represents i as in Lemma 29.

Formally: Write $|xs|$ for the length of a data term list xs, so the number of ; symbols occurring in it. Let *Base* be the set of all data terms $(u_0, \dots, u_b) : \mathtt{list}^{b+1}$ such that (a) $|u_0| < a$ and (b) for $0 < i \leq b$: $|u_i| \leq |cs|$. We say that $(u_0, \dots, u_b) \in \mathit{Base}$ *base-represents* $k \in \mathbb{N}$ if $k = \sum_{i=0}^{b} |u_i| \cdot (|cs|+1)^{b-i}$. (This follows the same idea as Lemma 29.) For a term s, we say $s \mapsto k$ if s reduces by $\to_R$ to an element of *Base* which base-represents k.

Now let $C_{\mathtt{exp}\langle a,b\rangle} := (\mathtt{list}^{b+1}, \Sigma^{\mathtt{exp}\langle a,b\rangle}, R^{\mathtt{exp}\langle a,b\rangle}, \mathcal{A}^{\mathtt{exp}\langle a,b\rangle}, \langle\cdot\rangle^{\mathtt{exp}\langle a,b\rangle})$, where:

- $\mathcal{A}_{cs}^{\mathtt{exp}\langle a,b\rangle}$ contains all (yss, zss) such that (a) all normal forms of yss or zss are in *Base*, and (b) for all $0 \leq i < a \cdot (|cs|+1)^b$: either $yss \mapsto i$ or $zss \mapsto i$, but not both.
- $\langle (s,t) \rangle_{cs}^{\mathtt{exp}\langle a,b\rangle} = \sum_{i=0}^{N} \{2^{N-i} \mid s \mapsto i\}$, where $N = a \cdot (|cs|+1)^b - 1$.
- $\Sigma^{\mathtt{exp}\langle a,b\rangle}$ consists of the defined symbols introduced in $R^{\mathtt{exp}\langle a,b\rangle}$, which are those in Lemma 32 with $\mathtt{seed}^1_{\mathtt{lin}}, \mathtt{zero}_{\mathtt{lin}}, \mathtt{pred}^1_{\mathtt{lin}}$ and $\mathtt{eqLen}$ replaced by $\mathtt{seed}_{\mathtt{base}}, \mathtt{zero}_{\mathtt{base}}$, $\mathtt{pred}_{\mathtt{base}}$ and $\mathtt{eqBase}$ respectively, along with the following supporting rules:

$$\begin{aligned}
\mathtt{seed}_{\mathtt{base}}\ cs &\to (\mathtt{0};\dots;\mathtt{0};[], cs, \dots, cs) \\
&\qquad [\![\text{with } |\mathtt{0};\dots;\mathtt{0};[]| = a-1]\!] \\
\mathtt{zero}_{\mathtt{base}}\ cs\ ([], \dots, []) &\to \mathtt{true} \\
\mathtt{zero}_{\mathtt{base}}\ cs\ (xs_0, \dots, xs_{i-1}, y;ys, [], \dots, []) &\to \mathtt{false} \qquad [\![\text{for } 0 \leq i \leq b]\!] \\
\mathtt{pred}_{\mathtt{base}}\ cs\ ([], \dots, []) &\to ([], \dots, []) \\
\mathtt{pred}_{\mathtt{base}}\ (c;zs)\ (xs_0, \dots, xs_{i-1}, y;ys, [], \dots, []) &\to (xs_0, \dots, xs_{i-1}, ys, c;zs, \dots, c;zs) \\
&\qquad [\![\text{for } 0 \leq i \leq b]\!]
\end{aligned}$$

$$\begin{aligned}
\mathtt{eqBase}\ ([], \dots, [])\ ([], \dots, []) &\to \mathtt{true} \\
\mathtt{eqBase}\ (xs_0, \dots, xs_{i-1}, y;ys, [], \dots, [])\ (zs_0, \dots, zs_{i-1}, [], [], \dots, []) &\to \mathtt{false} \\
\mathtt{eqBase}\ (xs_0, \dots, xs_{i-1}, [], [], \dots, [])\ (zs_0, \dots, zs_{i-1}, y;ys, [], \dots, []) &\to \mathtt{false} \\
\mathtt{eqBase}\ (xs_0, \dots, xs_{i-1}, y;ys, [], \dots, [])\ (zs_0, \dots, zs_{i-1}, n;ns, [], \dots, []) &\to \\
\mathtt{eqBase}\ (xs_0, \dots, xs_{i-1}, ys, [], \dots, [])\ (zs_0, \dots, zs_{i-1}, ns, [], \dots, [])&
\end{aligned}$$

This module functions as the ones from Lemmas 32 and 49. Note that in the rules for $\mathtt{pred}_{\mathtt{base}}$, we expanded the variable cs representing the input list to keep these rules product-cons-free. By using $c;zs$, the list is guaranteed to be normalized (and non-empty). □

Thus, combining Lemmas 52 and 30 with the rules of Figure 2, we obtain:

Corollary 53. *Any decision problem in EXP^K TIME is accepted by a K^{th}-order product-cons-free APTRS.*

By standard results, $\mathrm{E}^K\mathrm{TIME} \subsetneq \mathrm{EXP}^K\mathrm{TIME}$ for all $K \geq 1$, hence the addition of pairing materially increases expressivity. Conversely, we have:

Theorem 54. *Any set accepted by a K^{th}-order product-cons-free APTRS is in EXP^K TIME.*

Proof. Following the proof of Lemma 37, the complexity of Algorithm 35 is polynomial in the cardinality of the largest $[\![\sigma]\!]_{\mathcal{B}}$ used. The result follows by letting $[\![\iota_1 \times \cdots \times \iota_n]\!]_{\mathcal{B}}$ contain subsets of $\mathcal{B}^n$—which we can do because the only pairs occurring in a reduction are data.

Formally, let $(\mathcal{F}, \mathcal{R})$ be a product-cons-free APTRS. We first prove that any pair occurring in a reduction $s \to_{\mathcal{R}}^* t$ with s basic, is a data term. Let a term s be *product-$\mathcal{B}$-safe* if s is $\mathcal{B}$-safe and $s \trianglerighteq (s_1, s_2)$ implies $(s_1, s_2) \in \mathcal{DA}$ for all s_1, s_2. We observe: *(**) if s is product-$\mathcal{B}$-safe and $s \to_{\mathcal{R}} t$, then t is product-$\mathcal{B}$-safe.* $\mathcal{B}$-safety of t follows by Lemma 21 and for any $t \trianglerighteq (t_1, t_2)$: if not $s \trianglerighteq (t_1, t_2)$, then there are $\ell \to r \in \mathcal{R}$, substitution γ and r_1, r_2 such that $s \trianglerighteq \ell\gamma$ and $r \trianglerighteq (r_1, r_2)$ and $(t_1, t_2) = (r_1, r_2)\gamma$. By definition of product-cons-free, each r_i is a pair—so $r_i\gamma$ is data by induction on the size of r—or has the form $\mathtt{c}\ \vec{s}$—so $r_i\gamma$ is a data term by $\mathcal{B}$-safety of t—or is a variable x such that $\gamma(x) \in \mathcal{DA}$ by product-$\mathcal{B}$-safety of s. Thus, (r_1, r_2) is a pair of two data terms, and therefore data itself.

Thus, we can safely assume that the only product types that occur have type order 0, and remove constructors or defined symbols using higher order product types.

Next, we adapt Algorithm 35. We denote all types of order 0 as $\iota_1 \times \cdots \times \iota_n$ (ignoring bracketing) and let $[\![\iota_1 \times \cdots \times \iota_n]\!]_{\mathcal{B}} = \mathbb{P}(\{(s_1, \ldots, s_n) \mid s_i \in \mathcal{B} \wedge \vdash s_i : \iota_i \text{ for all } i\})$. Otherwise, the algorithm is unaltered. Let b be the longest length of any product type occurring in $\mathcal{F}$. As $\mathbb{P}(\mathcal{B}^b)$ has cardinality 2^{N^b}, the reasoning in Lemma 37 gives $\mathtt{card}([\![\sigma]\!]_{\mathcal{B}}) \leq \exp_2^{K+1}(d^K \cdot N^b)$ for a type of order k, which results in TIME $\left(\exp_2^K(a \cdot n^b)\right)$ for the algorithm. Lemma 41 goes through unmodified, Lemma 42 goes through if we define (s, t) to be computable if both s and t are, and Lemma 38 by using product-$\mathcal{B}$-safety instead of $\mathcal{B}$-safety in case (B). □

Thus we obtain:

Corollary 55. *A decision problem X is in EXP^K TIME if and only if there is a K^{th}-order product-cons-free APTRS which accepts X.*

7. Altering ATRSs

As demonstrated in § 6, the expressivity of cons-free term rewriting is highly sensitive in the presence of non-determinism: minor syntactical changes have the potential to significantly affect expressivity. In this section, we briefly discuss three other groups of changes.

7.1. **Strategy.** In moving from functional programs to term rewriting, we diverge from Jones' work in two major ways: by allowing non-deterministic choice, and by not imposing a reduction strategy. Jones' language in [15] employs *call-by-value* reduction. A close parallel in term rewriting is to consider *innermost* reductions, where a step $\ell\gamma \to_{\mathcal{R}} r\gamma$ may only be taken if all strict subterms of $\ell\gamma$ are in normal form. Based on results by Jones and Bonfante, and our own work on call-by-value programs, we conjecture the following claims:

(1) *confluent* cons-free ATRSs of order K, with innermost reduction, characterize $\mathrm{EXP}^{K-1}\mathrm{TIME}$; here, $\mathrm{EXP}^0\mathrm{TIME} = \mathrm{P}$, the sets decidable in polynomial time
(2) cons-free ATRSs of order 1, with innermost reduction, characterize P
(3) cons-free ATRSs of order > 1, with innermost reduction, characterize ELEMENTARY

(1) is a direct translation of Jones' result on time complexity from [15] to innermost rewriting. (2) translates Bonfante's result [10], which states that adding a non-deterministic choice operator to Jones' language does not increase expressivity in the first-order case. (3) is our own result, presented (again for call-by-value programs) in [17]. The reason for the explosion is that we can define a similar counting module as the one for pairing in Lemma 49.

Each result can be proved with an argument similar to the one in this paper: for one direction, a TM simulation with counting modules; for the other, an algorithm to evaluate the cons-free program. While the original results admit pairing, this adds no expressivity as the simulations can be specified without pairs. We believe that the proof is easily changed to accommodate innermost over call-by-value reduction, but have not done this formally.

Alternatively, we may consider *outermost* reductions steps, where rules are always applied at the highest possible position in a term. Outermost reductions are semi-outermost, but may behave differently in the presence of overlapping rules; for example, given rules $\mathtt{f}\ \mathtt{0} \to \mathtt{true}$ and $\mathtt{f}\ x \to \mathtt{false}$, an outermost evaluation would have to reduce $\mathtt{f}\ (\mathtt{0} + \mathtt{0})$ to $\mathtt{false}$, while in a semi-outermost evaluation we could also have $\mathtt{f}\ (\mathtt{0} + \mathtt{0}) \to_{\mathcal{R}} \mathtt{f}\ \mathtt{0} \to_{\mathcal{R}} \mathtt{true}$. We note that the ATRS from Figure 2 and all counting modules evaluate as expected using outermost reduction and that Theorem 45 does not consider evaluation strategy. This gives:

Corollary 56. *A decision problem X is in E^K TIME if and only if there is a K^{th}-order cons-free ATRS with outermost reduction which accepts X.*

7.2. **Constructor ATRSs and left-linearity.** Recall that we have exclusively considered *left-linear constructor ATRSs*. One may wonder whether these restrictions can be dropped.

The answer, however, is no. In the case of constructor ATRSs, this is easy to see: if we do not limit interest to constructor ATRSs—so if, in a rule $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ the terms ℓ_i are not required to be patterns—then "cons-free" becomes meaningless, as we could simply let $\mathcal{D} := \mathcal{F}$. Thus, we would obtain a Turing-complete language already for first-order ATRSs.

Removing the requirement of left-linearity similarly provides full Turing-completeness. This is demonstrated by the first-order cons-free ATRS in Figure 5 which simulates an arbitrary TM on input alphabet $I = \{0, 1\}$. A tape $x_0 \dots x_n \sqcup \sqcup \dots$ with the reading head at position i is represented by three parameters: $x_{i-1}{::}\dots{::}x_0$ and x_i and $x_{i+1}{::}\dots{::}x_n$. Here, the "list constructor" :: is a *defined symbol*, ensured by a rule which never fires. To split a "list" into a head and tail, the ATRS non-deterministically generates a *new* head and tail using two calls to $\mathtt{rndtape}$ (whose only shared reducts are fully evaluated "lists"), and uses a non-left-linear rule to compare their combination to the original "list".

7.3. **Variable binders.** A feature present in many styles of higher-order term rewriting is λ-*abstraction*; e.g., a construction such as $\lambda x.\mathtt{f}\ x$. Depending on the implementation, admitting λ-abstraction in cons-free ATRSs may blow up expressivity, or not affect it at all.

First, consider ATRSs with λ-abstractions used only in the right-hand sides of rules. Then all abstractions can be removed by introducing fresh function symbols, e.g., by replacing a rule $\mathtt{f}\ (\mathtt{c}\ y) \to \mathtt{g}\ (\lambda x.\mathtt{h}\ x\ y)$ by the two rules $\mathtt{f}\ (\mathtt{c}\ y) \to \mathtt{g}\ (\mathtt{f}_{\mathtt{help}}\ y)$ and $\mathtt{f}_{\mathtt{help}}\ y\ x \to \mathtt{h}\ x\ y$ (where $\mathtt{f}_{\mathtt{help}}$ is a fresh symbol). Since the normal forms of basic terms are not affected by this change, this feature adds no expressivity.

Second, some variations of higher-order term rewriting require that function symbols are always assigned to as many arguments as possible; abstractions are the *only* terms of

$$
\begin{array}{rcl@{\qquad}rcl}
\texttt{rndtape}\ x & \to & [] & \texttt{rnd} & \to & \texttt{0} \\
\texttt{rndtape}\ x & \to & \texttt{rnd} :: \texttt{rndtape}\ x & \texttt{rnd} & \to & \texttt{1} \\
\bot :: t & \to & t & \texttt{rnd} & \to & \texttt{B}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{translate}\ (\texttt{0};xs) & \to & \texttt{0} :: (\texttt{translate}\ xs) \\
\texttt{translate}\ (\texttt{1};xs) & \to & \texttt{1} :: (\texttt{translate}\ xs) \\
\texttt{translate}\ [] & \to & \texttt{B} :: (\texttt{translate}\ []) \\
\texttt{translate}\ [] & \to & [] \\
\texttt{equal}\ xl\ xl & \to & \texttt{true}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{start}\ cs & \to & \texttt{run start}\ []\ \texttt{B}\ (\texttt{translate}\ cs) \\
\texttt{run}\ \underline{\texttt{s}}\ xl\ \underline{\texttt{r}}\ yl & \to & \texttt{shift}\ \underline{\texttt{t}}\ xl\ \underline{\texttt{w}}\ yl\ \underline{\texttt{d}} \quad [\![\text{for every transition } \underline{\texttt{s}} \xrightarrow{\underline{\texttt{r}}/\underline{\texttt{w}}\ \underline{\texttt{d}}} \underline{\texttt{t}}]\!] \\
\texttt{shift}\ s\ xl\ c\ yl\ d & \to & \texttt{shift}_1\ s\ xl\ c\ yl\ d\ \texttt{rnd}\ (\texttt{rndtape 0})\ (\texttt{rndtape 1}) \\
\texttt{shift}_1\ s\ xl\ c\ yl\ d\ \underline{\texttt{b}}\ t\ t & \to & \texttt{shift}_2\ s\ xl\ c\ yl\ d\ \underline{\texttt{b}}\ t \quad [\![\text{for every } \underline{\texttt{b}} \in \{\texttt{0},\texttt{I},\texttt{B}\}]\!] \\
\texttt{shift}_2\ s\ xl\ c\ yl\ \texttt{R}\ z\ t & \to & \texttt{shift}_3\ s\ (c :: xl)\ z\ t\ (\texttt{equal}\ yl\ (z :: t)) \\
\texttt{shift}_2\ s\ xl\ c\ yl\ \texttt{L}\ z\ t & \to & \texttt{shift}_3\ s\ t\ z\ (c :: yl)\ (\texttt{equal}\ xl\ (z :: t)) \\
\texttt{shift}_3\ s\ xl\ c\ yl\ \texttt{true} & \to & \texttt{run}\ s\ xl\ c\ yl
\end{array}
$$

Figure 5: A first-order non-left-linear ATRS that simulates a given Turing machine

functional type. Clearly, this does not increase expressivity as it merely limits the number of programs (with λ-abstraction) that we can specify. Nor does it lower expressivity: the results in this paper go through in such a formalism, as demonstrated in [16]. It does, however, require some changes to the definition of a counting module.

Finally, if abstractions are allowed in the *left*-hand sides of rules, then the same problem arises as in Lemma 49: we can force a partial evaluation, and use this to define $(\lambda n.\, \exp_2^K(n))$-counting modules for arbitrarily high K without increasing type orders. This is because a rule such as $\texttt{f}\ (\lambda x.Z)$ matches a term $\texttt{f}\ (\lambda x.\texttt{0})$, but does *not* match $\texttt{f}\ (\lambda x.\texttt{g}\ x\ \texttt{0})$ because of how substitution works in the presence of binders. A full exposition of this issue would require a more complete definition of higher-order term rewriting with λ-abstraction, so is left as an exercise to interested readers. A restriction such as *fully extended rules* may be used to bypass this issue; we leave this question to future work.

8. Conclusions

We have studied the expressive power of cons-free higher-order term rewriting, and seen that restricting data order results in characterizations of different classes. We have shown that pairing dramatically increases this expressive power—and how this can be avoided by using additional restrictions—and we have briefly discussed the effect of other syntactical changes. The main results are displayed in Figure 6.

P	**C**	
confluent cons-free ATRSs with call-by-value reduction	$\text{EXP}^{K-1}\text{TIME}$	(translated from [15])
cons-free ATRSs	E^KTIME	(Corollary 46)
product-cons-free APTRSs	EXP^KTIME	(Corollary 56)
cons-free APTRSs (so with pairing)	$\geq$ ELEMENTARY	(Corollary 50)

Figure 6: Overview: systems **P** with type order K characterize the class **C**.

8.1. **Future work.** We see two major, natural lines of further inquiry, that we believe will also be of significant interest in the general—non-rewriting related—area of implicit complexity. Namely (I), the imposition of further restrictions, either on rule formation, reduction strategy or both that, combined with higher-order rewriting will yield characterization of *non*-deterministic classes such as NP, or of sub-linear time classes like LOGTIME. And (II), additions of *output*. While cons-freeness does not naturally lend itself to producing output, it is common in implicit complexity to investigate characterizations of sets of computable *functions*, e.g. the polytime-computable functions on integers, rather than decidable sets. This could for instance be done by allowing the production of constructors of specific types.

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.

[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.

[3] M. Avanzini and G. Moser. Polynomial path orders. *Logical Methods in Computer Science*, 9(4), 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.

[5] P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *CSL*, volume 16 of *LIPIcs*, pages 62–76, 2012.

[6] Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *ESOP*, volume 6012 of *LNCS*, pages 104–124, 2010.

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

[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.

[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.

[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] Lauri Hella and Jos Mara Turull-Torres. Computing queries with higher-order logics. *Theoretical Computer Science*, 355(2):197 – 214, 2006.

[13] M. Hofmann. Type systems for polynomial-time computation, 1999. Habilitationsschrift.

[14] N. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.

[15] N. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.

[16] C. Kop and J. Simonsen. Complexity hierarchies and higher-order cons-free rewriting. In *FSCD*, volume 52 of *LIPIcs*, pages 23:1–23:18, 2016.

[17] C. Kop and J. Simonsen. The power of non-determinism in higher-order implicit complexity. In *ESOP*, volume 10201 of *LNCS*, pages 668–695, 2017.

[18] L. Kristiansen and K. Niggl. On the computational complexity of imperative programming languages. *Theoretical Computer Science*, 318(1–2):139–161, 2004.

[19] Gabriel M. Kuper and Moshe Y. Vardi. On the complexity of queries in the logical data model. *Theoretical Computer Science*, 116(1):33 – 57, 1993.

[20] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[21] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.

[22] F. van Raamsdonk. Higher-order rewriting. In *Term Rewriting Systems*, Chapter 11, pages 588–667. Cambridge University Press, 2003.