Towards regular languages over infinite alphabets

Frank Neven[1]*, Thomas Schwentick[2], and Victor Vianu[3]**

[1] Limburgs Universitair Centrum
[2] Johannes Gutenberg-Universität Mainz
[3] U.C. San Diego

Abstract. Motivated by formal models recently proposed in the context of XML, we study automata and logics on strings over infinite alphabets. These are conservative extensions of classical automata and logics defining the regular languages on finite alphabets. Specifically, we consider register and pebble automata, and extensions of first-order logic and monadic second-order logic. For each type of automaton we consider one-way and two-way variants, as well as deterministic, non-deterministic, and alternating control. We investigate the expressiveness and complexity of the automata, their connection to the logics, as well as standard decision problems. Some of our results answer open questions of Kaminski and Francez on register automata.

1 Introduction

One of the significant recent developments related to the World Wide Web (WWW) is the emergence of the Extensible Markup Language (XML) as the standard for data exchange on the Web [1]. Since XML documents have a tree structure (usually defined by DTDs), XML queries can be modeled as mappings from trees to trees (tree transductions), and schema languages are closely related to tree automata, automata theory has naturally emerged as a central tool in formal work on XML [5, 16–22]. The connection to logic and automata proved very fruitful in understanding such languages and in the development of optimization algorithms and static analysis techniques. However, these abstractions ignore an important aspect of XML, namely the presence of *data values* attached to leaves of trees, and comparison tests performed on them by XML queries. These data values make a big difference – indeed, in some cases the difference between decidability and undecidability (e.g., see [4]). It is therefore important to extend the automata and logic formalisms to trees with data values. In this initial investigation we model data values by infinite alphabets, and consider the simpler case of strings rather than trees. Strings are also relevant in the tree case, as most formalisms allow reasoning along paths in the tree. In the case of XML, it would be more accurate to consider strings labeled by a finite alphabet and attach data values to positions in the string. However, this would render the formalism more complicated and has no bearing on the results. Although limited to strings, we believe that our results provide a useful starting point in investigating the problem. In particular, our lower-bound results will easily be extended to trees.

We only consider models which accept precisely the regular languages when restricted to finite alphabets. It is useful to observe that for infinite alphabets it is no longer sufficient to equip automata with states alone. Indeed, automata should at least be able to check equality of symbols. There are two main ways to do this: (1) store a finite set of positions and allow equality tests between the symbols on these positions; (2) store a finite set of symbols only and allow equality tests with these symbols. The first approach, however, leads to multi-head automata, immediately going beyond regular languages. Therefore, we instead equip automata with a finite set of *pebbles* whose use is restricted by a stack discipline. The

* Post-doctoral researcher of the Fund for Scientific Research, Flanders.
** This author supported in part by the National Science Foundation under grant number IIS-9802288.

automaton can test equality by comparing the pebbled symbols. In the second approach, we follow Kaminski and Francez [14, 13] and extend finite automata with a finite number of *registers* that can store alphabet symbols. When processing a string, an automaton compares the symbol on the current position with values in the registers; based on this comparison it can decide to store the current symbol in some register. In addition to automata, we consider another well-known formalism: monadic second-order logic (MSO). To be precise, we associate to strings first-order structures in the standard way, and consider the extensions of MSO and FO denoted by MSO^* and FO^*, as done by Grädel and Gurevich in the context of meta-finite models [8]. MSO has proven to be a good yardstick when other generalizations of regular languages were investigated, e.g., for trees, infinite strings [24] and graphs [6].

Our results concern the expressive power of the various models, provide lower and upper complexity bounds, and consider standard decision problems. For the above mentioned automata models we consider deterministic (D), non-deterministic (N), and alternating (A) control, as well as one-way and two-way variants. We denote these automata models by dC-X where $d \in \{1, 2\}$, $C = \{\mathrm{D}, \mathrm{N}, \mathrm{A}\}$, and $X \in \{\mathrm{RA}, \mathrm{PA}\}$. Here, 1 and 2 stand for one- and two-way, respectively, D, N, and A stand for deterministic, non-deterministic, and alternating, and PA and RA for pebble and register automata. Our main results are the following (results on expressiveness are graphically presented in Figure 1).

Registers. We pursue the investigation of register automata initiated by Kaminski and Francez [14]. In particular, we investigate the connection between RAs and logic and show that they are essentially incomparable. Indeed, we show that MSO^* cannot define 2D-RA. Furthermore, there are even properties in FO^* that cannot be expressed by 2A-RAs. The proof of the latter is a non-trivial argument based on communication complexity [12]. Next, we consider the relationship between the various RA models. We separate 1N-RAs, 2D-RAs, 2N-RAs, and 2A-RAs, subject to standard complexity-theoretic assumptions.

Pebbles. We consider two kinds of PAs: one where every new pebble is placed on the first position of the string and one where every new pebble is placed on the position of the current pebble. We refer to them as strong and weak PAs, respectively. Clearly, this pebble placement only makes a difference in the case of one-way PAs. In the one-way case, strong 1D-PA can simulate FO^* while weak 1N-PA cannot (whence the names). The proof of the latter separation is again based on communication complexity. Furthermore, we show that all pebble automata variants can be defined in MSO^*. Finally, we provide more evidence that strong PAs are a robust notion by showing that the power of strong 1D-PA, strong 1N-PA, 2D-PA, and 2N-PA coincide.

Decision Problems. Finally, we consider decision problems for RAs and PAs, and answer several open questions from Kaminsky and Francez. First, we show that universality and containment of 1N-RAs and non-emptiness of 2D-RA are undecidable. Next, we obtain that non-emptiness even for weak 1D-PAs is undecidable.

As RAs are orthogonal to logically defined classes one might argue that PAs are better suited to define the notion of regular languages over infinite alphabets. Indeed, they are reasonably expressive as they lie between FO^* and MSO^*. Furthermore, strong PAs form a robust notion. Adding two-wayness and nondeterminism does not increase expressiveness and the class of languages is defined under Boolean operations, concatenation and Kleene star. Capturing exactly MSO^* most likely requires significant extensions of PAs, as in MSO^* one can express complete problems for every level of the polynomial hierarchy, while computations of 2A-RAs are in P.

Related work. Kaminski and Francez were the first to consider RAs (which they called *finite-memory automata*) to handle strings over infinite alphabets. They showed that 1N-RAs are closed under union, intersection, concatenation, and Kleene star. They further showed that

non-emptiness is decidable for 1N-RAs and that containment is decidable for 1N-RAs when the automaton in which containment has to be tested has only two registers.

When the input is restricted to a finite alphabet, PAs recognize the regular languages, even in the presence of alternation [17]. We point out that the pebbling mechanism we employ is based on the one of Milo, Suciu, and Vianu [17] and is more liberal than the one used by Globerman and Harel [9]: indeed, in our case, after a pebble is placed the automaton can still walk over the whole string and sense the presence of the other pebbles. Globerman and Harel prove certain lower bounds in the gap of succinctness of the expressibility of their automata.

Overview. This paper is organized as follows. In Section 2, we provide the formal framework. In Section 3, we study register automata. In Section 4, we examine pebble automata. In Section 5, we compare the register and pebble models. In Section 6, we discuss decision problems. We conclude with a discussion in Section 7. Due to space limitations proofs are only sketched. Full proofs can be found in the Appendix.

2 Definitions

We consider strings over an infinite alphabet $\mathbf{D}$. Formally, a $\mathbf{D}$-*string* w is a finite sequence $d_1 \cdots d_n \in \mathbf{D}^*$. As we are often dealing with 2-way automata we delimit input strings by two special symbols, $\triangleright, \triangleleft$ for the left and the right end of the string, both of which are not in $\mathbf{D}$. I.e., automata always work on strings of the form $w = \triangleright v \triangleleft$, where $v \in \mathbf{D}^*$. By $\mathrm{dom}(w)$ we denote the set $\{1, \ldots, |w|\}$ with $|w|$ the length of w. For $i \in \mathrm{dom}(w)$, we also write $\mathrm{val}_w(i)$ for d_i.

2.1 Register automata

The following definition is the one from Kaminski and Francez [14, 13].

Definition 1. A *k-register automaton $\mathcal{B}$ over* $\mathbf{D}$ is a tuple (Q, q_0, F, τ_0, P) where

- Q is a finite set of states; $q_0 \in Q$ is the initial state; $F \subseteq Q$ is the set of final states;
- $\tau_0 : \{1, \ldots, k\} \to \mathbf{D} \cup \{\triangleright, \triangleleft\}$ is the initial register assignment; and,
- P is a finite set of transitions of the forms $(i, q) \to (q', d)$ or $q \to (q', i, d)$.
  Here, $i \in \{1, \ldots, k\}$, $q, q' \in Q$ and $d \in \{\mathrm{stay}, \mathrm{left}, \mathrm{right}\}$.

Given a string w, a *configuration of $\mathcal{B}$ on w* is a tuple $[j, q, \tau]$ where $j \in \mathrm{dom}(w)$, $q \in Q$, and $\tau : \{1, \ldots, k\} \to \mathbf{D} \cup \{\triangleright, \triangleleft\}$. The *initial* configuration is $\gamma_0 := [1, q_0, \tau_0]$. A configuration $[j, q, \tau]$ with $q \in F$ is *accepting*. Given $\gamma = [j, q, \tau]$, the transition $(i, p) \to \beta$ (respectively, $p \to \beta$) *applies* to γ iff $p = q$ and $\mathrm{val}_w(j) = \tau(i)$ (respectively, $\mathrm{val}_w(j) \neq \tau(i)$ for all $i \in \{1, \ldots, k\}$).

Given $\gamma = [j, q, \tau]$ and $\gamma' = [j', q', \tau']$, we define the one step transition relation $\vdash$ on configurations as follows: $\gamma \vdash \gamma'$ iff there is a transition $(i, q) \to (q', d)$ that applies to γ, $\tau' = \tau$, and $j' = j$, $j' = j - 1$, $j' = j + 1$ whenever $d = \mathrm{stay}$, $d = \mathrm{left}$, or $d = \mathrm{right}$, respectively; or there is a transition $q \to (q', i, d)$ that applies to γ, j' is as defined in the previous case, and τ' is obtained from τ by setting $\tau'(i)$ to $\mathrm{val}_w(j)$. We denote the transitive closure of $\vdash$ by $\vdash^*$. Intuitively, transitions $(i, q) \to (q', d)$ can only be applied when the value of the current position is in register i. Transitions $q \to (q', i, d)$ can only be applied when the value of the current position differs from all the values in the registers. In this case, the current value is copied into register i.

We require that the initial register assignment contains the symbols $\triangleright$ and $\triangleleft$, so automata can recognize the boundaries of the input. Furthermore, from a $\triangleright$ only right-transitions and from a $\triangleleft$ only left-transitions are allowed.

As usual, a string w is accepted by $\mathcal{B}$, if $\gamma_0 \vdash^* \gamma$, for some accepting configuration γ. The language $L(\mathcal{B})$ accepted by $\mathcal{B}$, is defined as $\{v \mid \triangleright v \triangleleft \text{ is accepted by } \mathcal{B}\}$.

The automata we defined so far are in general *non-deterministic*. An automaton is *deterministic*, if in each configuration at most one transition applies. If there are no left-transitions, then the automaton is *one-way*. Alternating automata will be defined below. As explained in the introduction, we refer to these automata as dC-RA where $d \in \{1, 2\}$ and $C = \{\mathrm{D}, \mathrm{N}, \mathrm{A}\}$. Clearly, when the input is restricted to a finite alphabet, RAs accept only regular languages.

2.2 Pebble automata

We borrow some notation from Milo, Suciu, and Vianu [17].

Definition 2. A *k-pebble automaton $\mathcal{A}$ over* $\mathbf{D}$ is a tuple (Q, q_0, F, T) where

- Q is a finite set of *states*; $q_0 \in Q$ is the *initial state*; $F \subseteq Q$ is the set of *final states*; and,
- T is a finite set of *transitions* of the form $\alpha \to \beta$, where α is of the form (i, s, P, V, q) or (i, P, V, q), where $i \in \{1, \ldots, k\}$, $s \in \mathbf{D} \cup \{\triangleright, \triangleleft\}$, $P, V \subseteq \{1, \ldots, i-1\}$, and β is of the form (q, d) with $q \in Q$ and $d \in \{\text{stay}, \text{left}, \text{right}, \text{place-new-pebble}, \text{lift-current-pebble}\}$.

Given a string w, a *configuration of $\mathcal{A}$ on w* is of the form $\gamma = [i, q, \theta]$ where $i \in \{1, \ldots, k\}$, $q \in Q$ and $\theta : \{1, \ldots, i\} \to \mathrm{dom}(w)$. We call θ a pebble assignment and i the *depth* of the configuration (and of the pebble assignment). Sometimes we denote the depth of a configuration γ (pebble assignment θ) by $\mathrm{depth}(\gamma)$ ($\mathrm{depth}(\theta)$). The *initial* configuration is $\gamma_0 := [1, q_0, \theta_0]$ where $\theta_0(1) = 1$. A configuration $[i, q, \theta]$ with $q \in F$ is *accepting*.

A transition $(i, s, P, V, p) \to \beta$ *applies to a configuration* $\gamma = [j, q, \theta]$, if

1. $i = j$, $p = q$,
2. $P = \{l < i \mid \mathrm{val}_w(\theta(l)) = \mathrm{val}_w(\theta(i))\}$,
3. $V = \{l < i \mid \theta(l) = \theta(i)\}$, and
4. $\mathrm{val}_w(\theta(i)) = s$.

A transition $(i, P, V, q) \to \beta$ *applies* to γ if (1)-(3) hold and no transition $(i', s', P', V', q') \to \beta$ applies to γ. Intuitively, $(i, s, P, V, p) \to \beta$ applies to a configuration, if i is the current number of placed pebbles, p is the current state, P is the set of pebbles that see the same symbol as the top pebble, V is the set of pebbles that are at the same position as the top pebble, and the current symbol seen by the top pebble is s.

We define the transition relation $\vdash$ as follows: $[i, q, \theta] \vdash [i', q', \theta']$ iff there is a transition $\alpha \to (p, d)$ that applies to γ such that $q' = p$ and $\theta'(j) = \theta(j)$, for all $j < i$, and

- if $d = \text{stay}$, then $i' = i$ and $\theta'(i) = \theta(i)$,
- if $d = \text{left}$, then $i' = i$ and $\theta'(i) = \theta(i) - 1$,
- if $d = \text{right}$, then $i' = i$ and $\theta'(i) = \theta(i) + 1$,
- if $d = \text{place-new-pebble}$, then $i' = i + 1$, $\theta'(i+1) = \theta'(i) = \theta(i)$,
- if $d = \text{lift-current-pebble}$ then $i' = i - 1$.

The definitions of the *accepted language*, *deterministic* and *one-way* are analogous to the case of register automata. We refer to these automata as dC-RA where d and C are as before.

In the above definition, new pebbles are placed at the position of the most recent pebble. An alternative would be to place new pebbles at the beginning of the string. While the choice makes no difference in the two-way case, it is significant in the one-way case. We refer to the model as defined above as *weak* pebble automata and to the latter as *strong* pebble automata. Strong pebble automata are formally defined by setting $\theta'(i+1) = 1$ (and keeping $\theta'(i) = \theta(i)$) in the *place-new-pebble* case of the definition of the transition relation.

2.3 Alternating Automata

For both automata models we also define an *alternating version*. Alternating automata $\mathcal{A}$ additionally have a set $U \subseteq Q$ of *universal states*. The sets from $Q - U$ are called *existential*. If $U = \emptyset$, then the automaton is *non-deterministic*.

A *run* of $\mathcal{A}$ on w is a tree where nodes are labeled with configurations as follows:

1. the root is labeled with the initial configuration;
2. every inner node labeled with an existential configuration γ has exactly one child γ' and $\gamma \vdash \gamma'$; and,
3. every inner node labeled with a universal configuration γ has exactly n children labeled with $\gamma_1, \ldots, \gamma_n$ and $\{\gamma_1, \ldots, \gamma_n\} = \{\gamma' \mid \gamma \vdash \gamma'\}$.

An *accepting* run is a run where every leaf node is labeled with a final configuration. The language accepted by $\mathcal{A}$, is defined as $L(\mathcal{A}) := \{w \mid \text{there is an accepting run of } \mathcal{A} \text{ on } \triangleright w \triangleleft\}$.

2.4 Logic

We consider first-order and monadic second-order logic over **D**-strings. The representation as well as the logics are special instances of the meta-finite structures and their logics as defined by Grädel and Gurevich [8]. A string w is represented by the logical structure with domain $\mathrm{dom}(w)$, the natural ordering $<$ on the domain, and a function $\mathrm{val} : \mathrm{dom}(w) \to \mathbf{D}$ instantiated by val_w. An atomic formula is of the form $x < y$, $\mathrm{val}(x) = \mathrm{val}(y)$, or $\mathrm{val}(x) = d$ for $d \in \mathbf{D}$, and has the obvious semantics. The logic FO^* is obtained by closing the atomic formulas under the boolean connectives and first-order quantification over $\mathrm{dom}(w)$. Hence, no quantification over **D** is allowed. The logic MSO^* is obtained by adding quantification over sets over $\mathrm{dom}(w)$; again, no quantification over **D** is allowed.

2.5 Complexity Classes over Infinite Alphabets

Some of our separating results are relative to complexity-theoretic assumptions. To this end, we assume the straightforward generalization of standard complexity classes (like LOGSPACE, NLOGSPACE, and PTIME) to the case of infinite alphabets that can be defined, e.g., by using multi-tape Turing machines which are able to compare and move symbols between the current head positions. It should be clear that the collapse of two of these generalized classes immediately implies the collapse of the respective finite alphabet classes.

3 Register Automata

We start by investigating RAs. In particular, we compare them with FO^* and MSO^*. Our main conclusion is that RAs are orthogonal to these logics as they cannot even express all FO^* properties but can express properties not definable in MSO^*. Further, we separate the variants of RAs subject to standard complexity-theoretic assumptions.

3.1 Expressiveness

Theorem 3. *MSO^* cannot define all 2D-RA.*

Proof. (sketch) Consider strings of the form $u\#v$ where $u, v \in (\mathbf{D} - \{\#\})^*$. Define N_u and N_v as the set of symbols occurring in u and v, respectively. Denote by n_u and n_v their cardinalities. We show that there is a 2D-RA $\mathcal{A}$ that accepts $u\#v$ iff $n_u = n_v$ while there is no such MSO^* sentence.

To show that a 2D-RA can check this property we introduce some notation. For a string w, denote by $\text{lmo}_w(d)$ the position in w of the left most occurrence of $d \in \mathbf{D}$. Suppose $N_u = \{a_1, \ldots, a_n\}$ and $N_v = \{b_1, \ldots, b_m\}$ where for every $i < j$, $\text{lmo}_u(a_i) < \text{lmo}_u(a_j)$ and $\text{lmo}_v(b_i) < \text{lmo}_v(b_j)$. Then the 2D-RA compares n_u with n_v by visiting $\text{lmo}_u(a_1)$, $\text{lmo}_v(b_1)$, $\text{lmo}_u(a_2)$, $\text{lmo}_v(b_2)$, ... in sequence. In this manner it can check whether $n = m$. More details are provided in the Appendix.

Asssume towards a contradiction that φ^* is an MSO* sentence such that $u\#v \models \varphi^*$ iff $n_u = n_v$. Let C be the set of $\mathbf{D}$-symbols mentioned in φ^*. We call a string $u\#v$ admissible iff $N_u \cap N_v = \emptyset$; each $\mathbf{D}$-symbol occurs at most once in u or v; and, no symbol in C occurs in u or v. Let φ be obtained from φ^* by replacing each occurrence of $\text{val}(x) = \text{val}(y)$ by $x = y$, and every occurence of $\text{val}(x) = d$ by *false*, if $d \neq \#$. Then for every admissible string $d_1 \cdots d_n \# e_1 \cdots e_m$, $a^n\#b^m \models \varphi$ iff the string satisfies φ^* iff $n = m$. Hence, $\{a^n\#b^n \mid n \in \mathbb{N}\}$ would be MSO definable and therefore regular, the desired contradiction. □

We next show that RAs cannot capture FO*, even with alternation. The proof is based on communication complexity.

Theorem 4. *2A-RA cannot express FO*.*

Proof. We start with some terminology. Let D be a finite or infinite set. A *1-hyperset over D* is a finite subset of D. For $i > 1$, an *i-hyperset over D* is a finite set of $(i-1)$-hypersets over D. For clarity, we will often denote i-hypersets with a superscript i, as in $S^{(i)}$.

Let us assume that $\mathbf{D}$ contains all natural numbers and let, for $j > 0$, $\mathbf{D}_j$ be $\mathbf{D} - \{1, \ldots, j\}$. Let $j > 0$ be fixed. We inductively define *encodings* of i-hypersets over $\mathbf{D}_j$. A string $w = 1d_1d_2 \cdots d_n1$ over $\mathbf{D}_j$ is an encoding of the 1-hyperset $H(w) = \{d_1, \ldots, d_n\}$ over $\mathbf{D}_j$. For each $i \leq j$, and encodings $w_1, \ldots, w_n$ of $(i-1)$-hypersets, $iw_1iw_2 \cdots iw_ni$ is an encoding of the i-hyperset $\{H(w_i) \mid i \leq n\}$. Define $L^m_=$ as the language

$$\{u\#v \mid u \text{ and } v \text{ are encodings of } m\text{-hypersets over } \mathbf{D}_m - \{\#\} \text{ and } H(u) = H(v)\}.$$

In the appendix we prove the following lemma.

Lemma 5. *For each m, $L^m_=$ is definable in FO*.*

Next, we show that no 2A-RA can recognize $L^m_=$ for $m > 4$. The underlying idea is that for large enough m, a 2A-RA simply cannot communicate enough information between the two sides of the input string to check whether $H(u)$ equals $H(v)$. Our proof is inspired by a proof of Abiteboul, Herr, and Van den Bussche [2]. To separate the temporal query languages ETL from TS-FO, they showed that every query in ETL on a special sort of databases can be evaluated by a communication protocol with a constant number of messages. This is not the case for TS-FO. To simulate 2A-RA, however, we need a more powerful protocol where the number of messages depends on the number of different data values in u and v. This protocol is defined next. First, define $\exp_0(n) := n$ and $\exp_i(n) := 2^{\exp_{i-1}(n)}$, for $i > 0$.

Definition 6. Let P be a binary predicate on i-hypersets over $\mathbf{D}$ and let $k, l \geq 0$. We say that P can be *computed by a (k,l)-communication protocol* between two parties (denoted by `I` and `II`) if there is a polynomial p such that for all i-hypersets $X^{(i)}$ and $Y^{(i)}$ over a finite set D there is a finite alphabet Δ of size at most $p(|D|)$ such that $P(X^{(i)}, Y^{(i)})$ can be computed as follows:

1. `I` gets $X^{(i)}$ and `II` gets $Y^{(i)}$; both know D and Δ;
2. `I` sends a message $a_1(D, X^{(i)})$ to `II` and `II` replies with a message $b_1 = b_1(D, Y^{(i)}, a_1)$ to `I`. Each message is a k-hyperset over Δ.
3. `I` sends a message $a_2 = a_2(D, X^{(i)}, b_1)$ to `II` and `II` replies with a message $b_2 = b_2(D, Y^{(i)}, a_2)$ to `I`.

4. After $\exp_l(p(|D|))$ rounds of message exchanges, both I and II have enough information to decide whether $P(X^{(i)}, Y^{(i)})$ holds. Formally, they apply a Boolean function $a_{r+1}(D, X^{(i)}, b_r)$ (for I) or $b_{r+1}(D, Y^{(i)}, a_r)$ (for II) that evaluates to true iff $P(X^{(i)}, Y^{(i)})$ holds.

So, formally, a protocol consists of the functions $a_1, \ldots, a_{r+1}, b_1, \ldots, b_{r+1}$. Note that the computing power of I and II can be completely arbitrary.

Lemma 7. *For $m > 4$, $L^m_=$ cannot be computed by a (2,2)-communication protocol.*

Proof. Suppose there is a protocol computing $L^m_=$. For every finite set D with d elements, the number of different possible messages is the number of 2-hypersets which is at most $\exp_2(p(d))$. Call a complete sequence of exchanged messages $a_1 b_1 a_2 b_2 \ldots$ a dialogue. Every dialogue has at most $\exp_2(p(d))$ rounds. Hence, there are at most $\exp_2(2d \cdot \exp_2(p(d)))$ different dialogues. However, the number of different m-hypersets over D is $\exp_m(d)$. Hence, for $m > 4$ and D large enough there are m-hypersets $X^{(m)} \neq Y^{(m)}$ such that the protocol gives the same dialogue for $P(X^{(m)}, X^{(m)})$ and $P(Y^{(m)}, Y^{(m)})$. But that means it also gives the same dialogue on $P(X^{(m)}, Y^{(m)})$ and $P(Y^{(m)}, X^{(m)})$. This leads to the desired contradiction. □

We refer to strings of the form $u\#v$, where u and v do not contain #, as *split strings*. A communication protocol computes on such strings by giving u to I and v to II.

Lemma 8. *On split strings, the language defined by a 2A-RA can be recognized by a (2,2)-communication protocol.*

Proof. (sketch) Let $\mathcal{B}$ be a 2A-RA working on split strings over $\mathbf{D}$. On an input string $w := u\#v$, $[e, q, \tau]$ is a #-configuration when e is the position of # in w. Define $p(n) := |Q|n^k$, where Q and k are the set of states and number of registers of $\mathcal{B}$, respectively. Then the number of #-configurations is $|Q|m^k$ where m is the number of different symbols in w. We assume w.l.o.g. that there are no transitions possible from final configurations. Further, we assume that $\mathcal{B}$ never changes direction or accepts at the symbol #. Hence, on w, when $\mathcal{B}$ leaves u to the right it enters v and vice versa. In essence, both parties compute partial runs where they send the #-configurations in which $\mathcal{B}$ walks off their part of the string to the other party. To be concrete, I computes all the runs of $\mathcal{B}$ on u the leaves of which consist of #-configurations or final configurations, and no inner vertex is labeled with a #-configuration. It then sends to II the set of all sets of #-configurations appearing at leaves of such runs. Party II in turn, computes the same information for runs starting from the sets of #-configurations it received and sends it to I. This process is repeated. If after $\exp_2(|Q|m^k)$ messages there is a message containing the empty set then the input is accepted (as final configurations are not transmitted, the presence of an empty set indicates a run where all leaves are accepting configurations).

A full proof is provided in the Appendix. □

Theorem 4 now follows from Lemmas 5, 7, and 8. □

When restricted to one-way computations, RAs can only express "regular" properties. The next proposition is easily shown using standard techniques.

Proposition 9. *MSO* can simulate every 1N-RA.*

3.2 Control

On strings of a special shape, RAs can simulate multi-head automata. These strings are of even length where the odd positions contain pairwise distinct elements and the even positions carry an a or a b. By storing the unique ids preceding the a's and b's, the RA can remember the positions of the heads of a multi-head automaton. Note that 2D-RAs can check whether the input string is of the desired form. As deterministic, nondeterministic, and alternating multi-head automata recognize precisely LOGSPACE, NLOGSPACE, and PTIME languages, respectively,

membership for 2D-RA, 2N-RA, and 2A-RA is hard for these classes, respectively [23, 15]. Furthermore, it is easy to see that the respective membership problems also belong to the infinite alphabet variants of these classes. Thus, we can show the following proposition in which all complexity classes are over infinite alphabets.

Proposition 10. *1. Membership of 2D-RA is complete for* LOGSPACE*;*
2. Membership of 2N-RA is complete for NLOGSPACE*; and*
3. Membership of 2A-RA is complete for PTIME*.*

The indexing technique above cannot be used for 1N-RAs, because they cannot check whether the odd positions form a unique index. However, we can extend (2) to 1N-RAs using a direct reduction from an NLOGSPACE-complete problem: ordered reachability. Details are provided in the Appendix.

Proposition 11. *Membership of 1N-RA is complete for* NLOGSPACE*.*

From Theorem 3 and Proposition 9 we can immediately conclude that the class accepted by 1N-RA is different from those accepted by 1N-RA and 2N-RA. It is a consequence of Proposition 11 that all four classes defined by the mentioned automata models are different unless the corresponding complexity classes collapse.

4 Pebble Automata

In this section we show that PAs are better behaved than RAs with regard to the connection to logic. In a sense, PAs are more "regular" than RAs. Indeed, we show that strong 1D-PAs can simlate FO* and that even the most liberal pebble model, 2A-PA, can be defined in MSO*. 1D-PAs are clearly more expressive than FO*; furthermore, we can separate 2A-RAs from MSO* under usual complexity-theoretic assumptions. Next, we show that weak one-way PAs do not suffice to capture FO*. Again, the proof is based on communication complexity. Finally, we prove that for strong PAs, the one-way, two-way, deterministic and nondeterministic variants collapse. Together with the straightforward closure under Boolean operations, concatenation and Kleene star, these results suggest that strong PAs define a robust class of languages.

4.1 Expressiveness

The proof of the next proposition is obvious and provided in the Appendix.

Proposition 12. *FO* is strictly included in strong 1D-PA.*

We next show that PAs are subsumed by MSO*. Thus, they behave in a "regular" manner.

Theorem 13. *MSO* can simulate 2A-PA.*

Proof. The proof is an extension to *infinite* alphabets of a proof in [17] where it is shown that alternating tree-walking pebble automata over *finite* alphabets can be simulated in MSO. In brief, we reduce the simulation problem of a k-pebble automaton to the *Alternating Graph Accessibility Problem*, AGAP [10]. An alternating graph (or *and/or graph*) is a graph $G = (V, E)$ whose nodes V are partitioned into *and*-nodes and *or*-nodes: $V = V_\wedge \cup V_\vee$. The problem consists in deciding whether a node $x \in V$ is *accessible* with accessibility defined as follows: an *and*-node is accessible if all its successors are accessible; an *or*-node is accessible if at least one of its successors is accessible. Note that *and* nodes with no successor are by definition accessible. It can be shown that the set of accessible nodes is definable in MSO and therefore in MSO*. Indeed, consider the formula

$$\varphi(x) := \forall S(\textit{reverse-closed}(S) \Rightarrow S(x)), \tag{1}$$

where *reverse-closed*(S) is:

$$\forall y(D_\vee(y) \wedge \exists z(E(y,z) \wedge S(z)) \Rightarrow S(y))$$
$$\wedge\ \forall y(D_\wedge(y) \wedge \forall z(E(y,z) \Rightarrow S(z)) \Rightarrow S(y)). \tag{2}$$

Here, $D_\vee$ and $D_\wedge$ are unary relations containing the *or*- and *and*-nodes, respectively.

Given an alternating k-pebble automaton $\mathcal{A} = (Q, U, q_0, F, T)$ over $\mathbf{D}$ and a string w, we construct the following *and/or* graph $G_{\mathcal{A},w} = (V, E)$. Its *or*-nodes are all configurations of $\mathcal{A}$ on w whose states are existential; its *and*-nodes are all configurations whose states are universal, together with an additional distinguished node ε. The set of edges E is $\{(\gamma, \gamma') \mid \gamma \vdash \gamma'\} \cup \{(\gamma, \epsilon) \mid \gamma \text{ is accepting}\}$. It follows directly from the definitions of $L(\mathcal{A})$ and of AGAP that a string w is in $L(\mathcal{A})$ iff the initial configuration γ_0 is accessible in the graph $G_{\mathcal{A},w}$. Hence, it only remains to show that we can express the AGAP problem on $G_{\mathcal{A},w}$ in MSO*. Here, the difficulty lies in the fact that the nodes in $G_{\mathcal{A},w}$ are tuples of nodes from the input structure w (a configuration $\gamma = [i, q, \theta]$ is represented by the i-tuple $(\theta(1), \ldots, \theta(i))$; q does not depend on w and will be encoded separately.) Hence, the set S in (1) is no longer unary. To circumvent that, we rely on a special property of $G_{\mathcal{A},w}$. Namely, if two nodes described by an i-tuple $(\theta(1), \ldots, \theta(i))$ and a j-tuple $(\theta'(1), \ldots, \theta'(j))$ are connected by an edge, then either $i = j$, or $i = j + 1$, or $i = j - 1$, and the tuples agree on all but the last position. This follows from the stack discipline on pebbles in $\mathcal{A}$ (only the last pebble can be moved) and allows us to quantify independently, on different portions of the graph. The construction of the MSO* formula relies on this observation; we outline this construction in the Appendix. □

It is open whether the above inclusion is strict. However, we can show the following.

Proposition 14. *For every $i \in \mathbb{N}$, there are MSO* formulas φ_i and ψ_i such that the model checking problem for φ_i and ψ_i is hard for Σ_i^P and Π_i^P, respectively. In contrast, membership for 2A-PAs is PTIME-complete.*

Here, the model checking problem for a logical formula consists in determining, given a string w, whether $w \models \varphi$. A proof is given in the Appendix. Since the first part of the proposition is already known for graphs, it suffices to observe that graphs can readily be encoded as strings. We end this section by considering weak PAs. Recall that this notion only makes a difference for one-way PAs. Unlike their strong counterparts, we show that they cannot simulate FO*, which justifies their name. The proof is again based on communication complexity.

Theorem 15. *Weak 1N-PA cannot simulate FO*.*

Proof. The proof is similar to the proof of Theorem 4. We show by a communication complexity argument that the FO*-expressible language $L^2_=$ defined in that proof cannot be recognized by a weak 1N-PA. In the current proof, however, we use a different kind of communication protocol which better reflects the behaviour of a weak 1N-PA. We define this protocol next. Recall that the strings we use are of the form $u\#v$ where u and v encode 2-hypersets. Let k be fixed and let S_1, S_2 be finite sets. The protocol has only one agent which has arbitrary access to the string u but only limited access to the string v. On u, its computational power is unlimited. The access to v is restricted as follows. There is a fixed function $f : \mathbf{D}^* \times \mathbf{D}^k \times S_1 \to S_2$ and the agent can evaluate f on all arguments $(v, \boldsymbol{d}, s)$, where $\boldsymbol{d}$ is a tuple of length k of symbols from u and $s \in S_1$. Based on this information and on u the agent decides whether $u\#v$ is accepted.

We show in the appendix that there is no function f such that there is an agent which recognizes $L^2_=$. It remains to show that on split strings a weak 1N-PA can be simulated by a protocol. Intuitively, this works as follows. On input $u\#v$, as we consider *one*-way *weak* PAs, whenever the current pebble enters v, the computation remains in v until that pebble is lifted. Therefore, the set of states which can be obtained when lifting the pebble only depends on v, the symbols below the pebbles placed in u and the placement information, that is, which

pebbles are located on the same positions. Hence, we define $f(v, \boldsymbol{d}, s)$ as the set of states that can be reached when pebble i enters v in state q and the pebbles in u are placed on $\boldsymbol{d}$. Here, i and q are coded into s. Moreover, s also contains the position placement of the pebbles in u. This function then provides enough information for the agent to simulate the 1N-PA. More details are given in the Appendix. □

4.2 Control

Our next result shows that all variants of strong pebble automata without alternation collapse. This suggests that strong PAs provide a robust model of automata.

Theorem 16. *The following have the same expressive power: 2N-PA, 2D-PA, strong 1N-PA and strong 1D-PA.*

Proof. We show that, for each 2N-PA $\mathcal{A}$, there is a strong 1D-PA $\mathcal{B}$ which accepts the same language. Actually, in our construction, $\mathcal{B}$ will use the same number of pebbles as $\mathcal{A}$. Let therefore $\mathcal{A} = (Q, q_0, F, T)$ be a 2N-PA with k pebbles.

For technical simplicity, we assume w.l.o.g. that $\mathcal{A}$ lifts pebbles only at the right delimiter (instead of lifting a pebble at an arbitrary position it can remember the target state q, go to the right delimiter and lift the pebble there, moving into state q).

First, we informally describe the idea of the construction. Recall the classical powerset construction which translates a non-deterministic 1-way automaton M (over a finite alphabet and without pebbles) into a deterministic one, M'. Intuitively, M' computes, for each prefix u of the input string $w = w_1 \cdots w_n$, the set of states that M might reach by reading u. M' performs an on-line simulation of M in the sense that each step in the computation of M corresponds to exactly one step of M'.

One cannot expect such an on-line simulation to work for 2-way automata (even for finite alphabets), as the non-deterministic behaviour of a 2-way automaton might involve moving in different directions. Instead (in the finite case without pebbles) the deterministic automaton can compute, for each position i in the input string w, a function f_i which describes the aggregated behaviour of M on $w_1 \cdots w_i$, i.e., the portion of the input which is at the left-hand side of the i-th position. The functions f_i can be computed inductively from left to right (forgetting f_{i-1} once f_i is computed). In the end, f_n and the knowledge of the possible first states that the automaton assumes at the right delimiter of the input provide all necessary information to decide whether w is accepted.

It is maybe a bit surprising that this approach can, by and large, be adapted to the case where pebbles are present and the alphabet is inifinte. We proceed as follows. First, we assume that $\mathcal{A}$ is further normalized in that it accepts its input only in configurations $[1, q, \theta]$, i.e., with only one pebble. By virtually adding two steps we view an accepting computation as consisting of (1) a first step in which the first pebble is placed at the first position, (2) a computation in which always at least one pebble is there, and (3) a final step in which the only remaining pebble is removed. Writing $[0, p, \theta_\emptyset]$ for a (virtual) configuration without pebble, to determine whether $\mathcal{A}$ accepts, one has to find out whether $[0, q_0, \theta_\emptyset] \vdash^* [0, q, \theta_\emptyset]$, for some final state q.

The latter can be done by recursively solving subproblems of the form $[i, q, \theta] \vdash^*_{>i} [i, q', \theta]$, where the subscript $> i$ indicates that only subcomputations are considered in which, at every step, more than i pebbles are present.

More formally, we show the following claim by induction on i (starting from $i = k$).

Claim. For each $i \in \{0, \ldots, k\}$ and each finite set R, there is a strong 1D-PA $\mathcal{B}_i$ (with k pebbles) such that, whenever $\mathcal{B}_i$ starts from a configuration $[i, p, \theta]$, where $p \in R$, the next configuration of depth i of $\mathcal{B}_i$ is $[i, (p, S), \theta]$, where $S = \{(q, q') \in Q \mid [i, q, \theta] \vdash^*_{>i} [i, q', \theta]\}$. In particular, the set of states of $\mathcal{B}_i$ contains R and $R \times 2^{Q \times Q}$.

First, it should be noted that the theorem follows from the claim, by setting $i = 0$. To this end, we let $R = \{p_0\}$ (the intended initial state of B_0) and obtain an automaton which ends up in a state (p_0, S), where S is the set $\{(q, q') \in Q \mid [0, q, \theta] \vdash^*_{>0} [0, q', \theta]\}$. The set of final states of $\mathcal{B}_0$ simply consists of all states (p_0, S), where S contains a pair (q_0, q) with $q \in F$.

For $i = k$ the proof of the claim is trivial, as there are no configurations of depth $> k$. Hence, $\mathcal{B}_k$ can compute (p, S) by a stay-transition. Therefore, let $i < k$ and suppose the claim holds for all $j > i$.

Intuitively, the set S can be computed by one left-to-right pass of the $(i+1)$st pebble. During this pass, $\mathcal{B}_i$ computes, for each position l in the string the sets of pairs (q, q'), such that there is a sub-computation which starts from state q at position 1 (position l, respectively) and ends in state q' at position l, without moving pebble $i + 1$ to positions $l' \geq l$. Note that such subcomputations might move pebbles $j > i + 1$ to positions $\geq l$. To compute this information, the automaton B_{i+1} is used repeatedly. More details are given in the Appendix. □

5 Registers versus Pebbles

The known inclusions between the classes that we considered are depicted in Figure 1. The pebble and register models are rather incomparable. Indeed, from the connection with logic we can deduce the following. As 2D-RA can already express non-MSO* definable properties, no two-way register model is subsumed by a pebble model. Conversely, as strong 1D-PAs can already express FO*, no strong pebble model is subsumed by any two-way register model. Some open problems about the relationships between register and pebble automata are given in Section 7.

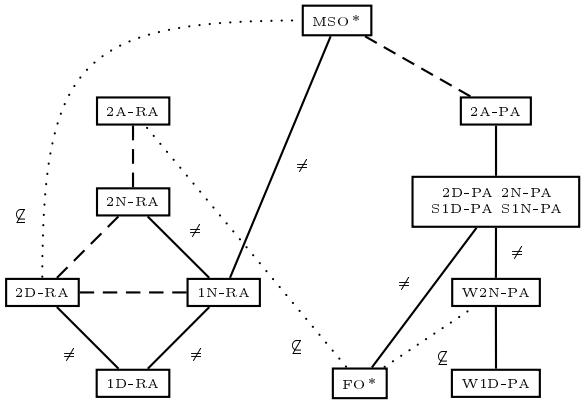

Fig. 1. Inclusions between the classes under consideration. Solid lines indicate inclusion (strictness shown as $\neq$), dotted lines indicate that the classes are incomparable. Dashed lines indicate strict inclusion subject to complexity-theoretic assumptions.

6 Decision Problems

We briefly discuss the standard decision problems for RAs and PAs. Kaminski and Francez already showed that non-emptiness of 1N-RAs is decidable and that it is decidable whether for a 1N-RA $\mathcal{A}$ and a 1N-RA $\mathcal{B}$ with 2-registers $L(\mathcal{A}) \subseteq L(\mathcal{B})$. We next show that *universality* (does an automaton accept every string) of 1N-RAs is undecidable, which implies that containment of *arbitry* 1N-RAs is undecidable. Kaminski and Francez further asked whether the decidability of non-emptiness can be extended to 2D-RAs: we show it cannot. Regarding PAs, we show that non-emptiness is already undecidable for weak 1D-PAs. This is due to the fact that, when PAs lift pebble i, the control is transferred to pebble $i - 1$. Therefore, even weak 1D-PAs can make several left-to-right sweeps of the input string.

6.1 Register Automata

Theorem 17. *It is undecidable whether a 1N-RA is universal.*

Proof. We use a reduction from Post's Correspondence Problem (PCP) which is well-known to be undecidable [11]. An *instance* of PCP is a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i, y_i \in \{a, b\}^*$ for $i = 1, \ldots, n$. This instance has a *solution* if there exist $m \in \mathbb{N}$ and $\alpha_1, \ldots, \alpha_m \in \{1, \ldots, n\}$ such that $x_{\alpha_1} \cdots x_{\alpha_m} = y_{\alpha_1} \cdots y_{\alpha_m}$.

We consider input strings of the form $w = u\#v$, where $\#$ is a delimiter and u, v are strings representing a candidate solution $(x_{\alpha_1}, \ldots, x_{\alpha_m}; y_{\beta_1}, \ldots, y_{\beta_m})$ for the PCP instance in a suitable way. To check whether such a candidate is indeed a solution, we roughly have to check whether (1) $\alpha_i = \beta_i$ for each i, that is, corresponding pairs are taken; and (2) both strings are the same, that is, corresponding positions in $x_{\alpha_1} \cdots x_{\alpha_m}$ and $y_{\alpha_1} \cdots y_{\alpha_m}$ carry the same symbol. To check (1) and (2), we use a double indexing system based on unique data values. The 1N-RA will only accept an input string when it is *not* of the required form or when the candidate solution does not represent a solution for the PCP instance. Hence, the 1N-RA accepts all inputs if and only if the PCP instance has *no* solution. Details are given in the Appendix. □

Corollary 18. *Containment of 1N-RAs is undecidable.*

The following question was also raised by Kaminski and Francez. In Section 3.2, we observed that two-way RAs can simulate multi-head automata on strings of a special shape. As non-emptiness of multi-head automata is undecidable the next proposition easily follows.

Proposition 19. *It is undecidable whether a 2D-RA is non-empty.*

6.2 Pebble Automata

The next result implies that all standard decision problems are undecidable for all classes of pebble automata.

Theorem 20. *It is undecidable whether a weak 1D-PA is non-empty.*

Proof. The proof is again a reduction from PCP and goes along the same lines as the proof of Theorem 17. Recall that in that proof the constructed 1N-RA accepts when it can guess an error. In contrast, the 1D-PA verifies one by one that there are no errors and accepts if this is the case. Details are given in the Appendix. □

7 Discussion

We investigated several models of computations for strings over an infinite alphabet. One main goal was to identify a natural notion of regular language and corresponding automata models. In particular, such a notion should agree in the finite alphabet case with the classical notion of regular language. We considered two plausible automata models: RAs and PAs. Our results tend to favor PAs as the more natural of the two. Indeed, the expressiveness of PAs lies between FO^* and MSO^*. The inclusion of FO^* provides a reasonable expressiveness lower bound, while the MSO^* upper bound indicates that the languages defined by PAs remain regular in a natural sense. Moreover, strong PAs are quite robust: all variants without alternation (one or two-way, deterministic or non-deterministic) have the same expressive power.

Some of the results in the paper are quite intricate. The proofs bring into play a variety of techniques at the confluence of communication complexity, language theory, and logic. Along the way, we answer several questions on RAs left open by Kaminski and Francez.

Several problems remain open: (*i*) can weak 1D-PA or weak 1N-PA be simulated by 2D-RAs? (*ii*) are 1D-RA or 1N-RA subsumed by any pebble model? (We know that they can be defined in MSO^*. As 1N-RAs are hard for NLOGSPACE they likely cannot be simulated by 2A-PAs.) (*iii*) are weak 1N-PAs strictly more powerful than weak 1D-PAs? (*iv*) are 2A-PAs strictly more powerful than 2N-PAs?

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
2. S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal connectives versus explicit timestamps to query temporal databases. *Journal of Computer and System Sciences*, 58(1):54–68, 1999.
3. M. Ajtai, R. Fagin, and L. J. Stockmeyer. The Closure of Monadic NP. *Journal of Computer and System Sciences*, 60(3):660–716, 2000.
4. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. Submitted, 2001.
5. G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. In Lloyd et al., editor, *Computational Logic – CL 2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1137–1151. Springer, 2000.
6. B. Courcelle. The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs. *Information and Computation*, 85(1):12–75, 1990.
7. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3. Springer, 1997.
8. E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation*, 140(1):26–81, 1998.
9. N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996.
10. R. Greenlaw, H. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation. P-Completeness Theory*. Oxford University Press, 1995.
11. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
12. J. Hromkovič. *Communication Complexity and Parallel Computing*. Springer-Verlag, 2000.
13. M. Kaminski and N. Francez. Finite-memory automata. In *Proceedings of 31th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 683–688, 1990.
14. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
15. A. K. Chandra, L. J. Stockmeyer. Alternation. In *Proceedings of 17th IEEE Symposium on Foundations of Computer Science (FOCS 1976)*, 98–108, 1976.
16. S. Maneth and F. Neven. Structured document transformations based on XSL. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming (DBPL'99)*, volume 1949 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2000.
17. T. Milo, D. Suciu, and V. Vianu. Type checking for XML transformers. In *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.
18. F. Neven. Extensions of attribute grammars for structured document queries. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming (DBPL'99)*, volume 1949 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2000.
19. F. Neven and T. Schwentick. Query automata. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS 1999)*, pages 205–214. ACM Press, 1999.
20. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of 19th ACM Symposium on Principles of Database Systems (PODS 2000), Dallas*, pages 145–156, 2000.
21. F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proceedings of 17th ACM Symposium on Principles of Database Systems (PODS 1998)*, pages 11–17. ACM Press, 1998.
22. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of 19th ACM Symposium on Principles of Database Systems (PODS 2000)*, pages 35–46. ACM, Press 2000.
23. I. H. Sudborough. On tape-bounded complexity classes and multihead finite automata. *Journal of Computer and System Sciences*, 10(1):62–76, 1975.
24. W. Thomas. Languages, automata, and logic. In Rozenberg and Salomaa [7], chapter 7.

Appendix

Proof of Theorem 3 continued. We explain how the 2D-RA can visit $\mathrm{lmo}_u(a_1)$, $\mathrm{lmo}_v(b_1)$, $\mathrm{lmo}_u(a_2)$, $\mathrm{lmo}_v(b_2)$, ... in sequence. Clearly, $\mathrm{lmo}_u(a_1)$ and $\mathrm{lmo}_v(b_1)$ are the first positions of u and v, respectively. If $\mathcal{A}$ has the values of a_i and b_i stored in its registers it can compute a_{i+1} and b_{i+1}. E.g., to compute a_{i+1} it proceeds as follows. It first moves its head to position $\mathrm{lmo}_w(a_i)$ by going to the left boundary and afterwards walking to the right until it encounters a_i. Now it tests, for all positions $\mathrm{lmo}_w(a_i)+j$, $j>0$, starting with $j=1$, whether they carry a leftmost occurrence of a symbol d. This is done as follows: from position $\mathrm{lmo}_w(a_i)+j$ it goes to the left until it either sees a d or reaches the left end of the string. In the former case, it goes back to $\mathrm{lmo}_w(a_i)+j$ (identified by the first d) and proceeds with $\mathrm{lmo}_w(a_i)+j+1$. In the latter case $\mathrm{lmo}_w(a_i)+j$ carries the leftmost d, therefore a_{i+1} is identified. The computation of b_{i+1} can be done in a similar way. □

Proof of Lemma 5. Let $k \geq 1$ be fixed. For each $i \leq k$, we define an FO* formula $\varphi_i(x_i^\ell, x_i^r, y_i^\ell, y_i^r)$ expressing on input w that the intervals $[x_i^\ell, x_i^r]$ and $[y_i^\ell, y_i^r]$ encode the same i-hyperset over $\mathbf{D}_j - \{\#\}$.

By $x \in [y,z]$ we abbreviate $y \leq x \wedge x \leq z$. By $\mathrm{clean}_i(x,y)$ we abbreviate the formula $\neg\exists z(x<z \wedge z<y \wedge \mathrm{val}(z) \in \{i,\ldots,k,\#\})$.

The formula φ_i is inductively defined as follows:

$$\begin{aligned}\varphi_1(x_1^\ell, x_1^r, y_1^\ell, y_1^r) := {} & \mathrm{val}(x_1^\ell)=1 \wedge \mathrm{val}(x_1^r)=1 \wedge \mathrm{val}(y_1^\ell)=1 \wedge \mathrm{val}(x_1^r)=1 \\ & \wedge \mathrm{clean}_1(x_1^\ell, x_1^r) \wedge \mathrm{clean}_1(y_1^\ell, y_1^r) \\ & \wedge \forall x(x \in [x_1^\ell, x_1^r] \rightarrow \exists y(y \in [y_1^\ell, y_1^r] \wedge \mathrm{val}(x)=\mathrm{val}(y)) \\ & \wedge \forall y(y \in [y_1^\ell, y_1^r] \rightarrow \exists x(x \in [x_1^\ell, x_1^r] \wedge \mathrm{val}(x)=\mathrm{val}(y)); \text{ and}\end{aligned}$$

for $i>1$, define

$$\begin{aligned}\varphi_i(x_i^\ell, x_i^r, y_i^\ell, y_i^r) := {} & \mathrm{val}(x_i^\ell)=i \wedge \mathrm{val}(x_i^r)=i \wedge \mathrm{val}(y_i^\ell)=i \wedge \mathrm{val}(x_i^r)=i \\ & \wedge \mathrm{clean}_i(x_i^\ell, x_i^r) \wedge \mathrm{clean}_i(y_i^\ell, y_i^r) \\ & \wedge \forall x_{i-1}^\ell, x_{i-1}^r(x_{i-1}^\ell, x_{i-1}^r \in [x_i^\ell, x_i^r] \wedge \mathrm{clean}_i(x_{i-1}^\ell, x_{i-1}^r) \rightarrow \\ & \exists y_{i-1}^\ell, y_{i-1}^r(y_{i-1}^\ell, y_{i-1}^r \in [y_i^\ell, y_i^r] \wedge \varphi_{i-1}(x_{i-1}^\ell, x_{i-1}^r, y_{i-1}^\ell, y_{i-1}^r)) \\ & \wedge \forall y_{i-1}^\ell, y_{i-1}^r(y_{i-1}^\ell, y_{i-1}^r \in [y_i^\ell, y_i^r] \wedge \mathrm{clean}_i(y_{i-1}^\ell, y_{i-1}^r) \rightarrow \\ & \exists x_{i-1}^\ell, x_{i-1}^r(x_{i-1}^\ell, x_{i-1}^r \in [x_i^\ell, x_i^r] \wedge \varphi_{i-1}(x_{i-1}^\ell, x_{i-1}^r, y_{i-1}^\ell, y_{i-1}^r)).\end{aligned}$$

The language $L_=^k$ is then expressed by the formula

$$\begin{aligned}\exists x_k^\ell, x_k^r, z, y_k^\ell, y_k^r(x_k^\ell = 1 \wedge x_k^r + 1 = z \wedge \mathrm{val}(z) = \# \\ \wedge z+1 = y_k^\ell \wedge y_k^r = \max \wedge \varphi_k(x_k^\ell, x_k^r, y_k^\ell, y_k^r)).\end{aligned}$$

Here, 1 and max refer to the first and last element of the string, respectively, and $+1$ is the successor function. □

Proof of Lemma 8. Let $\mathcal{B} = (Q, q_0, U, F, \tau, P)$ be a k-register 2A-RA working on split strings over $\mathbf{D}$. We assume w.l.o.g. that there are no transitions possible from final configurations. Further we assume that $\mathcal{B}$ never changes direction at the symbol #. Hence, on a split string $u\#v$, when it leaves u to the right it enters v and vice versa. Define p as the polynomial $p(n) := |Q|n^k$, that is, the number of configurations that can be assumed at the position labeled with # on strings with at most n different data values. Let $w = u\#v$ be an input split string; let D be the set of data values occurring in w. Let the position in the string of the split symbol # be e. Then, set $\Delta := \{[e,q,\tau] \mid q \in Q, \text{and } \tau : \{1,\ldots,k\} \rightarrow D\}$. Note that

$|\Delta| = p(|D|)$. Further, Δ is the set of configurations that can be assumed at the position of the split symbol. We refer to these as *#-configurations*.

A configuration $[i, q, \tau]$ is called a u-configuration if $i < e$ and a v-configuration if $i > e$. A #-configuration is called a $u\#$-configuration if it is assumed from the left and a $\#v$-configuration if it is assumed from the right. Note that, by our assumption that $\mathcal{B}$ never changes direction at #, we can distinguish these two sets of configurations.

We start by introducing the following notions. For an arbitrary configuration γ, a $(\gamma, \#)$-run is a run where the root is labeled with γ, all the leaves are labeled with final configurations or #-configurations and no inner vertex, besides possibly the root, is labelled by a #-configuration. For such a run t, we define Leaf-labels(t) as the set of #-configurations occurring at the leaves of t. For a set $C^{(1)} = \{\gamma_1, \ldots, \gamma_n\}$ of configurations, define $\ell(C^{(1)})$ as the set of sets of configurations

$$\{\bigcup_{i=1}^{n} \text{Leaf-labels}(t_i) \mid \text{for each } i,\ t_i \text{ is a } (\gamma_i, \#)\text{-run}\}.$$

Note that $\ell(C^{(1)})$ is computable for any set of configurations C. Finally, for a set $S^{(2)}$ of sets of configurations, define $\ell(S^{(2)}) := \bigcup_{C^{(1)} \in S^{(2)}} \ell(C^{(1)})$.

Let S_0 be the singleton 2-hyperset containing the singleton set $\{\gamma_0\}$, that is, $S_0 := \{\{\gamma_0\}\}$. Recall, that γ_0 is the initial configuration. Define a sequence of 2-hypersets of configurations by $S_i := \ell(S_{i-1})$, for all $i \geq 1$. Note that, for $i > 0$, if i is even then S_i is a 2-hyperset of $\#v$-configurations. If i is odd then S_i is a 2-hyperset of $u\#$-configurations.

Let us call a run t an i-pass run if all its leaf configurations are either #-configurations that are reached by computations that have visited # exactly i times. Clearly, if $C^{(1)} \in S_i$, then there is an i-pass run t of $\mathcal{B}$ on w such that Leaf-labels(t) $= C^{(1)}$ and vice versa. Hence, $\mathcal{B}$ accepts w iff there is an i and a $C^{(1)} \in S_i$ containing the empty set.

The protocol works as follows. Party `I` starts by sending S_1. For $i > 0$, when party `II` receives S_{2i-1} it responds with S_{2i}; when party `I` receives S_{2i} it responds with S_{2i+1}. The parties accept whenever a 2-hyperset with the empty set is transmitted. As there are only $\exp_2(|\Delta|)$ different 2-hypersets over Δ the parties can reject if the empty set was never obtained after $\exp_2(|\Delta|)$ rounds of messages. □

Proof of Proposition 9. Let $\mathcal{B} = (Q, q_0, F, \tau_0, T)$ be a 1N-RA. We describe the construction of an MSO* formula φ which holds for an input $w = w_1 \cdots w_n$ iff $\mathcal{B}$ accepts w, i.e., if $\mathcal{B}$ has an accepting computation on w. First of all, φ guesses, for each position i of w, the state that $\mathcal{B}$ takes after reading w_i. This can be done by existentially quantifying over sets $(S_q)_{q \in Q}$. Next, φ guesses, again for each position i, which transition $\mathcal{B}$ applies when it reads w_i. This is done by quantifying over sets $(T_t)_{t \in T}$. Now assume that there is an accepting computation of $\mathcal{B}$ on w and that $(S_q)_{q \in Q}$ and $(T_t)_{t \in T}$ are chosen accordingly. Then, for each register j, the register content of $\mathcal{B}$ before reading position i, can be determined as follows. It is the symbol w_l, where $l = \max\{m < i \mid m \in T_t, \text{ where } t \text{ is of the form } q \to (q', j, d)\}$. It is straightforward to express this in MSO* (even in FO*). With the ability to determine register contents it is now easy to check (again in FO*) that $(S_q)_{q \in Q}$ and $(T_t)_{t \in T}$ are consistent with the transition relation of $\mathcal{B}$. □

Proof of Proposition 11. Clearly, membership is in NLOGSPACE. For the hardness we use a reduction from ordered reachability: given an ordered graph with the property that if there is an edge from u to v then $u < v$; is there a path from the first node to the last one? This problem is hard for NLOGSPACE. Indeed, consider the following LOGSCPACE reduction from ordinary reachability. Given a graph G with n nodes, a source s and a sink t, we construct the graph G' with vertices $\{(i, j) \mid i, j \leq n\}$ where there is an edge from (i, j) to (i', j') iff $i' = i + 1$ and (j, j') is an edge in G. The source and sink then are $(1, s)$ and (n, t), respectively. Clearly, t is reachable from s in G iff (n, t) is reachable from $(1, s)$ in G'. Further, G' is computable in LOGSPACE. Hence, ordered reachability is hard for NLOGSPACE. We next

reduce the latter to the membership problem of 1N-RAs. The input to the 1N-RA is of the form $[1, abcd][2, efg]...[n]$. Here, $[i, a_1a_2a_3]$ encodes that there is an edge from i to each a_j. Then the 1N-RA accepts when n can be reached from 1 by following edges. Every ordered graph can be encoded as such a list. Hence, since ordered reachability is hard for NLOGSPACE, membership for 1N-RA is. □

Proof of Proposition 12. Clearly, FO* cannot define the **D**-strings of even length while even strong 1D-PAs can. An FO* sentence φ in prenex normal form can be evaluated in a rather straightforward way. We use one pebble for each quantifier. Pebble 1 is used for the outermost quantifier, the pebble with the largest number is used for the innermost quantifier. The automaton cycles through all possible assignments of positions to the pebbles hence to the variables. It maintains all information about equality and inequality between the symbols at the pebble positions in its state. □

Proof of Theorem 13. We continue the proof of Theorem 13. For simplicity, we can assume w.l.o.g. that for each universal state q and (i, s, P, V), either $\{\beta \mid (i, s, P, V, q) \to \beta\} = \emptyset$ or there are two states q_1, q_2 such that $\{\beta \mid (i, s, P, V, q) \to \beta\} = \{(q_1, stay), (q_2, stay)\}$. It will also be convenient to assume that Q is partitioned into disjoint $Q_1 \cup \ldots \cup Q_k$ such that states in Q_i "control" pebble i. Furthermore, we enumerate the states in Q such that $Q = \{q_0, q_1, \ldots, q_n\}$, and $Q_1 = \{q_0, \ldots, q_{n_1}\}$, $Q_2 = \{q_{n_1+1}, \ldots, q_{n_2}\}$, ..., $Q_k = \{q_{n_{k-1}+1}, \ldots, q_{n_k}\}$.

We show first the case when $\mathcal{A}$ uses a single pebble, to illustrate how one encodes the state in a configuration and how to encode the transitions. For simplicity we only consider transitions involving constants; dealing with transitions without constants is an easy generalization. In the case of only one pebble, configurations can be assimilated with pairs (q, x), and transitions can be simplified to $(s, q) \to (p, d)$ where $d \in \{stay, left, right\}$. The MSO* formula $\varphi_{\mathcal{A}}$ defining acceptance by $\mathcal{A}$ uses a different unary relation S_j for each state $q_j \in Q$. Namely $\varphi_{\mathcal{A}}$ is:

$$\varphi_{\mathcal{A}} := \forall S_0 \forall S_1 \ldots \forall S_{n_1} (\textit{reverse-closed} \to S_0(1)), \tag{3}$$

where *reverse-closed* is a sentence stating that $S_0, S_1, \ldots, S_{n_1}$ are closed under reverse transitions of $\mathcal{A}$ according to the *and/or* semantics. Thus, $\varphi_{\mathcal{A}}$ states that the initial configuration of $\mathcal{A}$ is accessible in the *and/or* graph $G_{\mathcal{A},w}$. It follows that $\varphi_{\mathcal{A}}$ holds iff $\mathcal{A}$ accepts w.

The formula *reverse-closed* is a direct representation of the transitions in $\mathcal{A}$ (and edges in $G_{\mathcal{A},w}$) in MSO*. For each transition $(a, q_u) \to \beta$ of $\mathcal{A}$ where q is existential, *reverse-closed* includes one conjunct. For example, if $\beta = (q_v, right)$, the corresponding conjunct is

$$\forall x \forall y ((val(x) = a \wedge succ(x, y) \wedge S_v(y)) \to S_u(x)),$$

where $succ(x, y)$ is the FO* formula defining the successor relation on $dom(w)$. If q_u is universal, $a \in \mathbf{D}$, and q_u's transitions under a are $(a, q_u) \to (q_v, stay)$ and $(a, q_u) \to (q_s, stay)$, *reverse-closed* includes the conjunct

$$\forall x ((val(x) = a \wedge S_v(x) \wedge S_s(x)) \to S_u(x)).$$

To see that $\varphi_{\mathcal{A}}$ holds on w iff $\mathcal{A}$ accepts w, it suffices to notice the similarity between (1) and (3). For that, one needs to observe how the formula for *reverse-closed* in a general graph (2) becomes the formula above when instantiated to $G_{\mathcal{A},w}$. For example notice that each *and*-node in $G_{\mathcal{A},w}$ has zero or two successors (hence the universal quantifier in (2) becomes a conjunction).

We now extend (3) to the case when k is arbitrary. We will define a predicate *reverse-closed*$^{(i)}$, for each $i = 1, \ldots, k$, stating that $S_{n_{i-1}+1}, \ldots, S_{n_i}$ are closed under reverse transitions of $\mathcal{A}$. Then, the MSO formula equivalent to $\mathcal{A}$ will be φ_A in (3), with *reverse-closed* replaced with *reverse-closed*$^{(1)}$. The predicate *reverse-closed*$^{(i)}$ assumes that pebbles $1, 2, \ldots, i-1$ are fixed, and their positions described by the free variables $x_1, \ldots, x_{i-1}$; it also has free variables

$S_0, S_1, \ldots, S_{n_{i-1}}$. The predicate only considers moves affecting pebbles $i, i+1, \ldots, k$. Partition $\mathcal{A}$'s transitions into $T = T_1 \cup \ldots \cup T_k$, where T_i is the set of transitions from states in Q_i. Then,

$$\textit{reversed-closed}^{(i)} := \bigwedge_{\tau \in T_i} \psi_\tau.$$

For transitions τ not lifting or placing a pebble, ψ_τ is the same as for *reverse-closed* above, except that now it also inspects the presence/absence of the previous $i-1$ pebbles and tests equality/inequality of the corresponding values with the current value. For example, for $i = 3$ and $\tau = ((3, a, \{1\}, \{2\}, q_u) \to (q_v, \textit{stay}))$ the corresponding ψ_τ is

$$\forall x(val(x) = a \ \wedge val(x) = val(x_1) \ \wedge \ val(x) \neq val(x_2) \ \wedge \ x \neq x_1 \ \wedge \ x = x_2 \ \wedge \ S_v(x) \to S_u(x))$$

In general, for a transition $(i, s, P, V, q) \to \beta$ we use the formulas

$$\xi_P(x_1, \ldots, x_i) := \bigwedge_{j \in P} (val(x_i) = val(x_j)) \wedge \bigwedge_{j \notin P} (val(x_i) \neq val(x_j)),$$

and

$$\psi_V(x_1, \ldots, x_i) := \bigwedge_{j \in V} (x_i = x_j) \wedge \bigwedge_{j \notin V} (x_i \neq x_j).$$

The new transitions are the *lift-current-pebble* and *place-new-pebble* transitions. These determine the following conjuncts in $\textit{reverse-closed}^{(i)}$:

- for $\tau = (i, s, P, V, q_u) \to (q_v, \textit{place-new-pebble})$,

  $$\psi_\tau = \forall x_i((val(x_i) = s \ \wedge \ \xi_P(x_1, \ldots, x_i) \ \wedge \ \psi_V(x_1, \ldots, x_i) \ \wedge \ \varphi^{(i+1)}) \Longrightarrow S_u(x_i)),$$

  where $\varphi^{(i+1)} = \forall S_{n_i+1} \ldots \forall S_{n_{i+1}} (\textit{reverse-closed}^{(i+1)} \Longrightarrow S_v(x_i))$. Note the resemblance of $\varphi^{(i+1)}$ to (3): here q_v acts as an initial state for pebble $i+1$.
- for $\tau = (i, s, P, V, q_u) \to (q_v, \textit{lift-current-pebble})$,

  $$\psi_\tau = \forall x_i((val(x_i) = s \ \wedge \ \xi_P(x_1, \ldots, x_i) \ \wedge \ \psi_V(x_1, \ldots, x_i) \ \wedge \ S_v(x_{i-1})) \to S_u(x_i)).$$

  Note that here q_u acts as a terminal state for pebble i.

This completes the proof of the translation of $\mathcal{A}$ into MSO*. Note that the stack discipline imposed on the use of pebbles is essential to the construction in the proof. □

Proof of Proposition 14. Ajtai, Fagin, and Stockmeyer [3] showed that for every level of the polynomial hierarchy (PH) there is an MSO formula over graphs such that model checking is hard for that level. Let φ be an MSO formula over graphs and let $G = (\{1, \ldots, n\}, E)$ be a graph such that determining $G \models \varphi$ is hard for a specific level of the PH. We next give an MSO* formula φ^* and a string $w(G)$ for which testing $w(G) \models \varphi^*$ is hard for that level of the PH. Define $w(G)$ as the string consisting of blocks of the form $\#i\, i_1 \ldots i_k$ where $i \in \{1, \ldots, n\}$ and $\{i_1, \ldots, i_k\} = \{j \mid G \models E(i, j)\}$. Further, let vertex$(x)$ be the formula $(\exists z')(\mathrm{val}(z') = \# \wedge \mathrm{succ}(z') = x)$. Then φ^* is obtained from φ by replacing every occurrence of $E(x, y)$ by

$$\exists y'(\mathrm{val}(y') = \mathrm{val}(y) \wedge x < y' \wedge \neg \exists z(x < z \wedge z < y \wedge \mathrm{val}(z) = \#).$$

Then inductively replace from the inside to the outside every occurrence of a subformula $\exists x \alpha$, $\forall \alpha$, $\exists X \alpha$, and $\forall X \alpha$, by $\exists x(\mathrm{vertex}(x) \wedge \alpha)$, $\forall x(\mathrm{vertex}(x) \to \alpha)$, $\exists X(\forall x'(X(x') \to \mathrm{vertex}(x)) \wedge \alpha)$, and $\forall X(\forall x'(X(x') \to \mathrm{vertex}(x)) \to \alpha)$.

A configuration of a k-pebble 2A-PA consists of a state and at most k positions of the pebbles. This takes only logarithmic space in the size of the input. Therefore, a 2A-PA can be executed in ALOGSPACE which equals PTIME. Hardness, for instance, follows from a reduction from AGAP [10] as defined in the proof of 13. □

Proof of Theorem 15 continued. First, we show that there is no function f such that there is an agent which recognizes $L^2_=$. Let D be a finite subset of $\mathbf{D}$, containing 1 and 2, and let $u\#v$ be an input string where u only consists of symbols from D. Let $d := |D|$, $m_1 := |S_1|$, and $m_2 := |S_2|$. We can assume w.l.o.g. that the agent always evaluates f for all possible arguments. As there are at most $d^k m_1$ many such arguments there are at most $m_2^{d^k m_1}$ many different "interactions" between the agent and function f. it is important here, that the protocol is non-adaptive, i.e., the order of the questions does not matter. Let $h = 2^{2^{d-2}}$. Let $u_1, \ldots, u_h$ be encodings of all the h possible 2-hypersets over $D - \{1,2\}$. If d is large enough with respect to k, m_1, and m_2 then there are more 2-hypersets on $D - \{1,2\}$ than different interactions, hence there must be encodings u, u' of 2-hypersets with $H(u) \neq H(u')$ such that the interactions on $u\#u$ and $u'\#u'$ are the same. Hence, the agent accepts $u\#u$ if and only if he accepts $u\#u'$, which is a contradiction.

It remains to show how to transform a weak 1N-PA that recognizes a language L of split strings into a protocol of the above type that recognizes L. We only have to define f and describe how the agent works. Let $\mathcal{A}$ be a weak 1N-PA for L with $k+1$ pebbles and which has Q as set of states. For simplicity, we assume that the automaton never places a new pebble on $\#$. To denote pebble placement, we use equivalence relations G that contain elements in $\{1, \ldots, k\}$ with the additional property that if $i+1 \in \mathrm{dom}(G)$ then $i \in \mathrm{dom}(G)$. Intuitively, each equivalence class of G contains all the pebbles that are placed on the same position in u. Let $S_1 := Q \times \mathcal{G}$, where $\mathcal{G}$ is the class of equivalence relations as described above. Further, $S_2 := \mathcal{P}(Q)$ is the powerset of Q. Let v be a string over $\mathbf{D}$, $\boldsymbol{d} := d_1, \ldots, d_k$ be a vector of k $\mathbf{D} \cup \{\triangleright\}$-values, $q \in Q$, and let $G \in \mathcal{G}$. Further, let u be the string $(d_1 \cdots d_k)^k$. The definition of f is based on a partial computation of $\mathcal{A}$ on $u\#v$. We inductively place pebbles on u according to $\boldsymbol{d}$ and G as follows. If $i \in \mathrm{dom}(G)$ is the smallest number in its equivalence class then pebble i is placed on a position which carries symbol d_i. If $j < i$ is the smallest number in i's equivalence class then pebble i is placed on the same position as pebble j. The first pebble not in $\mathrm{dom}(G)$ is put on $\#$. Now, we simulate all possible computations of $\mathcal{A}$ starting from this pebble assignment and state q until they either stop or reach a configuration in which no pebble is placed on v. The pebble assignment of the latter kind of configuration is the same as the one the computation started from, besides that there is no pebble on $\#$. The function value for f is now defined as the set of all states of these configurations.

To simulate $\mathcal{A}$, the agent proceeds as follows. On input $u\#v$ it simulates all possible computations of $\mathcal{A}$. Whenever in a computation a pebble reaches $\#$ it uses f to figure out the set of configurations that can be reached by a subcomputation which visits v. The argument that f is asked about consists of the state q, the vector of data values the pebbles are on (arbitrarily extended if less than k pebbles are placed) and an equivalence relation G which contains i if pebble i is placed; further, i and j are in the same equivalence class if pebbles i and j are at the same position. In this way, the agent can compute whether $\mathcal{A}$ accepts without knowing v. □

Proof of Theorem 16 continued. We introduce some more notation first. Let the input string w of length n be fixed. For $l \leq n$, let θ^l denote the $(i+1)$-pebble assignment which coincides with θ in the first i pebbles and for which $\theta^l(i+1) = l$. We write $S_{\rightleftharpoons}(\theta^l)$ for the set of pairs (q, q') of states such that there is a computation starting at $[i+1, q, \theta^l]$ and reaching $[i+1, q', \theta^l]$ which only includes configurations $[j, q'', \theta']$ that fulfil $j > i+1$ or ($j = i+1$ and $\theta'(i+1) \leq l$). Intuitively, this says that pebble $i+1$ is not allowed to move to the right of

position l. We write $S_{\rightharpoonup}(l)$ for the set of pairs (q, q') of states for which $[i+1, q', \theta^l]$ can be reached from $[i+1, q, \theta^1]$ by a subcomputation fulfilling the same property.

Now we are ready to finish the proof of the claim. The set $S_{\rightleftharpoons}(\theta^l)$ can be computed as follows. Let $R_{\rightleftharpoons}(\theta^l)$ be the set of pairs (q, q') for which one of the following conditions hold:

(a) There exist p_1, p_2 such that $[i+1, q, \theta^l] \vdash [i+1, p_1, \theta^{l-1}]$, $(p_1, p_2) \in S_{\rightleftharpoons}(\theta^{l-1})$, and $[i+1, p_2, \theta^{l-1}] \vdash [i+1, q', \theta^l]$;
(b) $[i+1, q, \theta^l] \vdash^*_{>i+1} [i+1, q', \theta^l]$;
(c) $[i+1, q, \theta^l] \vdash [i+1, q', \theta^l]$.

It is straightforward that $S_{\rightleftharpoons}(\theta)$ is simply the transitive closure of $R_{\rightleftharpoons}(\theta)$. The information needed for (a) can be computed in a left to right pass of pebble $i+1$. By induction we can assume a subautomaton $\mathcal{B}_{i+1}$ which computes, for each position l, the part of $R_{\rightleftharpoons}(\theta)$ contributed by condition (b). Note that (c) and the computation of the transitive closure do not require any pebble movements. During the same pass, the automaton can compute, for each position l, the set $S_{\rightharpoonup}(l)$. The computation of $S_{\rightharpoonup}(l)$ makes use of the sets $S_{\rightleftharpoons}(\theta)$.

From $S_{\rightleftharpoons}(\theta^n)$, $S_{\rightharpoonup}(n)$ and the transition relation of $\mathcal{A}$ it can deduce, during a lift-pebble step, the set S as in the claim. Note that n is the position of the right delimiter.

This completes the proof of the claim and of the theorem. □

Proof of Theorem 17. We use a reduction from Post's Correspondence Problem (PCP) which is well-known to be undecidable [11]. An *instance* of PCP is a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i, y_i \in \{a, b\}^*$ for $i = 1, \ldots, n$. This instance has a *solution* if there are $m \in \mathbb{N}$ and $\alpha_1, \ldots, \alpha_m \in \{1, \ldots, n\}$ such that $x_{\alpha_1} \cdots x_{\alpha_m} = y_{\alpha_1} \cdots y_{\alpha_m}$.

Suppose w.l.o.g. that the integer numbers $\{1, \ldots, n\}$ and the values $a, b, \&, \#$ are in $\mathbf{D}$. Denote the latter set of symbols by Sym. The initial register assignment assigns these values to the first $n+5$ registers. We consider input strings of the form $w = u\#v$, where $\#$ is a delimiter and, u and v are strings representing a candidate solution $(x_{\alpha_1}, \ldots, x_{\alpha_m}; y_{\beta_1}, \ldots, y_{\beta_m})$ for the PCP instance in a suitable way.

To check whether such a candidate is indeed a solution, we roughly have to check whether (1) $\alpha_i = \beta_i$ for each i, that is, corresponding pairs are taken; and (2) both strings are the same, that is, corresponding positions in $x_{\alpha_1} \cdots x_{\alpha_m}$ and $y_{\alpha_1} \cdots y_{\alpha_m}$ carry the same symbol. To check (1) and (2), we use a double indexing system based on unique data values.

We describe the construction in more detail. Each item x_{α_j} is encoded as a string of the form $\&\gamma\, \alpha_j\, \delta_1 a_1 \cdots \delta_k a_k$. Here, $\&$ is a separator, $\gamma \in \mathbf{D} - \mathrm{Sym}$ represents j by a unique data value, the a_i are from $\{a, b\}$ such that $x_{\alpha_j} = a_1 \cdots a_k$ and the δ_i represent the position of a_i in x by a unique data value. To achieve uniqueness, all γ- and δ-symbols are allowed to occur only once in u. Correspondingly, y_{β_j} is encoded by a string of the form $\&\gamma\, \beta_j\, \delta_1 a_1 \cdots \delta_k a_k$ such that $y_{\beta_j} = a_1 \cdots a_k$ and the corresponding conditions hold. A string $u\#v$ is *syntactically correct* if it has the properties described so far and fulfils the following two conditions.

- The γ-projection of u (i.e., the string consisting of the γ-entries of u) equals the γ-projection of v.
- The δ-projection of u equals the δ-projection of v.

A syntactically correct string $u\#v$ *represents a solution of the PCP instance*, if, for each δ, the symbol from $\{a, b\}$ at the right of δ is the same in u and in v.

We construct an 1N-RA $\mathcal{A}$ that accepts an input string w if and only if it is *not* syntactically correct or does *not* represent a solution. Hence, $\mathcal{A}$ accepts *all* inputs if and only if the PCP instance has *no* solution.

$\mathcal{A}$ checks that one of the following conditions holds for its input string w.

1. w is of the wrong form.

(a) w is not of the form $u\#v$ or u or v is not of the form $(\&\gamma i\ \delta_1 a_1 \cdots \delta_k a_k)^*$, $(i \in \{1, \ldots, n\})$.
(b) $x_i \neq a_1 \cdots a_k$ in some entry in u or $y_i \neq a_1 \cdots a_k$ in some entry in v.

2. The γ-projections are wrong.
   (a) the first γ in u differs from the first γ in v;
   (b) the last γ in u differs from the last γ in v;
   (c) two γ's in u are the same;
   (d) two γ's in v are the same; or
   (e) γ_1 and γ_2 are successors in u but not in v.

   The latter three conditions involve non-deterministic guesses of the positions where the failure takes place.
3. The δ-projections are wrong. This can be done in a completely analogous fashion.
4. w does not represent a solution:
   (a) The α-value for some γ in u is different from the corresponding β-value in v.
   (b) The a/b-value for some δ in u is different from the corresponding a/b-value in v.

Clearly, w is not a solution iff one of these conditions holds. □

Proof of Theorem 20. The proof is again a reduction from PCP. We use the notation introduced in the proof of Theorem 17. Consider the PCP instance $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i, y_i \in \{a, b\}^*$ for $i = 1, \ldots, n$. Again, we consider input strings of the form $u\#v$, where $\#$ is a delimiter and, u and v are strings representing a candidate solution for the PCP instance in the same way as in the proof of Theorem 17.

The weak 1D-PA $\mathcal{A}$ first checks whether the input is of the desired form and then accepts if the input encodes a solution of the PCP instance. As pebbles can only be moved to the right, we keep the first pebble on the first position and invoke subroutines at that position which are then performed by the other pebbles. $\mathcal{A}$ puts the first pebble down at the first position and then operates as follows.

1. $\mathcal{A}$ checks whether u and v are of the form $(\&\gamma\, i\, \delta_1 a_1 \cdots \delta_k a_k)^*$ and that $x_i = a_1 \cdots a_k$ and $y_i = a_1 \cdots a_k$, respectively, for all entries in u and v. This can be achieved by one left to right scan of the second pebble. When reaching the end of the string the pebble is simply lifted.
2. To check that w is syntactically correct, $\mathcal{A}$ further verifies the following.
   (a) All γ's in u are different: $\mathcal{A}$ places the second pebble on the first γ and scans the other γ's in u with the third pebble. If all are different from the first one, the second pebble is moved to the next γ and the process is repeated.
   (b) Checking that all γ's in v are different is similar.
   (c) The first γ in u equals the first γ in w: $\mathcal{A}$ puts the second pebble on the first γ and uses the third pebble to run to the first γ in w.
   (d) Checking that the last γ in u equals the last γ in w is similar.
   (e) If γ_1 and γ_2 are successors in u then they also are successors in v: this involves four pebbles (numbered 2 to 5). The second pebble cycles through all γ in u. For each such value d, the automaton proceeds as follows. The third pebble is placed on the γ right after the second pebble. The fourth pebble then cycles through the γ-symbols in v until it finds d. If d is found, the fifth pebble is placed on the γ right after d and consistency can be checked. If this check fails or d is not found in v then the input is rejected. Otherwise the three most recent pebbles are removed.
   (f) In an analogous way, it can also be verified that the δ's form an index.
3. To check that w represents a solution of the PCP instance $\mathcal{A}$ proceeds as follows.
   (a) $\mathcal{A}$ checks that when x_i is picked in u the corresponding choice in v is y_i. Hereto, the second pebble cycles through all γ values of u. $\mathcal{A}$ keeps the corresponding α-value in the finite memory, uses the third pebble to run to the same γ in v and checks whether the β-entry of the latter conforms to the α-entry of the former.

(b) In an analogous way, $\mathcal{A}$ can also check that the a-values at corresponding δ-entry are the same.

This completes the description of the construction of $\mathcal{A}$. It is straightforward to check that $\mathcal{A}$ accepts an input if and only if it represents a solution of the PCP instance, hence the PCP instance has a solution at all if and only if $L(\mathcal{A})$ is non-empty. $\square$