# Formal Reasoning on Infinite Data Values: An Ongoing Quest

Taolue Chen[1], Fu Song[2], and Zhilin Wu[3(✉)]

[1] Department of Computer Science, Middlesex University London, London, UK
[2] School of Information Science and Technology, ShanghaiTech University, Shanghai, China
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
wuzl@ios.ac.cn

**Abstract.** With motivations from formal verification and databases, formal models to reason about software systems that contain data values from an infinite domain became a research focus in theoretical computer science community during the last decade. In this chapter, we present a tutorial to summarise the state of the art of these formal models. We focus on automata models and logics. We organise the models according to the different approaches to deal with the data values from an infinite domain. Specifically, we present the following models, register automata (and related logics), data automata (and related logics), pebble automata, and symbolic automata and transducers. In addition, we also incorporate two application-oriented sections, respectively on formal models to reason about programs manipulating dynamic data structures, and on formal models for the static analysis of data-parallel programs. For these two sections, we choose to present separation logic with data constraints, logic of graph reachability and stratified sets, streaming transducers, and streaming numerical transducers. For each model, we introduce the basic definitions, use some examples to illustrate the model, and state the main theoretical properties of the model. We hope that this tutorial will be useful if one wants to have a bird's eye of view on this field and know the basic concepts underlying those models.

## 1 Introduction

In computer science, formal models usually refer to mathematical models to specify, recognise, generate, and transform a specific class of structures (e.g., words and trees). They typically include logic, automata, formal grammars, and rewriting systems. Formal models, as the basis of many branches of computer science, are subject to extensive investigations through the history of computer science [vL90]. Turing machines, together with $\lambda$-calculus, recursive functions, etc., are one of the first formal models of computation, which have a profound impact on almost every area of computer science. Another example is context-free grammars, which are the foundations of syntax analysis of programming languages, and hence all modern compliers.


J.P. Bowen et al. (Eds.): SETSS 2016, LNCS 10215, pp. 195–257, 2017.
DOI: 10.1007/978-3-319-56841-6_6

Logic and automata are two classes of the most well-known formal models. They have found numerous applications in algorithms and complexity, programming languages, verification, databases, artificial intelligence, etc. For instance, most automated verification techniques, in particular model checking, are based on logics and automata over infinite words and trees. In the database community, the query languages on semi-structured data (e.g. XML documents) are based on logics and automata over unranked trees. In addition, path query languages for graph databases are typically based on finite automata and regular expressions. Logic and automata are closely related: logics are usually succinct, declarative, and abstract, whilst automata are specific, imperative, and of low-level. It is quite common that logics are used as specification languages, and automata, accounting for the combinatorial aspect of the logics somehow, provide algorithmic means to reason about the specifications. A classical example is the satisfiability problem and model checking problem of *linear temporal logics* (LTL), which can be reduced to the nonemptiness and language inclusion problem of Büchi automata respectively [WVS83, VW86], yielding an efficient and elegant solution.

To some extent, it is fair to say that classical formal models deal with objects from a finite domain, which can be formalized by a finite alphabet. Intuitively, finite alphabets can be used to represent the events in concurrent systems and tags in XML documents. Formal models (logics and automata) over the finite alphabets have been investigated extensively and intensively. The Chomsky hierarchy classified the language (and the associated automata models) over finite alphabets into four levels: linear grammars and finite-state automata, context-free grammars and pushdown automata, context-sensitive grammars and linear-bounded automata, and phrase structure grammars and Turing machines. The theoretical properties of each level of the hierarchy, as well as their relationships, have been thoroughly investigated [HU79]. Over finite words and trees, finite-state automata have been shown to be expressively equivalent to the monadic second-order logic (MSO) [Büc60, Elg61, TW68]. On the other hand, over infinite words, Büchi automata and MSO have been proved to have the same expressibility [Büc62]. It is also worth mentioning that algebraic foundations of finite-state automata on finite words and trees have been established. One classical result in this field is that a regular language on finite words is expressible in first-order logic if and only if the syntactic monoid associated with the language is aperiodic [Sch65, MP71].

In the last decade, motivated by the formal analysis and verification of computer programs and query languages for XML documents and graph databases, formal models to reason about data values from an *infinite* domain have become a research focus of (theoretical) computer science [Seg06, D'A12, Kar16]. In these models, the alphabet is extended from a finite set $\Sigma$ to $\Sigma \times \mathbb{D}$, where $\mathbb{D}$ is an infinite data domain (e.g., the set of integers). These infinite alphabets can be intuitively interpreted as follows:

– if $\Sigma$ denotes the events, then $\mathbb{D}$ denotes the time of the events or the identifiers of the processes or threads where the event occurs,
– in XML documents and graph databases, if $\Sigma$ denotes tags of elements in XML documents or labels of graph nodes, then $\mathbb{D}$ denotes the attributes of elements or nodes.

In many cases, adding infinite data to the formal models with finite alphabets leads to undecidability of even very basic algorithmic problems. However, researchers have managed to discover quite a few remarkable exceptions where decidability, or even efficiency, are preserved. It is usually an art to identify the trade-off between decidability and expressiveness, which makes the field versatile and intricate. Nevertheless, this field is of vital importance from both theoretical and practical viewpoints: on the one hand, formal models over infinite alphabets are natural extensions of their counterparts over finite alphabets, so are of particular theoretical interests; on the other hand, they are intimately related to various applications from, for example, formal verification and XML databases.

The current chapter aims to provide a tutorial and survey for the state-of-the-art research in automata and logics over infinite alphabets and, in particular, their applications in program verification. This is not the first attempt, because of the importance of the subject. Segoufin provided an extensive survey on automata and logics over infinite alphabets in 2006 [Seg06]. In addition, we are aware of at least two other related surveys:

– D'Antoni's survey [D'A12] covered the automata and logics on data words and trees up to 2012, including register automata, data automata, pebble automata, symbolic automata, and related logics.
– Chap. 4 of Kara's dissertation [Kar16] included an up-to-date survey on automata and logics on data words, for instance, register automata, data automata, first-order logic, and temporal logics on data words.

This chapter provides a broader and up-to-date survey which covers the latest developments in this field (for example, formalisms for reasoning about dynamic data structures and data-parallel programs).

However, the reader should bear in mind that our survey is by no means comprehensive, nor subsumes the other excellent surveys mentioned above. Indeed, our selection of material may be subjective with respect to our own research interests and is bounded by the volume of this chapter. In particular

– we restrict the discussions to finite words and trees and do not present the results of these models and logics on $\omega$-words and trees,
– we are mostly driven by program verification, so do not include a huge body of work on atoms (also known as nominal sets or Fraenkel-Mostowski sets) which are used to define properties on data words and data trees in an abstract manner (see, e.g., [Boj13, BKLT13]),
– we do not include the work of extending Petri nets with data [HLL+16],
– finally, we do not cover the work on the automatic verification of database-driven systems [Via09].

*Plan of the Chapter.* Section 2 describes some notations used throughout this chapter. Section 3 presents register automata and related logics. Section 4 discusses data automata and first-order logic on data words. Section 5 introduces pebble automata, including their various sub-models. Section 6 discusses variable automata and temporal logics with data variable quantifications. Section 7 is devoted to symbolic automata and transducers. Sections 8 and 9 describe the formalisms for reasoning about programs manipulating dynamic data structures and data-parallel programs respectively.

## 2 Preliminaries

We use $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ to denote the set of natural numbers, the set of integers, and the set of rational numbers respectively. For any $n \in \mathbb{N}$, we write $[n]$ for $\{1, \cdots, n\}$.

We make use of a finite alphabet $\Sigma$ and an infinite set $\mathbb{D}$ of data values.

*Words and Data Words.* A *word* $w$ is a finite sequence over $\Sigma$. A *data word* $w$ is a finite sequence over $\Sigma \times \mathbb{D}$. In particular, $\varepsilon$ is used to denote the empty word or data word. A *language* is a set of words and a *data language* is a set of data words. Let $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ be a data word and $i \in [n]$. Then the *type* of $i$ in $w$, denoted by $\mathsf{type}_w(i)$, is $\triangleright$ if $i < n$ and $\overline{\triangleright}$ otherwise. Intuitively, $\triangleright$ means that the current position is not the rightmost position of the data word and $\overline{\triangleright}$ denotes the negation of this condition. In addition, the $\Sigma$-projection of $w$, denoted by $\mathsf{prj}_\Sigma(w)$, is $\sigma_1 \dots \sigma_n$. When $\Sigma$ is obvious from the context, we also write $\mathsf{prj}_\Sigma(w)$ as $\mathsf{prj}(w)$ for brevity. For a data word $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$, let $|w|$ denote the *length* of $w$, that is, $n$.

*Trees and Data Trees.* A *tree domain* $T$ is a nonempty finite subset of $\mathbb{N}^*$ such that (1) for every $xi \in T$ with $i \in \mathbb{N}$, we have $x \in T$, and (2) for every $xi \in T$ with $i \in \mathbb{N}$ and every $j : 0 \leq j < i$, we have $xj \in T$. In particular, we have $\varepsilon \in T$ for every tree domain $T$. Let $t, t' \in T$. We use $t \preceq_a t'$ to denote the fact that $t$ is an *ancestor* of $t'$, that is, $t' = tt''$ for some $t'' \in \mathbb{N}^*$. In addition, we use $t \preceq_s t'$ to denote the fact that $t$ a *left-sibling* of $t'$, that is, $t = t''i$ and $t' = t''j$ for some $t'' \in \mathbb{N}^*$ such that $i \leq j$. A *$\Sigma$-labeled tree* $\mathcal{T}$ is pair $(T, L)$, where $T$ is a tree domain and $L : T \to \Sigma$ is a labeling function. A *$\Sigma$-labeled data tree* $\mathcal{T}$ is a pair $(T, L, D)$, where $(T, L)$ is a $\Sigma$-labeled tree, and $D : T \to \mathbb{D}$ assigns each node a data value. A *tree language* is a set of $\Sigma$-labeled trees and a *data tree language* is a set of $\Sigma$-labeled data trees. Let $\mathcal{T}$ be a $\Sigma$-labeled tree $(T, L)$ or a $\Sigma$-labeled data tree $(T, L, D)$, and $t \in T$. Then the type of $t$ in $\mathcal{T}$, denoted by $\mathsf{type}_\mathcal{T}(t)$, is defined as a subset of $\{\triangledown, \overline{\triangledown}, \triangleright, \overline{\triangleright}\}$ such that

- if $ti \in T$ for some $i \in \mathbb{N}$, then $\triangledown \in \mathsf{type}_\mathcal{T}(t)$, otherwise, $\overline{\triangledown} \in \mathsf{type}_\mathcal{T}(t)$,
- if $t = t'i$ and $t'j \in T$ for some $j \in \mathbb{N}$ such that $j > i$, then $\triangleright \in \mathsf{type}_\mathcal{T}(t)$, otherwise, $\overline{\triangleright} \in \mathsf{type}_\mathcal{T}(t)$.

Intuitively, $\triangledown$ means that the current node is not a leaf and $\overline{\triangledown}$ denotes the negation of this condition. Similarly, $\triangleright$ means that the current node is not the

rightmost sibling of its parent and $\overline{\triangleright}$ denotes the negation of this condition. We use $\mathsf{TreeTypes}$ to denote the set of all possible types of nodes in trees or data trees. More specifically, $\mathsf{TreeTypes} = \{\{\mathsf{type}_1, \mathsf{type}_2\} \mid \mathsf{type}_1 \in \{\triangledown, \overline{\triangledown}\}, \mathsf{type}_2 \in \{\triangleright, \overline{\triangleright}\}\}$. When $\Sigma$ is obvious from the context, we usually use data trees to denote $\Sigma$-labeled data trees.

*Nondeterministic Finite-State Automata (NFA).* An NFA $\mathcal{A}$ is a tuple $(Q, \Sigma, q_0, \delta, F)$ such that $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of transitions, and $F \subseteq Q$ is a finite set of final states. A *deterministic* NFA (DFA) is an NFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ such that for each $(q, \sigma) \in Q \times \Sigma$, there is at most one $q' \in Q$ satisfying that $(q, \sigma, q') \in \delta$. A NFA or DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ is *complete* if for each $q \in Q$ and $\sigma \in \Sigma$, there is $q' \in Q$ such that $(q, \sigma, q') \in \delta$.

The semantics of NFAs is defined as follows: We use $\delta^*$ to denote the reflexive and transitive closure of $\delta$, that is, $(q, \varepsilon, q) \in \delta^*$ and $(q, w_1 w_2, q') \in \delta^*$ iff there is $q'' \in Q$ such that $(q, w_1, q'') \in \delta^*$ and $(q'', w_2, q') \in \delta^*$. A word $w$ is accepted by an NFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ if $(q_0, w, q') \in \delta^*$ for some $q' \in F$. Let $\mathcal{L}(\mathcal{A})$ denote the language defined by $\mathcal{A}$, that is, the set of words accepted by $\mathcal{A}$.

The following decision problems are considered for NFAs:

- Nonemptiness problem: Given an NFA $\mathcal{A}$, decide whether $\mathcal{L}(\mathcal{A}) \neq \varnothing$.
- Universality problem: Given an NFA $\mathcal{A}$, decide whether $\mathcal{L}(\mathcal{A}) = \Sigma^*$.
- Language inclusion problem: Given two NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, decide whether $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.
- Equivalence problem: Given two NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, decide whether $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$.

A language $L \subseteq \Sigma^*$ is *regular* if there is an NFA $\mathcal{A}$ defining $L$, that is, $L = \mathcal{L}(\mathcal{A})$. Given a regular language $L$, the *complement* language of $L$ is $\Sigma^* \setminus L$. We say that NFAs are *closed under intersection (resp. union)* if for every pair of NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, there is an NFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ (resp. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$). On the other hand, NFAs are closed under *complementation* if for each NFA $\mathcal{A}$, there is an NFA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. A complete DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ is *minimal* if for each complete DFA $\mathcal{A}' = (Q', \Sigma, q_0', \delta', F')$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$, it holds that $|Q| \leq |Q'|$.

**Theorem 1 ([HU79]).** *The following results hold for NFAs:*

- *NFAs are closed under all Boolean operations (i.e. intersection, union and complementation).*
- *For each NFA, an equivalent DFA can be constructed in exponential time.*
- *For each regular language $L$, there is an unique minimal complete DFA (up to isomorphism) defining $L$.*
- *The nonemptiness problem of NFAs is in NLOGSPACE and the universality problem (as well as language inclusion problem and equivalence problem) of NFAs is PSPACE-complete.*

*Many-Sorted First-Order Logic.* We assume a signature $\Omega = (\mathfrak{S}, \mathfrak{F}, \mathfrak{P})$, where $\mathfrak{S}$ is a countable set of *sorts*, $\mathfrak{F}$ is a countable set of *function symbols*, and $\mathfrak{P}$ is a countable set of *predicate symbols*. Each function and predicate symbol has an associated *arity*, which is a tuple of sorts in $\mathfrak{S}$. A function symbol with a single sort is called a *constant*. A predicate symbol with a single sort is called a *set*, which intuitively denotes a set of elements of that sort.

An $\Omega$-term is built as usual from the function symbols in $\mathfrak{F}$ and variables taken from a set $\mathcal{X}$ that is disjoint from $\mathfrak{S}$, $\mathfrak{F}$, and $\mathfrak{P}$. Each variable $x \in \mathcal{X}$ has an associated sort in $\mathfrak{S}$. In addition, we assume that the variables in $\mathcal{X}$ are linearly ordered $\preceq_{\mathcal{X}}$. When writing $t(\boldsymbol{x})$ for a vector of distinct variables $\boldsymbol{x}$ such that $\boldsymbol{x} = (x_1, \ldots, x_n)$ follows the ascending order of the linear order $\preceq_{\mathcal{X}}$, we assume that the variables occurring in the term $t$ are from $\boldsymbol{x}$. For a term $t(\boldsymbol{x})$ of sort $s$ such that $\boldsymbol{x} = (x_1, \ldots, x_n)$ and each $x_i$ for $i \in [n]$ is of sort $s_i \in \mathfrak{S}$, the term $t$ is said to be *of arity* $(s_1 \times \cdots \times s_n) \to s$. In addition, for a vector of terms $(t_1, \ldots, t_m)$ such that all the variables of $t_1, \ldots, t_m$ are from $\boldsymbol{x} = (x_1, \ldots, x_n)$, if $x_1 \preceq_{\mathcal{X}} x_2 \preceq_{\mathcal{X}} \cdots \preceq_{\mathcal{X}} x_n$, each $x_i$ for $i \in [n]$ is of sort $s_i$, and each $t_j$ for $j \in [m]$ is of sort $s'_j$, then $(t_1, \ldots, t_m)$ is said to be a term of arity $(s_1, \ldots, s_n) \to (s'_1, \ldots, s'_m)$. For readability, a term of arity $(s_1, \ldots, s_n) \to (s'_1, \ldots, s'_m)$ is also called a $(s_1, \ldots, s_n)/(s'_1, \ldots, s'_m)$-term. We use $(t_1, \ldots, t_m)(\boldsymbol{x})$ to denote a vector of terms whose variables are from $\boldsymbol{x}$. For convenience, we also write $t(\boldsymbol{x})$ as $\lambda \boldsymbol{x}.\ t$ and $(t_1, \ldots, t_m)(\boldsymbol{x})$ as $\lambda \boldsymbol{x}.\ (t_1, \ldots, t_m)$.

We assume the standard notions of $\Omega$-atoms, $\Omega$-literals, and $\Omega$-formulae, whose definitions can be found in some textbooks on mathematical logic (see e.g. [Gal85]). The set of free variables of a $\Omega$-formula $\psi$ is denoted by $\mathsf{free}(\psi)$. When writing $\psi(\boldsymbol{x})$, we assume that the free variables of $\psi$ are from $\boldsymbol{x}$. For a formula $\psi(\boldsymbol{x})$ such that $\boldsymbol{x} = (x_1, \ldots, x_n)$ and each $x_i$ for $i \in [n]$ is of sort $s_i \in \mathfrak{S}$, the formula $\psi$ is said to be *of arity* $s_1 \times \cdots \times s_n$. A formula $\psi$ that contains exactly one free variable (resp. two, $n \geq 3$ free variables) is called a *unary* (resp. *binary*, $n$-ary) $\Omega$-formula. A formula $\psi$ contains no free variables is called a 0-ary formula, aka a sentence. For $i, j \in \mathbb{N} \backslash \{0\}$, a formula $\psi(\boldsymbol{x})$ of arity $s^j$ (where $\boldsymbol{x} = (x_1, \ldots, x_j)$), and an $s^i/s^j$-term $\boldsymbol{f} = (f_1, \ldots, f_j)$, we use $\psi[\boldsymbol{f}/\boldsymbol{x}]$ to denote the formula obtained from $\psi$ by simultaneously replacing $x_1$ with $f_1$, ..., and $x_j$ with $f_j$.

An $\Omega$-interpretation $I$ maps: (i) each sort $s \in \mathfrak{S}$ to a set $s^I$, (ii) each function symbol $f \in \mathfrak{F}$ of arity $s_1 \times \cdots \times s_n \to s$ to a total function $f^I : s_1^I \times \cdots \times s_n^I \to s^I$ if $n > 0$, and to an element of $s^I$ if $n = 0$, and (iii) each predicate symbol $p \in \mathfrak{P}$ of sort $s_1 \times \cdots \times s_n$ to a subset of $p^I \subseteq s_1^I \times \cdots s_n^I$. An $\Omega$-assignment $\eta$ maps each variable $x \in \mathcal{X}$ of sort $s \in \mathfrak{S}$ to an element of $s^I$.

- For a term $t$, the interpretation of $t$ under $(I, \eta)$ for an $\Omega$-interpretation $I$ and $\Omega$-assignment $\eta$, denoted by $t^{(I,\eta)}$, can be defined inductively on the syntax of terms.
- The satisfiability relation between pairs of an $\Omega$-interpretation and an $\Omega$-assignment, and $\Omega$-formulae, written $I \models_\eta \psi$, is defined inductively, as usual.

We say that $(I, \eta)$ is a model of $\psi$ if $I \models_\eta \psi$. For an $\Omega$-sentence $\psi$, we also write $I \models \psi$ if there is an $\Omega$-assignment $\eta$ such that $I \models_\eta \psi$.

Let $\Omega$ be a signature and $\mathcal{I}$ be a set of $\Omega$-interpretations. Then $\mathsf{Th}(\mathcal{I})$, *the $\Omega$-theory associated with $\mathcal{I}$*, is the set of $\Omega$-sentences $\psi$ such that for each $I \in \mathcal{I}$, $I \models \psi$.

*Linear Temporal Logic.* Let $\Sigma$ be an alphabet. Then linear temporal logic (LTL) over $\Sigma$ is defined by the following rules,

$$\varphi \stackrel{\text{def}}{=} \sigma \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\ \varphi \mid \varphi\ \mathsf{U}\ \varphi,$$

where $\sigma \in \Sigma$.

Some additional temporal operators, $\mathsf{F}$ and $\mathsf{G}$, can be derived from $\mathsf{U}$, $\mathsf{F}\varphi_1 \equiv \mathsf{true}\ \mathsf{U}\ \varphi_1$ and $\mathsf{G}\varphi_1 \equiv \neg\mathsf{F}\neg\varphi_1$.

LTL formulae $\varphi$ are interpreted on pairs $(w, i)$, where $w$ is a word over $\Sigma$ and $i$ is a position of $w$. The semantics is formalised as a relation $(w, i) \models \varphi$ defined as follows. Let $\varphi$ be an LTL formula, $w = \sigma_1 \dots \sigma_n$ be a word, and $i \in [n]$.

- $(w, i) \models \sigma$ iff $\sigma_i = \sigma$,
- $(w, i) \models \neg\varphi_1$ iff not $(w, i) \models \varphi_1$,
- $(w, i) \models \varphi_1 \vee \varphi_2$ iff $(w, i) \models \varphi_1$ or $(w, i) \models \varphi_2$,
- $(w, i) \models \mathsf{X}\ \varphi_1$ iff $i < n$ and $(w, i + 1) \models \varphi_1$,
- $(w, i) \models \varphi_1\ \mathsf{U}\ \varphi_2$ iff there is $k : i \leq k \leq n$ such that $(w, k) \models \varphi_2$ and for each $j : i \leq j < k$, $(w, j) \models \varphi_1$.

In addition, LTL formulae can be turned into *positive normal forms*, where the negation symbols are only before atomic formulae, by introducing the dual operators $\overline{\mathsf{X}}$ and $\mathsf{R}$ for $\mathsf{X}$ and $\mathsf{U}$, that is, $\overline{\mathsf{X}}\varphi_1 \equiv \neg\mathsf{X}\neg\varphi_1$ and $\varphi_1\ \mathsf{R}\ \varphi_2 \equiv \neg((\neg\varphi_1)\ \mathsf{U}\ (\neg\varphi_2))$. To help understand the semantics of $\mathsf{R}$, we present its semantics explicitly here: $(w, i) \models \varphi_1\ \mathsf{R}\ \varphi_2$ iff either for all $k : i \leq k \leq n$, we have $(w, k) \models \varphi_2$, or there is $k : i \leq k \leq n$ such that $(w, k) \models \varphi_1$, and for each $j : i \leq j \leq k$, $(w, j) \models \varphi_2$. For instance, $\neg\mathsf{F}(a \wedge \mathsf{XG}b)$ can be turned into the positive normal form $\mathsf{G}(a \vee \overline{\mathsf{X}}\mathsf{F}\neg b)$.

More specifically, the positive normal forms of LTL formulae are defined by the following rules,

$$\varphi \stackrel{\text{def}}{=} \sigma \mid \neg\sigma \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\ \varphi \mid \overline{\mathsf{X}}\varphi \mid \varphi\ \mathsf{U}\ \varphi \mid \varphi\ \mathsf{R}\ \varphi,$$

where $\sigma \in \Sigma$.

## 3 Register Automata, LTL with Freeze Quantifiers, and XPath

Kaminski and Francez initialised the research of automata models over infinite alphabets. They introduced nondeterministic register automata ([KF94]), an extension of finite state automata with a set of registers which can store a symbol from an infinite alphabet.

Let $R$ be a finite set of registers. In addition, we assume that $\mathsf{cur} \notin R$ is a distinguished register which stores the data value in the current position of data

words. We use $R^{\copyright}$ to denote $R \cup \{\mathsf{cur}\}$. A *guard* formula over $R$ is defined by the rules $g \stackrel{\text{def}}{=} \mathsf{true} \mid \mathsf{false} \mid r_1 = r_2 \mid r_1 \neq r_2 \mid g \wedge g \mid g \vee g$, where $r_1, r_2 \in R^{\copyright}$. Let $\mathsf{G}_R$ denote the set of all guard formulae over $R$. An *assignment* $\eta$ over $R$ is a partial function from $R$ to $R^{\copyright}$. Let $\mathsf{A}_R$ denote the set of assignments over $R$. A *valuation* $\rho$ over $R$ is a function from $R^{\copyright}$ to $\mathbb{D}$. For a valuation $\eta$, $r \in R^{\copyright}$, and $d \in \mathbb{D}$, we use $\eta[d/r]$ to denote the valuation which is the same as $\eta$, except that $d$ is assigned to the register $r$.

**Definition 1 (Nondeterministic register automata).** *A nondeterministic register automaton (NRA) $\mathcal{A}$ is a tuple $(Q, \Sigma, R, q_0, \tau_0, \delta, F)$ where:*

- *$Q$ is a finite set of states,*
- *$\Sigma$ is the finite alphabet,*
- *$R$ is a finite set of registers,*
- *$q_0 \in Q$ is the initial state,*
- *$\tau_0 : R \to \mathbb{D}$ assigns initial values to the registers;*
- *$\delta \subseteq Q \times \Sigma \times \mathsf{G}_R \times \mathsf{A}_R \times Q$ is a finite set of transition rules (for readability, we also write a transition $(q, \sigma, g, \eta, q')$ as $q \xrightarrow{(\sigma,g,\eta)} q'$),*
- *$F \subseteq Q$ is the set of final states.*

*Semantics of NRAs.* Given an NRA $\mathcal{A} = (Q, R, q_0, \tau_0, \delta, F)$, a *configuration* of $\mathcal{A}$ is a pair $(q, \rho)$, where $q \in Q$ and $\rho$ is a valuation. A configuration $(q, \rho)$ is said to be *initial* if $q = q_0$ and $\rho(r) = \tau_0(r)$ for each $r \in R$. A *run* of $\mathcal{A}$ over a data word $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ is a sequence of configurations $(q_0, \rho_0) \dots (q_n, \rho_n)$ such that $(q_0, \rho_0)$ is the initial configuration, and for each $i \in [n]$, there is a transition $q_{i-1} \xrightarrow{(\sigma_i, g_i, \eta_i)} q_i$ in $\delta$ such that $\rho_{i-1}[d_i/\mathsf{cur}] \models g_i$ and $\rho_i$ is obtained from $\rho_{i-1}$ and $\eta_i$ as follows: for each $r \in R$, if $r \in \mathsf{dom}(\eta_i)$, then $\rho_i(r) = (\rho_{i-1}[d_i/\mathsf{cur}])(\eta_i(r))$, otherwise, $\rho_i(r) = \rho_{i-1}(r)$. A run is said to be *accepting* if $q_n \in F$. A data word $w$ is said to be *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on $w$. Let $\mathcal{L}(\mathcal{A})$ denote the set of data words accepted by $\mathcal{A}$. We say that a data language $L$ is defined by an NRA $\mathcal{A}$ if $\mathcal{L}(\mathcal{A}) = L$.

*Example 1.* Let $\Sigma = \{a\}$. The NRA illustrated in Fig. 1 defines the data language $L$ "in the data word, a data value occurs twice", where $q_0$ is the initial state, $q_2$ is an accepting state, and $\varnothing$ denotes the assignment with the empty domain.

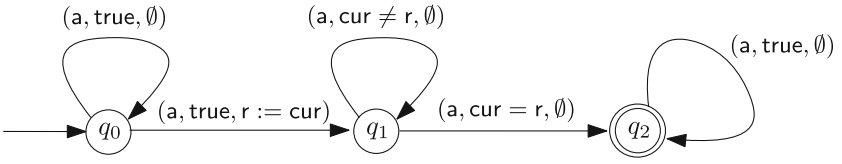



**Fig. 1.** An example of NRAs: "a data value occurs twice"

A *deterministic* NRA (DRA) is an NRA $\mathcal{A} = (Q, R, q_0, \tau_0, \delta, F)$ such that for each pair of distinct transitions $(q, \sigma, g_1, \eta_1, q_1')$ and $(q, \sigma, g_2, \eta_2, q_2')$ in $\mathcal{A}$, it holds that $g_1 \wedge g_2$ is unsatisfiable.

**Theorem 2 ([KF94,NSV01,DL09]).** *The following results hold for NRAs:*

- *NRAs are closed under union and intersection, but not closed under complementation.*
- *The nonemptiness problem of NRAs is PSPACE-complete, the universality problem of NRAs (as well as the language inclusion and equivalence problems) is undecidable.*
- *The nonemptiness, language inclusion, and equivalence problems of DRAs are PSPACE-complete.*

For instance, the complement of the data language $L$ in Example 1, that is, the data language comprising the data words where each data value occurs at most once, cannot be defined by NRAs. Intuitively, to guarantee that each data value occurs at most once, one needs unbounded many registers to store the data values that have been met so far when reading a data word from left to right.

Researchers also considered two extensions of nondeterministic register automata, alternating register automata and two-way nondeterministic register automata.

We next define alternating register automata over data words. We follow the notations in [Fig12].

**Definition 2 (Alternating register automata).** *An* alternating register automaton with $k$ registers *($ARA_k$) over data words is a tuple $\mathcal{A} = (\Sigma, R, Q, q_0, \tau_0, \delta)$, where $R = \{r_1, \ldots, r_k\}$ is a set of $k$ registers, $\Sigma, Q, q_0, \tau_0$ are the same as those in NRAs, and $\delta : Q \to \Phi$ is the transition function, where $\Phi$ is defined by the following grammar,*

$$\Phi \stackrel{def}{=} \mathsf{true} \mid \mathsf{false} \mid \sigma \mid \overline{\sigma} \mid \triangleright? \mid \overline{\triangleright}? \mid \mathsf{eq}_r \mid \overline{\mathsf{eq}_r} \mid q \vee q' \mid q \wedge q' \mid \mathsf{store}_r(q) \mid \triangleright q,$$

*where $r \in R$ and $q, q' \in Q$.*

Intuitively, $\sigma, \overline{\sigma}$ are used to detect the occurrences of letters from $\Sigma$. $\triangleright?$ and $\overline{\triangleright}?$ are used to describe the types of positions in data words, $\mathsf{eq}_r$ and $\overline{\mathsf{eq}_r}$ are used to check whether the data value in the register $r$ is equal to the current one, $q \vee q'$ makes a nondeterministic choice, $q \wedge q'$ creates two threads with the state $q$ and $q'$ respectively, $\mathsf{store}_r(q)$ stores the current data value to the register $r$ and transfers to the state $q$, $\triangleright q$ moves the reading head of the current thread one position to the right and transfers to the state $q$.

*Semantics of ARAs.* For defining semantics of ARAs, we introduce the concept of configurations.

Let $\mathcal{A}$ be an $\text{ARA}_k$. A *configuration* $c$ of $\mathcal{A}$ is a tuple $(i, \alpha, \sigma, d, \Lambda)$, where $i \in \mathbb{N} \setminus \{0\}$ denotes a position of a data word, $\alpha \in \{\triangleright, \overline{\triangleright}\}$ denotes the type of position $i$ in the data word, $(\sigma, d) \in \Sigma \times \mathbb{D}$ is the letter-data pair in position

$i$, $\Lambda \subseteq Q \times \mathbb{D}^R$ is a finite set of active threads in position $i$ where each thread $(q, \rho) \in \Lambda$ denotes that the state of the thread is $q$ and the valuation of the registers of the thread is $\rho$. Let $\mathcal{C}_\mathcal{A}$ denote the set of configurations of $\mathcal{A}$.

To define runs of $\mathcal{A}$ on data words, we introduce two types of transition relations on configurations, the *non-moving* relation $\longrightarrow_\epsilon \subseteq P_\mathcal{A} \times P_\mathcal{A}$ and the *moving* relation $\longrightarrow_\triangleright \subseteq \mathcal{C}_\mathcal{A} \times \mathcal{C}_\mathcal{A}$. For a given configuration $c = (i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda)$, the non-moving relation updates a thread $(q, \rho)$ of $c$ according to the transition function $\delta(q)$, and does not move the reading head. Formally, $\longrightarrow_\epsilon \subseteq \mathcal{C}_\mathcal{A} \times \mathcal{C}_\mathcal{A}$ is defined as follows,

- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \mathsf{true}$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \sigma$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \overline{\sigma'}$ and $\sigma \neq \sigma'$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \triangleright?$ and $\alpha = \triangleright$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \overline{\triangleright}?$ and $\alpha = \overline{\triangleright}$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = eq_r$ and $\rho(r) = d$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \overline{eq_r}$ and $\rho(r) \neq d$;
- for $j = 1, 2$, $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \{(q_j, \rho)\} \cup \Lambda)$, if $\delta(q) = q_1 \vee q_2$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \{(q_1, \rho), (q_2, \rho)\} \cup \Lambda)$, if $\delta(q) = q_1 \wedge q_2$;
- $(i, \alpha, \sigma, d, \{(q, \rho)\} \cup \Lambda) \longrightarrow_\epsilon (i, \alpha, \sigma, d, \{(q', \rho')\} \cup \Lambda)$, if $\delta(q) = \mathsf{store}_r(q')$ and $\rho' = \rho[d_i/r]$.

A configuration $(i, \alpha, \sigma, d, \Lambda)$ is *moving* if $\alpha = \triangleright$, $\Lambda \neq \varnothing$, and for every $(q, \rho) \in \Lambda$, we have $\delta(q) = \triangleright q'$. The moving relation $\longrightarrow_\triangleright$ advances some threads of a moving configuration to the right. More precisely,

$$(i, \alpha, \sigma, d, \Lambda) \longrightarrow_\triangleright (i + 1, \alpha', \sigma', d', \Lambda'),$$

if $(i, \alpha, \sigma, d, \Lambda)$ is a moving configuration, $\alpha' \in \{\triangleright, \overline{\triangleright}\}$, $\sigma' \in \Sigma$, $d' \in \mathbb{D}$, and $\Lambda' = \{(q', \rho) \mid (q, \rho) \in \Lambda, \delta(q) = \triangleright q'\}$.

Finally, we define the transition relation $\rightarrowtail \; = \; \longrightarrow_\varepsilon \cup \longrightarrow_\triangleright$.

A *run* of $\mathcal{A}$ over a data word $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ is a sequence of configurations $C_0 \dots C_m$ such that

- $C_0 = (1, \mathsf{type}_w(1), \sigma_1, d_1, \{(q_0, \tau_0)\})$,
- for each $j \in [m]$, there is $i \in [n]$ such that $C_j = (i, \mathsf{type}_w(i), \sigma_i, d_i, \Lambda)$,
- for each $j \in [m]$, $C_{j-1} \rightarrowtail C_j$.

A run $C_0 \dots C_m$ is *accepting* if $C_m = (i, \mathsf{type}_w(i), \sigma_i, d_i, \varnothing)$ for some $i \in [n]$. A data word $w$ is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ over $w$. Let $\mathcal{L}(\mathcal{A})$ denote the set of data words accepted by $\mathcal{A}$.

*Example 2.* Let $\Sigma = \{a\}$. Then the $\mathrm{ARA}_1$ $\mathcal{A} = (\{q_0, q_1, \dots, q_7, q_a, q_{\overline{eq_r}}\}, \Sigma, R = \{r\}, q_0, \tau_0, \delta)$ defines the data language "in the data word, no data values occur twice", that is, the complement language of $L$ in Example 1. Here $\tau_0(r) = c$ for some arbitrary $c \in \mathbb{D}$, $\delta(q_0) = q_a \wedge q_1$, $\delta(q_a) = a$, $\delta(q_1) = \mathsf{store}_r(q_2)$, $\delta(q_2) = \triangleright q_3$, $\delta(q_3) = q_a \wedge q_4$, $\delta(q_4) = q_{\overline{\mathsf{eq}_r}} \wedge q_5$, $\delta(q_{\overline{\mathsf{eq}_r}}) = \overline{\mathsf{eq}_r}$, $\delta(q_5) = q_6 \wedge q_7$, $\delta(q_6) = \triangleright q_3$, $\delta(q_7) = \mathsf{store}_r(q_3)$. Intuitively, in each position, the data value in the position is

stored into the register $r$ and a new thread is created, moreover, this data value (stored in $r$) will be checked to be different from each data value in the right of the position.

**Theorem 3** ([DL09, Fig12]). *The following facts hold for $ARA_k$'s:*

- *For each $k \geq 1$, $ARA_k$'s are closed under all Boolean operations.*
- *The nonemptiness problem of $ARA_2$'s is undecidable.*
- *The nonemptiness problem of $ARA_1$'s is decidable and non-primitive recursive.*

The nonemptiness of $\mathrm{ARA}_1$ was proved by defining a well-quasi-order over the set of configurations and utilising the framework of well-structured transition systems to achieve the decidability.

In [DL09], alternating register automata were introduced to solve the satisfiability problem of LTL with freezing quantifiers. Therefore, in the following, we define LTL with freezing quantifiers and illustrate how the satisfiability of LTL with freeze quantifiers can be reduced to the nonemptiness of alternating register automata.

**Definition 3 (LTL with freeze quantifiers).** *Let $R = \{r_1, \ldots, r_k\}$. The syntax of* Linear Temporal Logic with freeze quantifiers over $\Sigma$ *and $R$ (denoted by $LTL_k^{\downarrow}$) is defined by the following rules,*

$$\varphi \stackrel{def}{=} \sigma \mid \downarrow_{r_i} \varphi \mid \uparrow_{r_i} \mid \varphi \vee \varphi \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi \,\mathsf{U}\, \varphi,$$

*where $\sigma \in \Sigma, r_i \in R$.*

*Semantics of $LTL_k^{\downarrow}$.* $\mathrm{LTL}_k^{\downarrow}$ formulae are interpreted over a tuple $(w, j, \rho)$, where $w = (\sigma_1, d_1) \ldots (\sigma_n, d_n)$ is a data word, $j \in [n]$ is a position of $w$, and $\rho$ is an assignment over $R$. The semantics of $\mathrm{LTL}_k^{\downarrow}$ is classical for Boolean and temporal operators. For the modalities $\sigma$, $\downarrow_{r_i}$ and $\uparrow_{r_i}$,

- $(w, j, \rho) \models \sigma$, if $\sigma_j = \sigma$,
- $(w, j, \rho) \models \downarrow_{r_i} \varphi$, if $(w, j, \rho[d_j/r_i]) \models \varphi$,
- $(w, j, \rho) \models \uparrow_{r_i}$ if $\rho(r_i) = d_j$.

An $\mathrm{LTL}_k^{\downarrow}$ formula $\varphi$ is said to be *closed* if each occurrence of $\uparrow_{r_i}$ is in the scope of an occurrence of $\downarrow_{r_i}$. For a closed $\mathrm{LTL}_k^{\downarrow}$ formula $\varphi$, if $(w, 1, \rho) \models \varphi$, then $(w, 1, \rho') \models \varphi$ for any assignment $\rho'$. We define $\mathcal{L}(\varphi)$ as the set of data words $w$ such that $(w, 1, \rho) \models \varphi$ for some $\rho$. A data language $L$ is said to be defined by a closed $\mathrm{LTL}_k^{\downarrow}$ formula $\varphi$ if $\mathcal{L}(\varphi) = L$. An $\mathrm{LTL}_k^{\downarrow}$ formula $\varphi$ is said to be *satisfiable* if $\mathcal{L}(\varphi) \neq \varnothing$.

Similarly to LTL, $\mathrm{LTL}_k^{\downarrow}$ formulae can be turned into *positive normal form*, that is, the formulae where the negation symbols are only before atomic formulae $\sigma$ and $\uparrow_{r_i}$.

*Example 3.* The $\mathrm{LTL}_1^{\downarrow}$ formula $\mathsf{G}(\downarrow_{r_1} \neg\mathsf{XF} \uparrow_{r_1})$ defines the data language in Example 2. In addition, the $\mathrm{LTL}_1^{\downarrow}$ formula $\mathsf{G}(a \rightarrow \downarrow_{r_1} \mathsf{F}(b \wedge \uparrow_{r_1}))$, or $\mathsf{G}(\neg a \vee \downarrow_{r_1} \mathsf{F}(b \wedge \uparrow_{r_1}))$ in positive normal form, defines the data language "for each occurrence of $a$, there is an occurrence of $b$ on the right with the same data value".

**Theorem 4 ([DL09]).** *The following facts hold for $\mathrm{LTL}_k^{\downarrow}$:*

- *The satisfiability problem of $\mathrm{LTL}_2^{\downarrow}$ is undecidable.*
- *The satisfiability problem of $\mathrm{LTL}_1^{\downarrow}$ is decidable.*

From each $\mathrm{LTL}_1^{\downarrow}$ formula, an equivalent $\mathrm{ARA}_1$ can be constructed by an easy induction on the syntax of the positive normal form of $\mathrm{LTL}_1^{\downarrow}$ formulae. Then the decidability of $\mathrm{LTL}_1^{\downarrow}$ follows from Theorem 3. We use the following example to illustrate the construction of $\mathrm{ARA}_1$ from $\mathrm{LTL}_1^{\downarrow}$ formulae.

*Example 4.* Consider the $\mathrm{LTL}_1^{\downarrow}$ formula $\varphi = \mathsf{G}(\neg a \vee \downarrow_{r_1} \mathsf{F}(b \wedge \uparrow_{r_1}))$. From $\varphi$, we construct an $\mathrm{ARA}_1$ $\mathcal{A}_\varphi$ as follows:

- The set of states are $q_\psi$, where $\psi$ is a subformula of $\varphi$ or $\psi = \mathsf{X}\psi_1$ where $\psi_1$ is a subformula of $\varphi$.
- The initial state is $q_\varphi$.
- The transition function $\delta$ is defined as follows.
  - $\delta(q_\varphi) = q_{\neg a \vee \downarrow_{r_1}\mathsf{F}(b\wedge\uparrow_{r_1})} \wedge q_{\mathsf{X}\varphi}$, $\delta(q_{\mathsf{X}\varphi}) = \triangleright q_\varphi$,
  - $\delta(q_{\neg a \vee \downarrow_{r_1}\mathsf{F}(b\wedge\uparrow_{r_1})}) = q_{\neg a} \vee q_{\downarrow_{r_1}\mathsf{F}(b\wedge\uparrow_{r_1})}$, $\delta(q_{\neg a}) = \overline{a}$, $\delta(q_{\downarrow_{r_1}\mathsf{F}(b\wedge\uparrow_{r_1})}) = \mathsf{store}_{r_1}(q_{\mathsf{F}(b\wedge\uparrow_{r_1})})$,
  - $\delta(q_{\mathsf{F}(b\wedge\uparrow_{r_1})}) = q_{b\wedge\uparrow_{r_1}} \vee q_{\mathsf{XF}(b\wedge\uparrow_{r_1})}$,
  - $\delta(q_{b\wedge\uparrow_{r_1}}) = q_b \wedge q_{\uparrow_{r_1}}$, $\delta(q_b) = b$, $\delta(q_{\uparrow_{r_1}}) = \mathsf{eq}_{r_1}$, $\delta(q_{\mathsf{XF}(b\wedge\uparrow_{r_1})}) = \triangleright q_{\mathsf{F}(b\wedge\uparrow_{r_1})}$.

Two-way nondeterministic register automata can be defined as an extension of nondeterministic register automata in the same way as the two-way extension of finite-state automata. It turns out that the nonemptiness of two-way deterministic register automata is already undecidable.

**Theorem 5 ([DL09]).** *The nonemptiness problem of two-way deterministic register automata is undecidable.*

Alternating register automata on unranked trees have also been considered, mainly motivated to solve the satisfiability problem of fragments of Data-XPath (XPath with data value comparisons). In the following, we introduce a model of alternating one-register tree automata with guess and spread mechanism, denoted by $\mathrm{ATRA}_1$(guess, spread), then illustrate how the satisfiability of forward Data-XPath, a fragment of Data-XPath containing only forward navigation modalities, can be reduced to the nonemptiness of $\mathrm{ATRA}_1$(guess, spread).

**Definition 4 (Alternating one-register tree automata with guess and spread mechanism).** *An alternating one-register tree automaton with the guess and spread mechanism (denoted by $ATRA_1$(guess, spread)) $\mathcal{A}$ is defined as a tuple $(\Sigma, Q, q_0, \tau_0, \delta)$, such that $\Sigma, Q, q_0$ are as in $ARA_1$, $\tau_0 \in \mathbb{D}$ denotes the*

*initial value of the (unique) register, and* $\delta : Q \to \Phi$ *is the transition function, where* $\Phi$ *is defined by the following grammar,*

$$\Phi \stackrel{def}{=} \mathsf{true} \mid \mathsf{false} \mid \sigma \mid \overline{\sigma} \mid \odot? \mid \overline{\odot}? \mid \mathsf{eq} \mid \overline{\mathsf{eq}} \mid q \vee q' \mid q \wedge q' \mid \mathsf{store}(q) \mid \triangleright q \mid \triangledown q \mid \mathsf{guess}(q) \mid \mathsf{spread}(q, q'),$$

*where* $q, q' \in Q$*, and* $\odot \in \{\triangleright, \triangledown\}$*. An ATRA*$_1$ *is an ATRA*$_1$*(guess, spread) without guess and spread mechanisms.*

*Semantics of ATRA*$_1$*(guess, spread).* Let $\mathcal{A}$ be an ATRA$_1$(guess, spread). We introduce the concepts of node configurations and tree configurations as follows.

A *node configuration* $c$ of $\mathcal{A}$ is a tuple $(t, \alpha, \sigma, d, \Lambda)$, where $t \in \mathbb{N}^*$, $\alpha \in \mathsf{TreeTypes}$, $\sigma \in \Sigma$, $d \in \mathbb{D}$, and $\Lambda \subseteq Q \times \mathbb{D}$ is a finite set of active threads where each thread $(q, d) \in \Lambda$ denotes that the state of the thread is $q$ and the register holds the data value $d$.

A *tree configuration* $C$ of $\mathcal{A}$ is a finite set of node configurations. Let $N_{\mathcal{A}}$ denote the set of node configurations of $\mathcal{A}$, and $T_{\mathcal{A}} \subseteq 2^{N_{\mathcal{A}}}$ be the set of tree configurations. In addition, to define a run of $\mathcal{A}$, we introduce two types of transition relations, the *non-moving* relation $\longrightarrow_\epsilon \subseteq N_{\mathcal{A}} \times N_{\mathcal{A}}$ and the *moving* relation $\longrightarrow_\triangleright \subseteq N_{\mathcal{A}} \times N_{\mathcal{A}}$. For a given node configuration $c = (t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda)$, the non-moving relation updates a thread $(q, d')$ of $c$ according to the transition function $\delta(q)$, and does not move the reading head. Formally, $\longrightarrow_\epsilon \subseteq N_{\mathcal{A}} \times N_{\mathcal{A}}$ is defined as follows:

- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \mathsf{true}$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \sigma$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \overline{\sigma'}$ and $\sigma \neq \sigma'$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \odot?$ and $\odot \in \alpha$, where $\odot = \triangledown$ or $\triangleright$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \overline{\odot}?$ and $\overline{\odot} \in \alpha$, where $\odot = \triangledown$ or $\triangleright$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \mathsf{eq}$ and $d' = d$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \Lambda)$, if $\delta(q) = \overline{\mathsf{eq}}$ and $d' \neq d$;
- for $j = 1, 2$, $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \{(q_j, d')\} \cup \Lambda)$, if $\delta(q) = q_1 \vee q_2$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \{(q_1, d'), (q_2, d')\} \cup \Lambda)$, if $\delta(q) = q_1 \wedge q_2$;
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \{(q', d)\} \cup \Lambda)$, if $\delta(q) = \mathsf{store}(q')$.
- $(t, \alpha, \sigma, d, \{(q, d')\} \cup \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \{(q', d'')\} \cup \Lambda)$ for each $d'' \in \mathbb{D}$, if $\delta(q) = \mathsf{guess}(q')$.
- $(t, \alpha, \sigma, d, \Lambda) \longrightarrow_\epsilon (t, \alpha, \sigma, d, \{(q', d') \mid (q, d') \in \Lambda\} \cup \Lambda)$, if $\delta(q) = \mathsf{spread}(q, q')$.

A node configuration $(t, \alpha, \sigma, d, \Lambda)$ is *moving* if

- $\Lambda \neq \varnothing$, and for every $(q, d) \in \Lambda$, we have $\delta(q) = \triangledown q'$ or $\triangleright q'$,
- if there is $(q, d) \in \Lambda$ such that $\delta(q) = \odot q'$, then $\odot \in \alpha$, where $\odot = \triangledown$ or $\triangleright$.

The moving relation $\longrightarrow_{\triangledown}$ (resp. $\longrightarrow_{\triangleright}$) advances some threads of a moving node configuration to its leftmost child (resp. to its right sibling). Suppose $(t, \alpha, \sigma, d, \Lambda)$ is a moving node configuration.

- If $\triangledown \in \alpha$, then

$$(t, \alpha, \sigma, d, \Lambda) \longrightarrow_{\triangleright} (t0, \alpha', \sigma', d', \Lambda'),$$

  where $\alpha' \in \mathsf{TreeTypes}$, $\sigma' \in \Sigma$, $d' \in \mathbb{D}$, and $\Lambda' = \{(q', d) \mid (q, d) \in \Lambda, \delta(q) = \triangledown q'\}$.
- If $\triangleright \in \alpha$ and $t = t'i$, then

$$(t, \alpha, \sigma, d, \Lambda) \longrightarrow_{\triangleright} (t'(i+1), \alpha', \sigma', d', \Lambda'),$$

  where $\alpha' \in \mathsf{TreeTypes}$, $\sigma' \in \Sigma$, $d' \in \mathbb{D}$, and $\Lambda' = \{(q', d) \mid (q, d) \in \Lambda, \delta(q) = \triangleright q'\}$.

The transition relation $\rightarrowtail$ of tree configurations is defined as follows. Let $C_1, C_2$ be two tree configurations. Then $C_1 \rightarrowtail C_2$ if one of the following conditions hold:

- $C_1 = \{c\} \cup C'$ and $C_2 = \{c'\} \cup C'$ such that $c \longrightarrow_{\varepsilon} c'$.
- $C_1 = \{c\} \cup C'$, $c = (t, \alpha, \sigma, d, \Lambda)$, $\alpha = \{\triangleright, \overline{\triangledown}\}$, $C_2 = \{c'\} \cup C'$ such that $c \longrightarrow_{\triangleright} c'$.
- $C_1 = \{c\} \cup C'$, $c = (t, \alpha, \sigma, d, \Lambda)$, $\alpha = \{\overline{\triangleright}, \triangledown\}$, $C_2 = \{c'\} \cup C'$ such that $c \longrightarrow_{\triangledown} c'$.
- $C_1 = \{c\} \cup C'$, $c = (t, \alpha, \sigma, d, \Lambda)$, $\alpha = \{\triangleright, \triangledown\}$, $C_2 = \{c'_1, c'_2\} \cup C'$ such that $c \longrightarrow_{\triangleright} c'_1$ and $c \longrightarrow_{\triangledown} c'_2$.

A *run* of $\mathcal{A}$ over a data tree $\mathcal{T} = (T, L, D)$ is a sequence of tree configurations $C_0 \ldots C_n$ such that

- $C_0 = \{(\varepsilon, \mathsf{type}_{\mathcal{T}}(\varepsilon), L(\varepsilon), D(\varepsilon), \{(q_0, \tau_0)\})\}$,
- for each $i \in [n]$ and each $(t, \alpha, \sigma, d, \Lambda) \in C_i$, we have $t \in T$, $\alpha = \mathsf{type}_{\mathcal{T}}(t)$, $\sigma = L(t)$, and $d = D(t)$,
- for each $i \in [n]$, $C_{i-1} \rightarrowtail C_i$.

A run $C_0 \ldots C_n$ is *accepting* if $C_n \subseteq \{(t, \mathsf{type}_{\mathcal{T}}(t), L(t), D(t), \varnothing) \mid t \in T\}$. A data tree $\mathcal{T}$ is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ over $\mathcal{T}$. Let $\mathcal{L}(\mathcal{A})$ denote the set of data trees accepted by $\mathcal{A}$.

**Theorem 6 ([Fig12]).** *The following results hold for $ATRA_1$'s and $ATRA_1$(guess, spread)'s:*

- *$ATRA_1$'s are closed under all Boolean operations. On the other hand, $ATRA_1$(guess, spread)'s are closed under union and intersection, but not closed under complementation.*
- *The nonemptiness problem of $ATRA_1$(guess, spread)'s is decidable and non-primitive recursive.*

It was shown in [Fig12] that the data tree language $L$ in Example 5 is not definable in ATRA$_1$(guess, spread)'s. On the other hand, the complement of $L$ is definable in ATRA$_1$(guess, spread)'s. This demonstrates that ATRA$_1$(guess, spread)'s are not closed under complementation. Similarly to ARA$_1$'s, the decidability of the nonemptiness problem of ATRA$_1$(guess, spread)'s is also proved by utilising well-structured transition systems.

Data trees can also be seen as an abstraction of XML documents. XPath is a widely used query and navigation language for XML documents.

**Definition 5 (Data-aware XPath).** *Let* $\mathcal{O} \subseteq \{\downarrow, \uparrow, \rightarrow, \leftarrow, \downarrow^*, \uparrow^*, \rightarrow^*, \leftarrow^*\}$. *Data-aware XPath with set of axes from* $\mathcal{O}$*, denoted by XPath*$(\mathcal{O}, =)$*, comprises two types of formulae, path expressions* $\alpha$ *and node expressions* $\varphi$*, defined as follows:*

$$\alpha \stackrel{def}{=} o \mid [\varphi] \mid \alpha \cdot \alpha, \text{ where } o \in \mathcal{O},$$
$$\varphi \stackrel{def}{=} \sigma \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\alpha\rangle \mid \langle\alpha = \alpha\rangle \mid \langle\alpha \neq \alpha\rangle, \text{ where } \sigma \in \Sigma.$$

*Suppose* $\mathcal{F} = \{\downarrow, \rightarrow, \downarrow^*, \rightarrow^*\}$. *Then we call XPath*$(\mathcal{F}, =)$ *as the* forward fragment *of XPath*$(\mathcal{O}, =)$.

*Semantics of XPath*$(\mathcal{O}, =)$. XPath$(\mathcal{O}, =)$ formulae are interpreted on data trees $\mathcal{T} = (T, L, D)$. The semantics of path expressions and node expressions are specified by $[\![\alpha]\!]^{\mathcal{T}} \subseteq T \times T$ and $[\![\varphi]\!]^{\mathcal{T}} \subseteq T$ as follows:

- $[\![\downarrow]\!]^{\mathcal{T}} = \{(t, ti) \mid ti \in T\}$, $[\![\uparrow]\!]^{\mathcal{T}} = \{(ti, t) \mid ti \in T\}$,
- $[\![\rightarrow]\!]^{\mathcal{T}} = \{(ti, t(i+1)) \mid t(i+1) \in T\}$, $[\![\leftarrow]\!]^{\mathcal{T}} = \{(t(i+1), ti) \mid t(i+1) \in T\}$,
- $[\![\downarrow^*]\!]^{\mathcal{T}} = \{(t, t') \mid t, t' \in T, t \preceq_a t'\}$, $[\![\uparrow^*]\!]^{\mathcal{T}} = \{(t', t) \mid t, t' \in T, t \preceq_a t'\}$,
- $[\![\rightarrow^*]\!]^{\mathcal{T}} = \{(t, t') \mid t, t' \in T, t \preceq_s t'\}$, $[\![\leftarrow^*]\!]^{\mathcal{T}} = \{(t', t) \mid t, t' \in T, t \preceq_s t'\}$,
- $[\![[\varphi]]\!]^{\mathcal{T}} = \{(t, t) \mid t \in [\![\varphi]\!]^{\mathcal{T}}\}$,
- $[\![\alpha_1 \cdot \alpha_2]\!]^{\mathcal{T}} = \{(t, t') \in T \times T \mid \exists t'' \in T.\ (t, t'') \in [\![\alpha_1]\!]^{\mathcal{T}}, (t'', t') \in [\![\alpha_2]\!]^{\mathcal{T}}\}$,
- $[\![\sigma]\!]^{\mathcal{T}} = \{t \in T \mid L(t) = \sigma\}$, $[\![\neg\varphi]\!]^{\mathcal{T}} = T \setminus [\![\varphi]\!]^{\mathcal{T}}$,
- $[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{T}} = [\![\varphi_1]\!]^{\mathcal{T}} \cup [\![\varphi_2]\!]^{\mathcal{T}}$, $[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{T}} = [\![\varphi_1]\!]^{\mathcal{T}} \cap [\![\varphi_2]\!]^{\mathcal{T}}$,
- $[\![\langle\alpha\rangle]\!]^{\mathcal{T}} = \{t \in T \mid \exists t'.\ (t, t') \in [\![\alpha]\!]^{\mathcal{T}}\}$,
- $[\![\langle\alpha_1 = \alpha_2\rangle]\!]^{\mathcal{T}} = \{t \in T \mid \exists t', t''.\ (t, t') \in [\![\alpha_1]\!]^{\mathcal{T}}, (t, t'') \in [\![\alpha_2]\!]^{\mathcal{T}}, D(t') = D(t'')\}$,
- $[\![\langle\alpha_1 \neq \alpha_2\rangle]\!]^{\mathcal{T}} = \{t \in T \mid \exists t', t''.\ (t, t') \in [\![\alpha_1]\!]^{\mathcal{T}}, (t, t'') \in [\![\alpha_2]\!]^{\mathcal{T}}, D(t') \neq D(t'')\}$.

Let $\varphi$ be a node expression in XPath$(\mathcal{O}, =)$ and $\mathcal{T}$ be a data tree. Then $\mathcal{T}$ satisfies $\varphi$, denoted by $\mathcal{T} \models \varphi$, if $\varepsilon \in [\![\varphi]\!]^{\mathcal{T}}$. We use $\mathcal{L}(\varphi)$ denote the set of data trees satisfying $\varphi$. The satisfiability problem of XPath$(\mathcal{O}, =)$ is defined as follows: Given a node expression $\varphi$, decide whether $\mathcal{L}(\varphi) \neq \varnothing$. The query containment problem of XPath$(\mathcal{O}, =)$ is defined as follows: Given two node expressions $\varphi_1, \varphi_2$, decide whether $\mathcal{L}(\varphi_1) \subseteq \mathcal{L}(\varphi_2)$. Since the node expressions of XPath$(\mathcal{O}, =)$ are closed under complementation, it follows that the query containment problem of XPath$(\mathcal{O}, =)$ can be reduced to the satisfiability problem.

*Example 5.* Let $L$ be the data tree language comprising the data trees such that "no data values in two distinct positions are the same". Then $L$ can be defined by the XPath$(\mathcal{F}, =)$ formula $\varphi$,

$$\varphi \stackrel{\text{def}}{=} \neg\langle\downarrow^* [\langle\alpha_\varepsilon =\downarrow^+\rangle \vee \langle\downarrow^* = \rightarrow^+ \downarrow^*\rangle]\rangle,$$

where $\alpha_\varepsilon \stackrel{\text{def}}{=} [\bigvee_{\sigma\in\Sigma} \sigma]$, $\downarrow^+ = \downarrow \cdot \downarrow^*$ and $\rightarrow^+ = \rightarrow \cdot \rightarrow^*$.

**Theorem 7 ([Fig12]).** *The satisfiability problem (hence the query containment problem) of XPath$(\mathcal{F}, =)$ is decidable.*

Theorem 7 is proved by a reduction to the nonemptiness of ATRA$_1$(guess, spread), that is, for each XPath$(\mathcal{F}, =)$ node expression $\varphi$, an ATRA$_1$(guess, spread) $\mathcal{A}_\varphi$ can be constructed such that $\varphi$ is satisfiable iff $\mathcal{A}_\varphi$ is nonempty. Nevertheless, although the satisfiability of XPath$(\mathcal{F}, =)$ can be reduced to the nonemptiness of ATRA$_1$(guess, spread), ATRA$_1$(guess, spread)'s are still unable to capture XPath$(\mathcal{F}, =)$. For instance, the XPath$(\mathcal{F}, =)$ formula $\varphi$ in Example 5 is not definable in ATRA$_1$(guess, spread)'s [Fig12].

*Further Reading.* Kaminski and Tan initialised the investigation on regular expressions over infinite alphabets in [KT06]. Later on, with the motivations from path query processing in graph databases, Libkin et al. revisited this topic, proposed regular expressions with memories, and showed they are expressively equivalent to NRAs [LTV15]. Cheng and Kaminski investigated context free languages over infinite alphabets and showed that context free grammars over infinite alphabets and pushdown register automata are expressively equivalent [CK98]. Murawski et al. showed that the emptiness problem of pushdown register automata is EXPTIME-complete [MRT14].

## 4 Data Automata and First-Order Logic on Data Words

In the following, we will introduce data automata and its variants, as well as first-order logic on data words. Data automata were introduced by Bojanczyk et al. in [BMS+06,BDM+11], aiming at solving the satisfiability problem of first-order logic with two variables on data words.

We introduce some additional notations for data words first.

Let $w = (\sigma_1, d_1)\dots(\sigma_n, d_n)$ be a data word and $i \in [n]$. The *profile* of $w$, denoted by $\mathsf{prof}(w)$, is $\sigma'_1 \dots \sigma'_n$ such that $\sigma'_1 = (\sigma, \bot)$, and for each $i : 2 \le i \le n$, $\sigma'_i = (\sigma, \top)$ if $d_i = d_{i-1}$, and $\sigma'_i = (\sigma, \bot)$ otherwise. *A class of* $w$ is a maximal nonempty set of positions $X \subseteq [n]$ with the same data value. Let $X \subseteq [n]$. Then $w|_X$ denotes the restriction of $w$ to the set of positions in $X$. For instance, let $w = (a,1)(b,2)(a,2)(b,1)$, then $\mathsf{prof}(w) = (a,\bot)(b,\bot)(a,\top)(b,\bot)$, the class of $w$ corresponding to the data value 1 is $X = \{1,4\}$, and $w|_X = (a,1)(b,1)$.

The concept of class strings is used in the definition of data automata and its variants.

**Definition 6 (Class strings).** *Suppose the alphabet $\Sigma$ satisfies that $0, 1 \notin \Sigma$. For a data word $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ and a class $X$ of $w$:*

- *the* $X$-class string *of $w$, denoted by $\mathsf{cstr}_X(w)$, is defined as $w|_X$,*
- *the* position-preserving $X$-class string *of $w$, denoted by $\mathsf{pcstr}_X(w)$, is defined as the word $\sigma'_1 \dots \sigma'_n$ such that for each $i \in [n]$, if $i \in X$, then $\sigma'_i = \sigma'_i$, otherwise, $\sigma'_i = 0$,*
- *the* letter-preserving $X$-class string *of $w$, denoted by $\mathsf{lcstr}_X(w)$, is defined as the word $(\sigma_1, b_1) \dots (\sigma_n, d_n)$ such that for each $i \in [n]$, if $i \in X$, then $b_i = 1$, otherwise, $b_i = 0$.*

*Example 6.* Suppose $w = (a,1)(b,2)(a,2)(b,1)$ and $X = \{1, 4\}$. Then $\mathsf{cstr}_X(w) = w|_X = ab$, $\mathsf{pcstr}_X(w) = a00b$, and $\mathsf{lcstr}_X(w) = (a,1)(b,0)(a,0)(b,1)$.

**Definition 7 (Data automata).** *A* data automaton *(DA) $\mathcal{D}$ is a tuple $(\mathcal{A}, \mathcal{B})$ s.t. $\mathcal{A} = (Q_1, \Sigma \times \{\bot, \top\}, \Gamma, q_{1,0}, \delta_1, F_1)$ is a nondeterministic letter-to-letter transducer over finite words from the alphabet $\Sigma \times \{\bot, \top\}$ to some output alphabet $\Gamma$, and $\mathcal{B} = (Q_2, \Gamma, q_{2,0}, \delta_2, F_2)$, called the class condition, is a finite-state automaton over $\Gamma$.*

*Semantics of DAs.* We first introduce the concept of class strings. Let $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ be a DA and $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ be a data word. Then $w$ is accepted by $\mathcal{D}$ if over $\mathsf{prof}(w)$, the transducer $\mathcal{A}$ produces a word $\gamma_1 \dots \gamma_n$ over the alphabet $\Gamma$, such that for each class $X$ of $w' = (\gamma_1, d_1) \dots (\gamma_n, d_n)$, $\mathsf{cstr}_X(w')$, the $X$-class string of $w'$, is accepted by $\mathcal{B}$. Let $\mathcal{L}(\mathcal{D})$ denote the set of data words accepted by $\mathcal{D}$.

*Example 7.* Let $\Sigma = \{a\}$. Then the data language comprising the "data words where at least one data value occurs twice" is accepted by the data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ (see Fig. 2, where $(a, \bot)/\#$ denotes the input and output letter are $(a, \bot)$ and $\#$, similarly for $(a, \top)/\#$, and so on), where

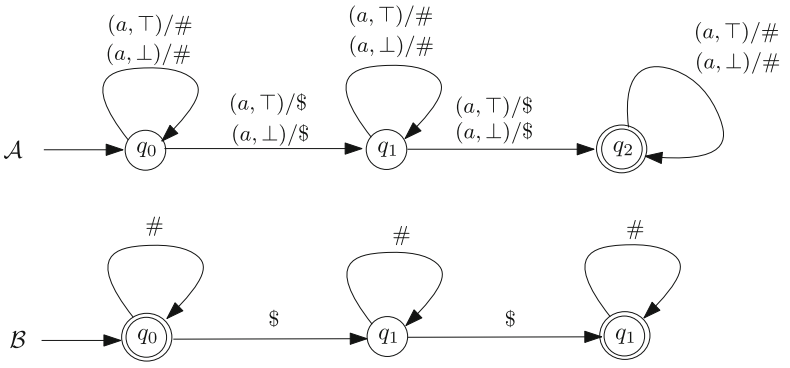



**Fig. 2.** An example of data automata

– $\mathcal{A}$, upon reading $\mathsf{prof}(w)$ for a data word $w$, guesses two positions, relabels the two positions by \$, and relabels all the other positions by #,
– and $\mathcal{B}$ accepts the language $\#^*\$\#^*\$\#^* \cup \#^*$.

Let $w = (a,1)(a,2)(a,3)(a,1)$. Then $\mathcal{A}$ produces a word \$\#\#\$ on $\mathsf{prof}(w)$. Since the three class strings of $w' = (\$,1)(\#,2)(\#,3)(\$,1)$, that is, \$\$, #, and #, are accepted by $\mathcal{B}$, it follows that $w$ is accepted by $\mathcal{D}$.
On the other hand, let $\Sigma = \{a,b\}$, then the data language comprising the data words $w$ such that "for each occurrence of $a$, there is an occurrence of $b$ in the right with the same data value" can be accepted by the data automaton $\mathcal{D}' = (\mathcal{A}', \mathcal{B}')$, where

– $\mathcal{A}'$ is the transducer that outputs $a$ (resp. $b$) when reading $(a,\bot)$ or $(a,\top)$ (resp. $(b,\bot)$ or $(b,\top)$),
– and $\mathcal{B}'$ is the finite-state automaton accepting $a^*b$.

Let $w = (a,1)(a,2)(b,2)(a,1)(b,1)$. Then $\mathcal{A}'$ outputs $w' = aabab$ on $\mathsf{prof}(w)$. Let $X_1$ and $X_2$ be two classes of $w'' = w$ corresponding to the data value 1 and 2 respectively. Then the $X_1$-class string and $X_2$-class string of $w''$, that is, $\mathsf{prj}(w''|_{X_1}) = aab$ and $\mathsf{prj}(w''|_{X_2}) = ab$, are accepted by $\mathcal{B}'$, it follows that $w$ is accepted by $\mathcal{D}'$.

**Theorem 8** ([BMS+06,BDM+11,BS10]). *The following facts hold for DAs:*

– *DAs are closed under intersection and union, but not under complementation.*
– *DAs are strictly more expressive than NRAs.*
– *The nonemptiness problem of DAs is decidable and has the same complexity as the reachability problem of Petri nets.*

By using data automata, it was shown in [BDM+11] that the satisfiability problem of first-order logic with two variables on data words is decidable, whereas, the satisfiability problem of first-order logic with three variables on data words is undecidable.

**Definition 8 (FO over data words).** *Let* $\mathsf{Vars}$ *denote a countably infinite set of variables. First-order logic over data words (FO[*$+1, <, \sim$*]) comprises the formulae* $\varphi$ *defined by the rules,*

$$\varphi \stackrel{def}{=} x = y \mid x + 1 = y \mid x < y \mid P_\sigma(x) \mid x \sim y \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\ \varphi,$$

*where* $x, y \in \mathsf{Vars}$ *and* $\sigma \in \Sigma$*. Intuitively,* $x \sim y$ *is used to denote the equivalence of data values in two positions represented by* $x, y$*. In addition, FO2[*$+1, <, \sim$*] (resp. FO3[*$+1, <, \sim$*]) is used to denote the fragment of FO[*$+1, <, \sim$*] where only two variables (resp. three variables) can be used.*

*Semantics of FO[*$+1, <, \sim$*].* An FO$[+1, <, \sim]$ formula $\varphi$ is interpreted on a tuple $(w, \theta)$, where $w = (\sigma_1, d_1)\ldots(\sigma_n, d_n)$ is a data word, and $\theta : \mathsf{free}(\varphi) \to [n]$ assigns each free variable of $\varphi$ a position of $w$:

- $(w, \theta) \models x = y$ iff $\theta(x) = \theta(y)$,
- $(w, \theta) \models x + 1 = y$ iff $\theta(x) + 1 = \theta(y)$,
- $(w, \theta) \models x < y$ iff $\theta(x) < \theta(y)$,
- $(w, \theta) \models P_\sigma(x)$ iff $\sigma_{\theta(x)} = \sigma$,
- $(w, \theta) \models x \sim y$ iff $d_{\theta(x)} = d_{\theta(y)}$,
- $(w, \theta) \models \neg\varphi$ iff not $(w, \theta) \models \varphi$,
- $(w, \theta) \models \varphi_1 \vee \varphi_2$ iff $(w, \theta) \models \varphi_1$ or $(w, \theta) \models \varphi_2$,
- $(w, \theta) \models \exists x.\ \varphi_1$ iff there is $i' \in [n]$ such that $(w, \theta[i'/x]) \models \varphi_1$, where $\theta[i'/x]$ is the same as $\theta$, except assigning $i'$ to $x$.

Let $\varphi$ be a FO$[+1, <, \sim]$ sentence. Then a data word $w$ satisfies $\varphi$, denoted by $w \models \varphi$, if $(w, \theta) \models \varphi$ for some $\theta$. Let $\mathcal{L}(\varphi)$ denote the set of data words satisfying $\varphi$. The satisfiability problem of FO$[+1, <, \sim]$ is to decide whether $\mathcal{L}(\varphi) \neq \varnothing$, for a given FO$[+1, <, \sim]$ sentence $\varphi$.

*Example 8.* The data language "each data value occurs at most once" can be expressed by the FO2$[+1, <, \sim]$ formula $\varphi = \forall x.\ \forall y.\ (x < y \rightarrow \neg\, x \sim y)$.

**Theorem 9** ([BMS+06,BDM+11]). *The following facts hold for FO$[+1, <, \sim]$:*

- *The satisfiability problem of FO3$[+1, <, \sim]$ is undecidable.*
- *The satisfiability problem of FO2$[+1, <, \sim]$ is decidable.*

The decidability of FO2$[+1, <, \sim]$ is proved by a reduction to the nonemptiness problem of data automata, that is, for each FO2$[+1, <, \sim]$ formula $\varphi$, a data automaton $\mathcal{D}_\varphi$ can be constructed such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{D}_\varphi)$.

In [ACW12], Alur et al. considered a variant of data automata, called extended data automata, defined as follows.

**Definition 9 (Extended data automata).** *An* extended data automaton *(EDA) $\mathcal{D}$ is a tuple $(\mathcal{A}, \mathcal{B})$ s.t. $\mathcal{A} = (Q_1, \Sigma \times \{\bot, \top\}, \Gamma, q_{1,0}, \delta_1, F_1)$ is a nondeterministic letter-to-letter transducer over finite words from the alphabet $\Sigma$ to some output alphabet $\Gamma$, and $\mathcal{B} = (Q_2, \Gamma \cup \{0\}, q_{2,0}, \delta_2, F_2)$ is a finite-state automaton over $\Gamma \cup \{0\}$ such that $0 \notin \Gamma$.*

*Semantics of EDAs.* The semantics of EDAs is defined similarly as that of DAs, with $\mathsf{cstr}_X(w')$ replaced by $\mathsf{pcstr}_X(w')$.

It turns out that the expressibility of EDAs is the same as that of DAs.

**Theorem 10** ([ACW12]). *EDAs are expressively equivalent to DAs.*

Since it is a famous open problem whether the reachability of Petri nets can be decided with elementary complexity, it is also open whether the nonemptiness of data automata can be decided in elementary time. In order to lower the complexity, weaker versions of data automata were introduced. Kara et al. introduced weak data automata (WDA) in [KST12] and showed that the nonemptiness problem of WDAs can be decided in 2NEXPTIME (nondeterministic double exponential time). Later on, Wu introduced commutative data automata (CDA), which are strictly more expressive than WDAs, showed that the nonemptiness problem of CDAs can be solved in 3NEXPTIME (nondeterministic triple exponential time) [Wu12].

**Definition 10 (Weak data automata).** *A* weak data automaton *(WDA) is a tuple* $(\mathcal{A}, \mathcal{C})$ *such that* $\mathcal{A} = (Q, \Sigma \times \{\bot, \top\}, \Gamma, \delta, q_0, F)$ *is a letter-to-letter transducer and* $\mathcal{C}$ *is a class condition specified by a collection of*

- *key constraints of the form* $\mathsf{Key}(\gamma)$ *(where* $\gamma \in \Gamma$*), interpreted as "every two* $\gamma$*-positions have different data values",*
- *inclusion constraints of the form* $D(\gamma) \subseteq \bigcup_{\gamma' \in R} D(\gamma')$ *(where* $\gamma \in \Gamma, R \subseteq \Gamma$*), interpreted as "for every data value occurring in a* $\gamma$*-position, there is* $\gamma' \in R$ *such that the data value also occurs in a* $\gamma'$*-position",*
- *and denial constraints of the form* $D(\gamma) \cap D(\gamma') = \varnothing$ *(where* $\gamma, \gamma' \in \Gamma$*), interpreted as "no data value occurs in both a* $\gamma$*-position and a* $\gamma'$*-position".*

*Semantics of WDAs.* A data word $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ is *accepted* by a WDA $\mathcal{D} = (\mathcal{A}, \mathcal{C})$ iff there is an accepting run of $\mathcal{A}$ over $\mathsf{prof}(w)$ which produces a word $\gamma_1 \dots \gamma_n$ such that the data word $w' = (\gamma_1, d_1) \dots (\gamma_n, d_n)$ satisfies all the constraints in $\mathcal{C}$, where the satisfaction of the constraints on $w'$ is defined as follows:

- $w'$ satisfies $\mathsf{Key}(\gamma)$ iff for every pair of positions $i, j \in [n]$ such that $i \neq j$ and $\gamma_i = \gamma_j = \gamma$, it holds that $d_i \neq d_j$,
- $w'$ satisfies $D(\gamma) \subseteq \bigcup_{\gamma' \in R} D(\gamma')$ iff for each $i \in [n]$ such that $\gamma_i = \gamma$, there is $j \in [n]$ such that $\gamma_j \in R$ and $d_i = d_j$.
- $w'$ satisfies $D(\gamma) \cap D(\gamma') = \varnothing$ iff for every pair of positions $i, j \in [n]$ such that $\gamma_i = \gamma$ and $\gamma_j = \gamma'$, it holds that $d_i \neq d_j$.

Let $L$ be a language over the alphabet $\Sigma$. Then $L$ is *commutative* iff for every $\sigma_1, \sigma_2 \in \Sigma$ and $u, v \in \Sigma^*$, $u\sigma_1\sigma_2 v \in L$ iff $u\sigma_2\sigma_1 v \in L$. Commutative regular languages have a characterisation in quantifier-free simple Presburger formulae defined in the following: *Quantifier-free simple Presburger formulae* (QFSP formulae) over a variable set $X$ are Boolean combinations of atomic formulae of the form $x_1 + \dots + x_m \leq c$, or $x_1 + \dots + x_m \geq c$, or $x_1 + \dots + x_m = c$, or $x_1 + \dots + x_m \equiv r \bmod p$, where $x_1, \dots, x_m \in X$, $c, r, p \in \mathbb{N}$, $p \geq 2$, and $0 \leq r < p$.

Suppose $\Sigma = \{\sigma_1, \dots, \sigma_k\}$ and $v \in \Sigma^*$. The *Parikh image* of $v$, denoted by $\mathsf{Parikh}(v)$, is a $k$-tuple $(\#_{\sigma_1}(v), \dots, \#_{\sigma_k}(v))$, where for each $i : 1 \leq i \leq k$, $\#_{\sigma_i}(v)$ is the number of occurrences of $\sigma_i$ in $v$. Let $V_\Sigma = \{x_{\sigma_1}, \dots, x_{\sigma_k}\}$ and $\varphi$ be an QFSP formula over $V_\Sigma$. The word $v$ is said to satisfy $\varphi$, denoted by $v \models \varphi$, iff $\varphi[\mathsf{Parikh}(v)/V_\Sigma]$ holds, where $\varphi[\mathsf{Parikh}(v)/V_\Sigma]$ denotes the formula obtained from $\varphi$ by replacing each $x_{\sigma_i}$ with $\#_{\sigma_i}(v)$. The *language defined by* $\varphi$, denoted by $\mathcal{L}(\varphi)$, is the set of words $v \in \Sigma^*$ such that $v \models \varphi$.

**Definition 11 (Commutative data automata).** *A* commutative data automaton *(CDA)* $\mathcal{D}$ *is a tuple* $(\mathcal{A}, \varphi)$ *such that* $\mathcal{A} = (Q, \Sigma \times \{\bot, \top\}, \Gamma, \delta, q_0, F)$ *is a letter-to-letter transducer and* $\varphi$ *is a QFSP formula over the variable set* $V_\Gamma$*, where* $V_\Gamma = \{x_\gamma \mid \gamma \in \Gamma\}$.

*Semantics of CDAs.* A data word $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ is *accepted* by a CDA $\mathcal{D} = (\mathcal{A}, \varphi)$ iff there is an accepting run of $\mathcal{A}$ over $\mathsf{prof}(w)$ which produces a word $\gamma_1 \dots \gamma_n$ such that the data word $w' = (\gamma_1, d_1) \dots (\gamma_n, d_n)$ satisfies that for each class $X$ of $w'$, $\mathsf{cstr}_X(w') \models \varphi$.

**Theorem 11 ([KST12, Wu12]).** *The following results hold for WDAs and CDAs:*

- *DAs are strictly more expressive than CDAs, which is in turn strictly more expressive than WDAs.*
- *WDAs and CDAs are closed under union and intersection, but not under complementation.*
- *The nonemptiness problem of WDAs and CDAs can be decided in 2NEXPTIME and 3NEXPTIME respectively.*

An extension of data automata, called class automata, were introduced, in order to capture the expressiveness of XPath with data comparison modalities ([BL12]). Class automata in [BL12] were defined on data trees, here for simplicity, we restrict our attention to class automata on data words.

**Definition 12 (Class automata).** *A class automaton (CA) $\mathcal{C}$ is a tuple $(\mathcal{A}, \mathcal{B})$ such that $\mathcal{A} = (Q, \Sigma \times \{\bot, \top\}, \Gamma, \delta, q_0, F)$ is a letter-to-letter transducer and $\mathcal{B} = (Q_2, \Gamma \times \{0, 1\}, q_{2,0}, \delta_2, F_2)$ is a finite-state automaton over $\Gamma \times \{0, 1\}$.*

*Semantics of CAs.* The semantics of CAs is defined similarly as that of DAs, with $\mathsf{cstr}_X(w')$ replaced by $\mathsf{lcstr}_X(w')$.

It turns out class automata are expressive enough to simulate two-counter machines and its nonemptiness problem is undecidable.

**Theorem 12 ([BL12]).** *The nonemptiness problem of CAs is undecidable.*

In [Wu11], Wu proposed a restriction of class automata, called class automata with *priority class condition* (PCA), and showed that PCAs strictly extend data automata, and at the same time have a decidable nonemptiness problem.

*Further Reading.* Manuel and Ramanujam proposed class counting automata, which includes a counter for each data value occurring in a data word, and showed that the nonemptiness problem of class counting automata is EXPSPACE-complete [MR11a]. The model is in a style similar to class memory automata. In addition, Tan studied data trees over a linearly ordered infinite data domain and proposed ordered-data tree automata, which is in the same flavour as data automata, and showed their nonemptiness problem can be solved in 3NEXPTIME [Tan14]. To solve the satisfiability problem of an extension of LTL over multi-attributed data words (i.e. data words where a tuple of data values, instead of a single one, occur in each position), Decker et al. introduced nested data automata (NDA) and showed that although the nonemptiness of NDAs is undecidable in general, the nonemptiness problems of two natural sub-models are decidable [DHLT14].

## 5 Pebble Automata

Pebble automata were introduced by Neven et al. in [NSV01,NSV04]. In contrast to register automata which are finite state machines equipped with *registers*, pebble automata are finite state machines equipped with a finite number of *pebbles*. These pebbles are placed on, or lifted from, the input data word in a *stack* discipline, i.e., first in last out, with the purpose of marking positions of the data word. One pebble can only mark one position and the most recently placed pebble serves as the head of the automaton.

As we are dealing with two-way automata here, as a convention, we delimit the input data word by two special symbols $\{\triangleleft, \triangleright\} \notin \mathbb{D}$ for the left and the right hand of the data word. Hence, automata always work on the extended data word of the form $\triangleright w \triangleleft$. The positions of $\triangleright$ and $\triangleleft$ are 0 and $|w|+1$, respectively. (Recall that $|w|$ denotes the length of the data word $w$.)

**Definition 13 (Pebble automata, [NSV04]).** *A* nondeterministic two-way *$k$-pebble automaton (2N-kPA) $\mathcal{A}$ is a tuple $(Q, \Sigma, q_0, \delta, F)$ where:*

- *$Q$ is a finite set of* states*,*
- *$\Sigma$ is a finite* alphabet*,*
- *$q_0 \in Q$ is the* initial *state,*
- *$F \subseteq Q$ is the set of* final *states, and*
- *$\delta$ is a finite set of* transitions *of the form $\alpha \to \beta$ where:*
  - *$\alpha$ is of the form $(i, \sigma, V, q)$, where $i \in [k]$, $\sigma \in \Sigma$, $V \subseteq [i-1]$; and*
  - *$\beta$ is of the form $(q, \mathsf{act})$ with $q \in Q$ and*

$$\mathsf{act} \in \{\mathit{stay}, \mathit{left}, \mathit{right}, \mathit{place\text{-}new\text{-}pebble}, \mathit{lift\text{-}current\text{-}pebble}\}.$$

*Semantics of 2N-kPAs.* Given a data word $w = (\sigma_1, d_1) \ldots (\sigma_n, d_n)$, a *configuration* of $\mathcal{A}$ on $w$ is a triple $[i, q, \theta]$ where $i \in [k]$, $q \in Q$, and $\theta : [i] \to [n] \cup \{0, n+1\}$. The function $\theta$ is the *pebble assignment* which defines the positions of the pebbles. (Recall that, as mentioned earlier, we assume an extended data word where position 0 is $\triangleright$ and position $(n+1)$ is $\triangleleft$.) The initial configuration is $\gamma_0 = [1, q_0, \theta_0]$ where $\theta_0(1) = 0$ is the initial pebble assignment. A configuration $(i, q, \theta)$ is *accepting* if $q \in F$.

A transition $(i, \sigma, V, p) \to \beta$ applies to a configuration $[j, q, \theta]$ if the following three conditions hold:

1. $i = j$ and $p = q$;
2. $V = \{l < i \mid d_{\theta(l)} = d_{\theta(i)}\}$;
3. $\sigma_{\theta(i)} = \sigma$

Intuitively, in a configuration $[i, q, \theta]$, pebble $i$ is in control, serving as the head pebble. $(i, \sigma, V, p) \to \beta$ applies to the configuration if pebble $i$ is the current head, $p$ is the current state, $V$ is the set of pebbles that see the same data value as the head pebble, and the current symbol seen by the head pebble is $\sigma$.

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

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

*Strong vs Weak PAs.* In the above definition, new pebbles are placed at the position of the most recent pebble. (Formally, in the definition of $\mathsf{act} = \mathsf{place\text{-}new\text{-}pebble}$, one has $\theta'(i + 1) = \theta'(i) = \theta(i)$.) An alternative would be to place new pebbles at the beginning of the data word. Formally, in the $\mathsf{place\text{-}new\text{-}pebble}$ case, one has $\theta'(i + 1) = 1$, and $\theta'(i) = \theta(i)$. In literature, the former is often referred to as *weak* PAs, and the latter is referred to as *strong* (a.k.a., ordinary) PAs. While the choice makes no difference in the two-way case (as defined here), it is significant in the *one-way* case (i.e., when $\mathsf{act} = \mathsf{left}$ is not allowed). For instance, it is known that one-way non-deterministic weak PAs are weaker than one-way strong PAs, see [NSV04, Theorem 4.5].

*Alternating PAs.* As in Sect. 3, we can define the *alternating* version of PAs. Alternating automata additionally have a set $U \subseteq Q$ of *universal* states. The sets from $Q \setminus U$ are called *existential*. (Clearly, if $U = \varnothing$, then we have a nondeterministic PA as in Definition 13.)

*Acceptance.* The acceptance criteria are based on the notion of *leads to acceptance* as follows. For every configuration $\gamma = [i, q, \theta]$,

– if $q \in F$, then $\gamma$ leads to acceptance;
– if $q \in U$, then $\gamma$ leads to acceptance if and only if for *all* configurations $\gamma'$ such that $\gamma \vdash \gamma'$, $\gamma'$ leads to acceptance;
– if $q \notin F \cup U$, then $\gamma$ leads to acceptance if and only if there is at least one configuration $\gamma'$ such that $\gamma \vdash \gamma'$ and $\gamma'$ leads to acceptance.

A data word $w$ is said to be *accepted* by $\mathcal{A}$ if $\gamma_0$ leads to acceptance. Let $\mathcal{L}(\mathcal{A})$ denote the set of data words accepted by $\mathcal{A}$. We say that a data language $L$ is defined by a PA $\mathcal{A}$ if $\mathcal{L}(\mathcal{A}) = L$.

*Remark 1.* In Definition 13, we adopt the pebble numbering from [NSV04], in which the pebbles placed on the input word are numbered from 1 to $i$. However, in some literature, for instance, in [Tan10, BSSS06], the pebble numbering is used differently—it is from $k$ to $i$. The reason for this reverse numbering is that it allows to view the computation between placing and lifting pebble $i$ as a computation of an $(i - 1)$-pebble automaton.

*Example 9.* To show how a PA works, we consider the data language $L$ comprising the data words where at least one data value occurs twice. The reader should be easily convinced that $L$ is accepted by the 1N-2PA $\mathcal{A} = (Q, q_1, F, \delta)$, where

- $Q = \{q_1, q_2, q_{\rightarrow}, q_{\text{acc}}\}$;
- $F = \{q_{\text{acc}}\}$;
- $\delta$ consists of the following transitions:
  1. $(1, \sigma, \varnothing, q_1) \rightarrow (q_1, \mathsf{right})$
  2. $(1, \sigma, \varnothing, q_1) \rightarrow (q_{\rightarrow}, \mathsf{place\text{-}new\text{-}pebble})$
  3. $(2, \sigma, \{1\}, q_{\rightarrow}) \rightarrow (q_2, \mathsf{right})$
  4. $(2, \sigma, \varnothing, q_2) \rightarrow (q_2, \mathsf{right})$
  5. $(2, \sigma, \{1\}, q_2) \rightarrow (q_{\text{acc}}, \mathsf{stay})$

Some sub-classes of PAs can be defined in a standard way. A PA is *deterministic*, if in each configuration at most one transition rule applies. And, as mentioned before, if there are no left-transitions, then the PA is one-way. For the automata models we consider "control" as deterministic (D), non-deterministic (N), or alternating (A), as well as the one-way and two-way variants. We denote these automata models by dC-kPA where $d \in \{1, 2\}$, C $= \{$D, N, A$\}$, and $k \in \mathbb{N} \setminus \{0\}$. Here, 1 and 2 stand for one- and two-way, respectively, D, N, and A stand for deterministic, non-deterministic, and alternating, and $k$ stands for the number of pebbles. In addition, when necessary we also write S for Strong and W for Weak, which are specific to one-way PAs.

*Expressiveness of PA Models.* As we have introduced a variety of pebble automata, it is natural to ask their expressiveness. A class $C_1$ of PAs is strictly stronger than the class $C_2$ of PAs is for all data languages $L$ accepted by a PA in $C_2$, $L$ can be accepted by a PA in $C_1$ and there exists at least one language which can be accepted by a PA in $C_1$, but not by any PA in $C_1$. Figure 3 summarises the known results, where, all classes of PAs in the same box are equivalent in expressiveness, while $\rightarrow$ means the source class is at least as expressive as the target, and the arrow decorated by $\neq$ means it is strictly more expressive. The only class which was not addressed in Fig. 3 is strong 1A-PAs, whose relation with other classes does not appear in literature and is to be studied.

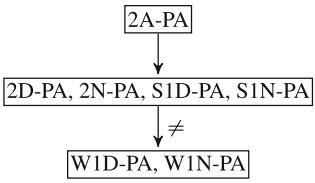



**Fig. 3.** Expressiveness of PAs

### 5.1 (Un)Decidability of Emptiness of PAs

As usual, one of the fundamental problems regarding PAs is the *emptiness problem*, which is to determine, given a PA $\mathcal{A}$, whether $L(\mathcal{A}) = \varnothing$. It was shown in [NSV04] that this problem is generally undecidable, even for weak 1D-PAs. The

intuition is, when a PA lifts 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 data word. This result is very strong in the sense that it implies that almost all standard decision problems are undecidable for virtually all classes of pebble automata (cf. Fig. 3).

More technically, [NSV04] gave a reduction from the PCP to the emptiness of weak 1D-5PAs. Tan observed that the proof can be adapted to weak 1D-3PAs, yielding an even stronger result. In [Tan10, KT10], a tighter boundary between decidability and undecidability was drawn in terms of the number of pebbles. In summary,

**Theorem 13 ([NSV04, Tan10, KT10]).** *The following facts hold for pebble automata:*

- *The nonemptiness problem for strong 2N-2PAs is undecidable.*
- *The nonemptiness problem for weak 1D-3PAs is undecidable.*
- *The nonemptiness problem for weak 1D-2PAs is decidable, but is not primitive recursive.*

*Top View Weak PAs.* Theorem 13 suggests that PAs are in general highly undecidable. To mitigate this, Tan [Tan10] proposed a subclass of pebble automata, the *top view weak* pebble automata. Roughly speaking, top view weak PAs are weak one-way PAs where the equality test is performed only between the data values seen by the *two most recently placed pebbles*. That is, if pebble $i$ is the head pebble, then it can only compare the data value it reads with the data value read by pebble $(i-1)$. It is not allowed to compare its data value with those read by pebble $1, \ldots, (i-2)$. Formally,

**Definition 14 (Top view weak PA, [Tan10]).** *A top view (weak) $k$-PA is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ where $Q, q_0, F$ are defined as before, and $\delta$ consists of transitions of the form $(i, \sigma, V, q) \to (q', \mathsf{act})$, where $V$ is either $\varnothing$ or $\{i-1\}$ and $\mathsf{act} \neq \mathsf{left}$.*

A transition $(i, \sigma, V, q) \to (q', \mathtt{act})$ applies to a configuration $[j, q, \theta]$ if

1. $i = j$ and $p = q$;
2. $V = \begin{cases} \varnothing & \text{if } d_{\theta(i-1)} \neq d_{\theta(i)} \\ \{i-1\} & \text{if } d_{\theta(i-1)} = d_{\theta(i)} \end{cases}$
3. $\sigma_{\theta(i)} = \sigma$

Note that evidently top view weak 2-PAs and weak 2-PAs are the same.

**Theorem 14 ([Tan10]).** *For every top view weak $k$-PA $\mathcal{A}$, there is a (one-way) $ARA_1$ $\mathcal{A}'$ such that they accept the same language. Moreover, the construction of $\mathcal{A}'$ is effective.*

The following result follows from Theorem 14 and Theorem 3.

**Corollary 1.** *The emptiness problem for* top view weak *$k$-PAs is decidable.*

It turns out that top view weak PAs admits many nice properties [Tan13]:

- Expressiveness: it is shown that for every $\text{LTL}_1^{\downarrow}$ formula $\psi$, there exists a weak $k$-PA $\mathcal{A}_\psi$, such that $\mathcal{L}(A_\psi) = \mathcal{L}(\psi)$. It turns out that the automaton $\mathcal{A}_\psi$ is a top view weak $k$-PA. Thus, the class of languages accepted by top view weak $k$-PAs contains the languages definable by LTL with one freeze quantifier.
- Decidability: The emptiness problem is decidable.
- Efficiency: The membership problem, that is, testing whether a given data word of length $n$ is accepted by a deterministic top view weak $k$-PA can be solved in $\mathcal{O}(n^k)$ time.
- Closure properties: Top view weak $k$-PAs are closed under all boolean operations.
- Robustness: Alternation and non-determinism do *not* add expressive power to top view weak $k$-PAs.

Tan [Tan10] observed that the finiteness of the number of pebbles for top view weak PAs is not necessary. He defined top view weak PAs with unbounded number of pebbles, i.e., top view weak unbounded PAs. It is straightforward to show that 1-way deterministic 1-RAs can be simulated by top view weak unbounded PAs. (Each time the register automaton changes the content of the register, the top view weak unbounded PAs places a new pebble.) Furthermore, top view weak unbounded PAs can be simulated by $\text{ARA}_1$'s (1-way alternating one-register automata), similar to Theorem 14. Thus, the emptiness problem for top view unbounded weak PAs is still decidable.

*Further Reading.* Tan [Tan13] used graph reachability problem to investigate the strict hierarchy of pebble automata based on the number of pebbles and the comparison of the expressiveness of pebble automata with the other formalisms over infinite alphabets. [BSSS06] studied pebble tree-walking automata on trees.

## 6 Variable Automata and LTL with Data Variable Quantifications

Another idea to deal with data values from an infinite data domain is to use *logical variables* to represent data values. The differences between logical variables and registers are as follows: While logical variables and registers are both used to represent the data values from an infinite data domain, logical variables are *declarative* in the sense that they cannot be updated, but can be existentially or universally quantified, on the other hand, registers are *imperative* in the sense that they can be updated, but cannot be quantified.

In this section, we introduce variable automata, LTL with data variable quantifications, and its variant, indexed temporal logics, where the data values are interpreted as process identifiers.

Variable automata were proposed by Grumberg et al. as a natural extension of NFAs to infinite alphabets [GKS10].

**Definition 15 (Variable automata).** *Let $\Sigma$ be a finite alphabet and $X \cup \{y\}$ be a finite set of variables, where $X$ is a set of bound variables, and $y \notin X$ is a free variable. A variable automaton (VA) $\mathcal{A}$ is an NFA $(Q, \Sigma \times (X \cup \{y\}), q_0, \delta, F)$.*

*Semantics of VAs.* Suppose $\mathcal{A} = (Q, \Sigma \times (X \cup \{y\}), q_0, \delta, F)$ is a VA and $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ is a data word. A *run* of $\mathcal{A}$ on $w$ is a sequence of transitions $q_0 \xrightarrow{(\sigma_1, z_1)} q_1 \dots q_{n-1} \xrightarrow{(\sigma_n, z_n)} q_n$ such that

- for each $i \in [n]$, $(q_{i-1}, (\sigma_i, z_i), q_i) \in \delta$,
- for every $i, j \in [n]$ such that $z_i, z_j \in X$, it holds that $z_i = z_j$ iff $d_i = d_j$,
- for each $i, j \in [n]$ such that $z_i \in X$ and $z_j = y$, it holds that $d_i \neq d_j$.

A run of $\mathcal{A}$ on $w$ is accepting if $q_n \in F$. Let $\mathcal{L}(\mathcal{A})$ denote the set of data words accepted by $\mathcal{A}$.

*Example 10.* Let $\Sigma = \{a, b\}$ and $L$ be the data language comprising the data words $w = (a, d_1)(b, d_2) \dots (b, d_{n-1})(a, d_n)$ such that $d_1 = d_n$ and for each $i : 1 < i < n$, $d_i \neq d_1$. Then $L$ can be defined the VA $\mathcal{A}$ illustrated in Fig. 4.

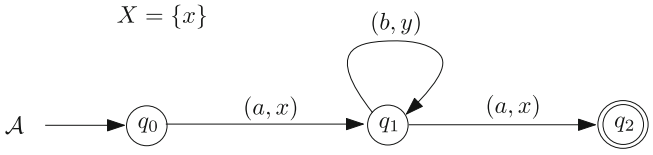



**Fig. 4.** An example of VA

**Theorem 15 ([GKS10]).** *The following results hold for variable automata:*

- *VAs are closed under union, but not closed under intersection or complementation.*
- *VAs and NRAs are incomparable with respect to the expressive power.*
- *The nonemptiness problem of VAs is NL-complete, the universality and language inclusion problems of VAs are undecidable.*

Mens and Rahonis consisdered variable tree automata (VTA) in [MR11b]. They showed VTAs have similar theoretical properties as VAs.

LTL with data variable quantifications (VLTL) is obtained by extending LTL with existential and universal quantifications on data variables. VLTL was first considered by Grumberg et al. in [GKS12, GKS13, GKS14]. Later on, Song and Wu did an extensive investigation on the decision problems of different fragments of VLTL in [SW14, SW16].

**Definition 16 (LTL with data variable quantifications).** *Let $\mathcal{X}$ be a countable set of variables. Then LTL with data variable quantifications (denoted by VLTL) is defined by the following rules:*

$$\varphi \stackrel{def}{=} \sigma \mid \mathsf{val}(x) \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi \ \mathsf{U} \ \varphi \mid \exists x.\ \varphi,$$

*where $\sigma \in \Sigma$ and $x \in \mathcal{X}$.*

The set of free variables of VLTL formula $\varphi$, denoted by $\mathsf{free}(\varphi)$, can be defined in a standard way as first-order logics. A VLTL formula $\varphi$ is *closed* if $\mathsf{free}(\varphi) = \varnothing$.

*Semantics of VLTL.* VLTL formulae $\varphi$ are interpreted on a tuple $(w, i, \theta)$, where $w = (\sigma_1, d_1) \dots (\sigma_n, d_n)$ is a data word, $i$ is a position of the data word, and $\theta : \mathsf{free}(\varphi) \to \mathbb{D}$ assigns each variable from $\mathsf{free}(\varphi)$ a data value:

- $(w, i, \theta) \models \sigma$ iff $\sigma_i = \sigma$,
- $(w, i, \theta) \models \mathsf{val}(x)$ iff $d_i = \theta(x)$,
- $(w, i, \theta) \models \neg\varphi$ iff not $(w, i, \theta) \models \varphi$,
- $(w, i, \theta) \models \varphi_1 \vee \varphi_2$ iff $(w, i, \theta) \models \varphi_1$ or $(w, i, \theta) \models \varphi_2$,
- $(w, i, \theta) \models \mathsf{X}\varphi$ iff $i < n$ and $(w, i+1, \theta) \models \varphi$,
- $(w, i, \theta) \models \varphi_1 \ \mathsf{U} \ \varphi_2$ iff there exists $k : i \leq k \leq n$ such that $(w, k, \theta) \models \varphi_2$ and for each $j : i \leq j < k$, $(w, j, \theta) \models \varphi_1$,
- $(w, i, \theta) \models \exists x.\ \varphi$ iff there exists $d \in \mathbb{D}$ such that $(w, i, \theta[d/x]) \models \varphi$, where $\theta[d/x]$ denotes the assignment function that is the same as $\theta$, except that $x$ is assigned with the data value $d$.

Similarly to LTL, we can also define the positive normal form of VLTL. Specifically, VLTL formulae in positive normal form are defined by the following rules,

$$\varphi \stackrel{\text{def}}{=} \sigma \mid \neg\sigma \mid \mathsf{val}(x) \mid \neg\mathsf{val}(x) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \overline{\mathsf{X}}\varphi \mid \varphi \ \mathsf{U} \ \varphi \mid \varphi \ \mathsf{R} \ \varphi \mid \exists x.\ \varphi \mid \forall x.\ \varphi.$$

In the following, we assume that all VLTL formulae are in positive normal form.

We consider the following fragments of VLTL:

- Let $\exists^*$-VLTL denote the fragment of VLTL where no universal quantifiers appear.
- Let NN-$\exists^*$-VLTL denote the fragment of $\exists^*$-VLTL where the existential quantifiers are non-nested, more precisely, the formulae $\varphi$ in $\exists^*$-VLTL such that for each subformula $\exists x.\ \varphi'$ and each subformula of $\exists y.\ \varphi''$ of $\varphi'$, there are no free occurrences of $x$ in $\varphi''$.
- Let VLTL$_{pnf}$ denote the fragment of VLTL where the formulae in prenex normal form, that is, VLTL formulae of the form $\mathcal{Q}_1 x_1.\ \dots \mathcal{Q}_n x_n.\ \varphi$, where $\mathcal{Q}_1, \dots, \mathcal{Q}_n \in \{\exists, \forall\}$ and $\varphi$ is a quantifier-free VLTL formula. Moreover, for a quantifier prefix $\Theta = \mathcal{Q}_1 \dots \mathcal{Q}_k \in \{\exists, \forall\}^+$, let $\Theta$-VLTL$_{pnf}$ denote the fragment of VLTL$_{pnf}$ where all the formulae are of the form $\mathcal{Q}_1 x_1.\ \dots \mathcal{Q}_k x_k.\ \varphi$, where $\varphi$ is a quantifier-free VLTL formula.
- Let $\forall$-VLTL$_{pnf}^{gdlt}$ denote the set of $\forall$-VLTL$_{pnf}$ formulae $\forall x.\ \psi$ such that all the occurrences of $\sigma$ and $\neg\sigma$ in $\psi$ are *guarded* by the positive occurrences of $\mathsf{val}(x)$. More precisely, $\psi$ is a quantifier-free VLTL formula defined by the following rules,

$$\psi := \sigma \wedge \mathsf{val}(x) \mid \neg(\sigma \wedge \mathsf{val}(x)) \mid \neg\sigma \wedge \mathsf{val}(x) \mid \neg(\neg\sigma \wedge \mathsf{val}(x)) \\ \mathsf{val}(x) \mid \neg\mathsf{val}(x) \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathsf{X}\psi \mid \overline{\mathsf{X}}\psi \mid \psi \ \mathsf{U} \ \psi \mid \psi \ \mathsf{R} \ \psi,$$

where $\sigma \in \Sigma$, $x \in X$, and the superscript "*gdlt*" means "guarded letters". For instance, the formula $\forall x.\ \mathsf{G}[(openFile \wedge \mathsf{val}(x)) \rightarrow \mathsf{XF}(closeFile \wedge \mathsf{val}(x))]$ is in $\forall\text{-VLTL}^{gdlt}_{pnf}$, while the formula

$$\forall x.\ \mathsf{G}[(openFile \wedge \mathsf{val}(x)) \rightarrow (write \wedge \neg\mathsf{val}(x))\ U\ (closeFile \wedge \mathsf{val}(x))]$$

is not, since the occurrence of *write* is not guarded by a positive occurrence of $\mathsf{val}(x)$.

**Theorem 16 ([SW14,SW16]).** *The following results hold for VLTL:*

- *The satisfiability problem of $\exists^*$-VLTL is undecidable.*
- *The satisfiability problem of $\forall$-$VLTL_{pnf}$ is undecidable.*
- *The satisfiability problem of NN-$\exists^*$-VLTL is decidable and non-primitive recursive.*
- *The satisfiability problem of $\forall$-$VLTL^{gdlt}_{pnf}$ is decidable.*

As mentioned before, since process identifiers are a concrete type of data values, indexed linear temporal logic (ILTL) used to specify and reason about parameterized concurrent systems can be seen as variants of VLTL. ILTL was first proposed by German and Sistla in ([SG87,GS92]). They showed that the validity (resp. model checking) problem of the indexed LTL is decidable (resp. undecidable). The differences between ILTL and VLTL are as follows:

- VLTL interpreted over data words where each position carries only one data value or a fixed number of data values, whereas ILTL is interpreted over computation traces in parameterised systems (cf. the semantics of ILTL formulae below).
- While computation traces can also be seen as data words by treating process identifiers as data values, these data words are significantly different than the traditional ones studied before. Namely, each position of these data words carries an *unbounded* number of data values, and all the data values occur in every position.

**Definition 17 (Indexed Linear Temporal Logics).** *Let $AP$ and $AP'$ be the set of global and local atomic propositions. The formulae of indexed linear temporal logic (ILTL) are defined by the following rules,*

$$\varphi \stackrel{def}{=} \mathsf{true} \mid \mathsf{false} \mid p \mid \neg p \mid p'(x) \mid \neg p'(x) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi\ \mathsf{U}\ \varphi \mid \varphi\ \mathsf{R}\ \varphi \mid \exists x.\ \varphi \mid \forall x.\ \varphi,$$

*where $p \in AP$, $p' \in AP'$, and $x \in X$.*

Let $\mathsf{free}(\varphi)$ denote the set of free variables occurring in $\varphi$. An ILTL formula containing no free variables is called a *closed* ILTL formula. For an ILTL formula $\varphi$, let $\neg\varphi$ denote its complement (negation), and let $\overline{\varphi}$ denote the *positive normal form* of $\neg\varphi$, that is obtained by pushing the negation inside of operators. For instance, if $\varphi = \exists x.\ \mathsf{F}p'(x)$, then $\overline{\varphi} = \forall x.\ \mathsf{G}\neg p'(x)$.

*Semantics of ILTL.* ILTL formulae are interpreted over computation traces of parameterised systems. Let $\mathcal{I}$ be an infinite set of process identifiers. A *computation trace* over $AP \cup AP'$ is a tuple $trc = (\alpha, I, (\beta_i)_{i \in I})$, where $\alpha \in (2^{AP})^\omega$ is an $\omega$-sequence of valuations over the global atomic propositions from $AP$, $I \subseteq \mathcal{I}$ is a *finite* set of process identifiers, and for each $i \in I$, $\beta_i \in (2^{AP'})^\omega$ is a local computation trace, i.e. an $\omega$-sequence of valuations over the local atomic propositions from $AP'$.

Let $\varphi$ be an ILTL formula, $trc = (\alpha, I, (\beta_i)_{i \in I})$ be a computation trace, $\theta : \mathsf{free}(\varphi) \to I$ be an assignment of the process identifiers (from $I$) to the free variables in $\varphi$, and $n \in \mathbb{N}$. Then $(trc, \theta, n)$ satisfies $\varphi$, denoted by $(trc, \theta, n) \models \varphi$, is defined as follows:

- $(trc, \theta, n) \models p$ (resp. $\neg p$) if $p \in \alpha[n]$ (resp. $p \notin \alpha[n]$),
- $(trc, \theta, n) \models p'(x)$ (resp. $\neg p'(x)$) if $p' \in \beta_{\theta(x)}[n]$ (resp. $p' \notin \beta_{\theta(x)}[n]$),
- $(trc, \theta, n) \models \exists x.\ \varphi_1$ if there is $i \in I$ such that $(trc, \theta[i/x], n) \models \varphi_1$, where $\theta[i/x]$ is the same as $\theta$, except for assigning $i$ to $x$,
- $(trc, \theta, n) \models \forall x.\ \varphi_1$ if for each $i \in I$, $(trc, \theta[i/x], n) \models \varphi_1$,
- $(trc, \theta, n) \models \varphi_1 \vee \varphi_2$ if $(trc, \theta, n) \models \varphi_1$ or $(trc, \theta, n) \models \varphi_2$,
- $(trc, \theta, n) \models \varphi_1 \wedge \varphi_2$ if $(trc, \theta, n) \models \varphi_1$ and $(trc, \theta, n) \models \varphi_2$,
- $(trc, \theta, n) \models \mathsf{X}\varphi$ if $(trc, \theta, n+1) \models \varphi$,
- $(trc, \theta, n) \models \varphi_1 \ \mathsf{U}\ \varphi_2$ if there is $m \geq n$ s.t. $(trc, \theta, m) \models \varphi_2$, and for all $l : n \leq l < m$, $(trc, \theta, l) \models \varphi_1$,
- $(trc, \theta, n) \models \varphi_1 \ \mathsf{R}\ \varphi_2$ if either for all $m \geq n$, $(trc, \theta, m) \models \varphi_2$, or there is $m \geq n$ s.t. $(trc, \theta, m) \models \varphi_1$, and for all $l : n \leq l \leq m$, $(trc, \theta, l) \models \varphi_2$.

Note that if $\varphi$ is a closed ILTL formula, then $\theta$ has an empty domain and thus is omitted. Namely we simply write $(trc, n) \models \varphi$. In addition, for a closed ILTL formula $\varphi$, we use $trc \models \varphi$ to abbreviate $(trc, 0) \models \varphi$. For a closed ILTL formula $\varphi$, let $\mathcal{L}(\varphi)$ denote the set of computation traces $trc$ such that $trc \models \varphi$. The *satisfiability* problem of ILTL is defined as follows: Given a closed ILTL formula $\varphi$, decide whether $\mathcal{L}(\varphi)$ is empty.

We shall consider the following fragments of ILTL with abbreviations:

- $\text{ILTL}_{pnf}$ denotes the fragment of ILTL where formulae are in *prenex normal form*, that is $\{\forall, \exists\}$ quantifications appear only at the beginning of the formula. In particular, let $\Theta \subseteq \{\exists, \forall\}^*$. Then $\Theta$-$\text{ILTL}_{pnf}$ denotes the fragment of $\text{ILTL}_{pnf}$ where the quantifier prefixes belong to $\Theta$.
- NN-ILTL denotes the fragment of ILTL where the quantifiers are *not* nested, that is, for each formula $\mathcal{Q}_1 x.\varphi_1$ such that $\mathcal{Q}_2 y.\varphi_2$ is a subformula of $\varphi_1$, it holds that $x$ is not a free variable of $\varphi_2$, where $\mathcal{Q}_1, \mathcal{Q}_2 \in \{\forall, \exists\}$.
- $\text{ILTL}(\mathcal{O})$ for $\mathcal{O} \subseteq \{\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}, \mathsf{R}\}$ denotes the fragment of ILTL where only temporal operators from $\mathcal{O}$ are used. Moreover, we use $\text{ILTL}\backslash\mathsf{X}$ as an abbreviation of $\text{ILTL}(\mathsf{U}, \mathsf{R})$, where the $\mathsf{X}$ operator is forbidden.
- $\text{ILTL}^{locap}$ denotes the fragment of ILTL where there are no global atomic propositions, that is, $AP = \varnothing$.

These notations might be combined to define more (refined) fragments, e.g. the logic $(\text{ILTL}(\mathsf{F}, \mathsf{G}))_{pnf}$ denotes the fragment of $\text{ILTL}_{pnf}$ where only temporal operators $\mathsf{F}$ and $\mathsf{G}$ are used.

**Theorem 17.** *The following results hold for ILTL:*

- *The satisfiability problems of* $\forall\exists$*-ILTL*$_{pnf}$ *and* $\exists\forall\exists$*-ILTL*$^{locap}_{pnf}$ *are undecidable.*
- *The satisfiability problems of* $\exists^*\forall^*$*-ILTL*$_{pnf}$ *and* $\exists^*\forall^*$*-(ILTL* $\setminus\mathsf{X})_{pnf}$ *are EXPSPACE-complete, and the satisfiability problem of* $\exists^*\forall^*$*-(ILTL*$(\mathsf{F},\mathsf{G}))_{pnf}$ *is NEXPTIME-complete.*
- *The satisfiability problems of NN-ILTL, NN-ILTL*$(\mathsf{X},\mathsf{F},\mathsf{G})$*, NN-ILTL*$\setminus\mathsf{X}$*, and NN-ILTL*$(\mathsf{F},\mathsf{G})$ *are EXPSPACE-complete.*

## 7 Symbolic Automata and Transducers

In this section, we introduce symbolic automata and transducers, another line of work to reason about data values from an infinite domain. Unlike the data domain $\mathbb{D}$ discussed in previous sections, where only the equality and inequality relation between data values are available, the data domain $\mathbb{D}$ in this section has a richer structure where more complex predicates, e.g. the predicate defining the set of even natural numbers, can be used. Over an infinite data domain, where complex predicates can be used, symbolic automata and transducers are natural extensions of finite automata and transducers over finite alphabets, by replacing the letters from a finite alphabet with the predicates over the infinite data domain. The concept of symbolic finite-state automata/transducers was initially introduced by Watson in [Wat96], then investigated by van Noord and Gerdemann in [vNG01], with motivation from natural language processing. The recent development of this topic by Veanes, Bjørner, et al., was mainly driven by regular expression analysis and advanced web security analysis [VB11a, VHL+12, Vea13].

In the following, we first present symbolic automata and then symbolic transducers. In the literature on symbolic automata and transducers, a data word is normally defined as an element of $\mathbb{D}^*$, instead of an element of $(\Sigma \times \mathbb{D})^*$ as in the previous sections. In this section, we follow this convention and define data words as elements of $\mathbb{D}^*$.

### 7.1 Symbolic Automata

The data domain $\mathbb{D}$ equipped with predicates used in symbolic automata is formalised by effective Boolean algebra, which is defined as follows.

**Definition 18 (Effective Boolean algebra).** *An* effective Boolean algebra $\Upsilon$ *is a tuple* $(\Omega, \|\circ\|, \Psi)$ *satisfying the following constraints:*

- $\Omega = (\mathfrak{S}, \mathfrak{F}, \mathfrak{P})$ *is a signature such that* $\mathfrak{S}$ *is a singleton set* $\{s\}$*,* $\mathfrak{F}$*, and* $\mathfrak{P}$ *are recursively enumerable sets.*
- $\|\circ\|$ *is an* $\Omega$*-interpretation such that* $\|s\|$ *is a recursively enumerable set, called the universe (denoted by* $\mathbb{D}$*).*

– $\Psi = \bigcup_{i \in \mathbb{N} \setminus \{0\}} \Psi^{(i)}$ *such that for each* $i \in \mathbb{N} \setminus \{0\}$, $\Psi^{(i)}$ *is a recursively enumerable set of* $i$*-ary* $\Omega$*-formulae closed under Boolean connectives* $\vee, \wedge, \neg$. *For each* $\psi(\boldsymbol{x}) \in \Psi$, *we use* $\|\psi\|$ *to denote the set* $\{\eta(\boldsymbol{x}) \mid \eta$ *is an* $\Omega$*-assignment, and* $\|\circ\| \models_\eta \psi(\boldsymbol{x})\}$. *Elements of* $\|\psi\|$ *are called the* witnesses *of* $\psi$.

Let $\Upsilon = (\Omega, \|\circ\|, \Psi)$ be an effective Boolean algebra and $\psi \in \Psi$. Then $\psi$ is *satisfiable*, denoted by $\mathsf{isSat}(\psi)$, if $\|\psi\| \neq \varnothing$. In addition, $\Upsilon$ is *decidable* iff it is decidable to check $\mathsf{isSat}(\psi)$ for $\psi \in \Psi$.

**Definition 19 (Symbolic finite-state automata).** *A* symbolic finite-state automaton *(SFA) is a tuple* $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$, *where:*

– $Q$ *is a finite set of states,*
– $\Upsilon = (\Omega, \|\circ\|, \Psi)$ *is a decidable effective Boolean algebra,*
– $q_0 \in Q$ *is the initial state,*
– $F \subseteq Q$ *is the set of final states,*
– $\delta \subseteq Q \times \Psi^{(1)} \times Q$ *is a finite set of symbolic transitions.*

*An SFA* $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$ *is* deterministic *if for every* $(q_1, \psi, q_2), (q_1, \psi', q_2') \in \delta$, *if* $\mathsf{isSat}(\psi \wedge \psi')$ *holds, then* $q_2 = q_2'$.

*Semantics of SFAs.* Let $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$ be an SFA. A *symbolic* transition $t = (q_1, \psi, q_2) \in \delta$ in the SFA $\mathcal{A}$ can be concretised into a set $\|t\|$ of *concrete (standard)* transitions $\rightarrow \subseteq Q \times \mathbb{D} \times Q$ defined as follows: For every $d \in \mathbb{D}$, $q_1 \xrightarrow{d} q_2 \in \|t\|$ iff $d \in \|\psi\|$. Intuitively, suppose that $\mathcal{A}$ is in the state $q_1$ and reading a data value $d$, if there is a transition $(q_1, \psi, q_2) \in \delta$ such that $d \in \|\psi\|$, then $\mathcal{A}$ moves from $q_1$ to $q_2$ after consuming the input data value $d$.

Given a data word $w = d_1 ... d_n \in \mathbb{D}^*$, $q_1 \overset{w}{\twoheadrightarrow} q_{n+1}$ if there exist states $q_2, ..., q_n \in Q$ such that for all $i \in [n]$, $q_i \xrightarrow{d_i} q_{i+1} \in \|t\|$ for some transition $t \in \delta$. A data word $w$ is *accepted* at the state $q$ of $\mathcal{A}$ iff there exists a state $q_f \in F$ such that $q \overset{w}{\twoheadrightarrow} q_f$. Let $\mathcal{L}_q(\mathcal{A})$ denote the set of data words accepted at the state $q$ of $\mathcal{A}$. Then the data language defined by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is $\mathcal{L}_{q_0}(\mathcal{A})$.

*Example 11.* Let us consider the language $L_{2^{31}}$ over integers, in which either the second letter is less than $-2^{31}$ and the last letter is greater than $2^{31}$, or the second letter is greater than $2^{31}$ and the last letter is less than $-2^{31}$. $L_{2^{31}}$ cannot be defined by any finite state automaton. Let $\mathcal{A}_{2^{31}} = (\{q_0, q_1, q_2, q_3, q_4\}, \Upsilon, q_0, \delta, \{q_4\})$ be the SFA such that $\Upsilon$ is linear arithmetic over integers and $\delta$ is illustrated in Fig. 5 (where $2^{31}$ is an abbreviation the sequence of 32 bits $10^{31}$). $\mathcal{A}_{2^{31}}$ defines the data language $L_{2^{31}}$.

An $\epsilon$-SFA $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Upsilon \cup \{\epsilon\}, q_0, \delta, F)$, where $Q$, $\Upsilon$, $q_0$ and $F$ are defined as for SFAs, and $\delta \subseteq Q \times (\Psi^{(1)} \cup \{\epsilon\}) \times Q$. An $\epsilon$-transition $(q_1, \epsilon, q_2)$ in an $\epsilon$-SFA $\mathcal{A}$ allows it to move from the state $q_1$ to the state $q_2$ without consuming any input data value. The semantics of $\epsilon$-SFA can be defined as a natural extension of that of SFAs.

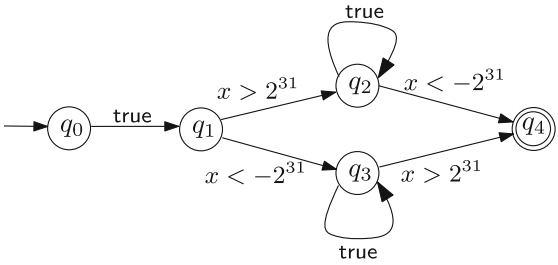



**Fig. 5.** The SFT $\mathcal{A}_{2^{31}}$

Let $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$ be an SFA. A state $q \in Q$ is called *partial* if there is $d \in \mathbb{D}$ such that there are no $q' \in Q$ satisfying that $q \xrightarrow{d} q'$. Note that given a state $q \in Q$, we can decide whether $q$ is partial by checking whether $\bigvee_{(q,\psi,q')\in\delta} \psi$ is valid, that is, whether $\bigwedge_{(q,\psi,q')\in\delta} \neg\psi$ is unsatisfiable. Then $\mathcal{A}$ is *minimal* if the following conditions hold:

- $\mathcal{A}$ is deterministic,
- $\mathcal{A}$ is complete, that is, $\mathcal{A}$ contains no partial states,
- $\mathcal{A}$ is clean, that is, for every $(q_1, \psi, q_2) \in \delta$, it holds that $\mathsf{isSat}(\psi)$ and there is $w \in \mathbb{D}^*$ such that $q_0 \xrightarrow{w} q_1$,
- $\mathcal{A}$ is normalized, that is, for each pair of states $q_1, q_2 \in Q$, there is at most one transition between them (otherwise, two transitions $(q_1, \psi_1, q_2), (q_1, \psi_2, q_2) \in \delta$ can be combined into one transition $(q_1, \psi_1 \vee \psi_2, q_2)$),
- for all $q_1, q_2 \in Q$, $q_1 = q_2$ iff $\mathcal{L}_{q_1}(\mathcal{A}) = \mathcal{L}_{q_2}(\mathcal{A})$.

Let $\mathcal{L} \subseteq \mathbb{D}^*$ be a data language. Then the *Kleene-closure* of $\mathcal{L}$, denoted by $\mathcal{L}^*$, is defined as $\{\varepsilon\} \cup \{w_1 \dots w_n \mid n \geq 1, w_i \in \mathcal{L}\}$. The *reversal* of $\mathcal{L}$, denoted by $\mathcal{L}^{\mathsf{rev}}$, is defined as $\{d_1 \dots d_n \mid d_n \dots d_1 \in \mathcal{L}\}$.

It turns out SFAs preserve all the nice properties of finite-state automata.

**Theorem 18** ([vNG01, VHL+12, DV14])**.** *The following results hold for SFAs:*

- *Each $\epsilon$-SFA can be transformed into an equivalent SFA in linear time.*
- *SFAs are closed under determinization, all the Boolean operations, concatenation, Kleene-closure and reversal.*
- *The nonemptiness, the universality and the equivalence problems of SFAs are decidable.*

SFAs can only enforce constraints on the data value of a single position, and are incapable of comparing data values in different positions, which is the main reason why SFAs preserve all the nice properties of finite-state automata. In the following, we introduce an extension of SFAs that are capable of comparing data values in different positions, called extended symbolic finite-state automata (ESFA). ESFAs extend SFAs with lookahead, that is, by allowing to read several consecutive input data values in a single transition.

**Definition 20 (Extended symbolic finite-state automata).** *An* extended symbolic finite-state automaton *(ESFA) over the sort $s$ is a tuple $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$, where:*

- *$Q$ is a finite set of states including a specific state $q_f$,*
- *$\Upsilon = (\Omega, \| \circ \|, \Psi)$ is a decidable effective Boolean algebra such that $\Omega = (\mathfrak{S}, \mathfrak{F}, \mathfrak{P})$ and $\mathfrak{S} = \{s\}$,*
- *$q_0 \in Q$ is the initial state,*
- *$\delta$ is a finite set of transition rules of the form $t = (q_1, \ell, \psi, q_2)$, where*
  - *$q_1 \in Q \setminus \{q_f\}$ and $q_2 \in Q$ are respectively the source and target states of $t$,*
  - *$\ell \in \mathbb{N} \setminus \{0\}$ is the* lookahead *of $t$,*
  - *$\psi \in \Psi^{(\ell)}$, that is, $\psi$ is an $\ell$-ary formula in $\Psi$,*
- *$F$ is a set of final rules of the form $t = (q_1, \ell, \psi, q_f)$ such that if $\ell > 0$, then $t$ satisfies the same constraints as for transition rules, otherwise (i.e. $\ell = 0$), then $\psi = \mathsf{true}$. Intuitively, a final rule $(q_1, \ell, \psi, q_f)$ is used when the rest of the input data word is of length $\ell$, where $\ell = 0$ corresponds to the situation that $\mathcal{A}$ already reaches the right end of the data word. It is a generalisation of final states in finite-state automata.*

*The lookahead of an ESFA $\mathcal{A}$ is the maximum of the lookaheads of the (transition or final) rules in $\mathcal{A}$.*

*Semantics of ESFAs.* Let $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$ be an ESFA. The semantics of the rules $t = (q_1, \ell, \psi, q_2) \in \delta$ of $\mathcal{A}$ is defined as follows: If $\ell = 0$, then $\psi = \mathsf{true}$ and $q_2 = q_f$, therefore, $\|t\| = \{q_1 \xrightarrow{\varepsilon} q_f\}$. Otherwise,

$$\|t\| = \{q_1 \xrightarrow{w} q_2 \mid w = d_1 \dots d_\ell \in (\mathbb{D}_s)^\ell, (d_1, \dots, d_\ell) \in \|\psi\|\}.$$

Intuitively, using the transition $t = (q_1, \ell, \psi, q_2)$, $\mathcal{A}$ reads the next $\ell$ input data values $w$ (including the one in the current position), if the corresponding tuple of data values satisfies $\psi$, then $\mathcal{A}$ consumes the word $w$ and moves from the state $q_1$ to the state $q_2$.

Given a data word $w \in \mathbb{D}^*$, $q_1 \overset{w}{\twoheadrightarrow} q_{n+1}$ if there exist states $q_2, ..., q_n \in Q$ and data words $w_1, ..., w_n \in \mathbb{D}^*$ such that $w = w_1 ... w_n$ and for all $i \in [n]$, $q_i \xrightarrow{w_i} q_{i+1}$. A data word $w \in \mathbb{D}^*$ is *accepted* by $\mathcal{A}$ iff $q_0 \overset{w}{\twoheadrightarrow} q_f$. The data language defined by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of data words accepted by $\mathcal{A}$.

An ESFA $\mathcal{A} = (Q, \Upsilon, q_0, \delta, F)$ is *deterministic* if for every pair of rules $(q_1, \ell, \psi, q_2)$ and $(q_1, \ell', \psi', q_2')$ in $\mathcal{A}$,

- if $q_2,\ q_2' \in Q \setminus \{q_f\}$ and $\mathsf{isSat}(\psi \wedge \psi')$, then $q_2 = q_2'$ and $\ell = \ell'$;
- if $q_2 = q_2' = q_f$ and $\mathsf{isSat}(\psi \wedge \psi')$, then $\ell = \ell'$;
- if $q_2 \in Q \setminus \{q_f\}$, $q_2' = q_f$ and $\mathsf{isSat}(\psi \wedge \psi')$, then $\ell > \ell'$.

*Example 12.* Let us consider the ESFA $\mathcal{A}_{script} = (\{q_0, q_1, q_f\}, \Upsilon, q_0, \delta, F)$ such that

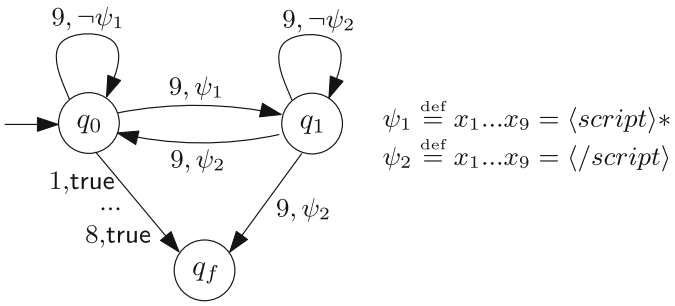



**Fig. 6.** The ESFA $\mathcal{A}_{script}$

- $\Upsilon$ is the theory of UTF-8 characters where all the function symbols are constants and the set of predicate symbols is empty,
- $\delta$ and $F$ are illustrated in Fig. 6, where $\psi_1$ is an abbreviation of the formula $x_1 = \langle \wedge\ x_2 = s \wedge x_3 = c \wedge \cdots \wedge\ x_8 = \rangle$ and $\psi_2$ is an abbreviation of the formula $x_1 = \langle \wedge\ x_2 = / \wedge x_3 = s \wedge \cdots \wedge x_9 = \rangle$.

Then the ESFA $\mathcal{A}_{script}$ defines the set of words $w$ such that each occurrence of $\langle script \rangle$ is followed by an occurrence of $\langle /script \rangle$ in the future. Note that although $\mathcal{A}_{script}$ is over a finite alphabet, it is much more succinct than the corresponding finite-state automaton defining the same language, where an enumeration of all the possible subwords of length 9 satisfying $\neg\psi_1$ or $\neg\psi_2$ is necessary.

A formula $\psi \in \Psi^{(\ell)}$ (where $\ell > 0$) is *Cartesian* if $\|\psi\|$ is equivalent to $D_1 \times \cdots \times D_\ell$ for some $D_1, \ldots, D_\ell \subseteq \mathbb{D}_s$. An ESFA $\mathcal{A}$ is *Cartesian* if for each rule $(q_1, \ell, \psi, q_2)$ in $\mathcal{A}$ such that $\ell > 0$, it holds that $\psi$ is Cartesian. For a satisfiable formula $\psi(\boldsymbol{x}) \in \Psi^{(\ell)}$ (where $\ell > 0$), to decide whether $\psi$ is Cartesian is equivalent to check whether for some witness $(d_1, ..., d_\ell)$ of $\psi$,

$$\forall x_1, ...x_\ell(\psi(x_1, ..., x_\ell)) \iff \bigwedge_{1 \le i \le \ell} \psi(d_1, ..., d_{i-1}, x_i, d_{i+1}, ..., d_\ell).$$

A formula $\psi(\boldsymbol{x}) \in \Psi^{(\ell)}$ (where $\ell > 0$) is *monadic* if it is equivalent to a Boolean combination of unary formulae. For instance, $\psi(x_1, x_2) \stackrel{\text{def}}{=} x_1 = x_2 \bmod 2$ is a monadic formula since it is equivalent to $(x_1 = 0 \bmod 2 \wedge x_2 = 0 \bmod 2) \vee (x_1 = 1 \bmod 2 \wedge x_2 = 1 \bmod 2)$, while $\psi(x_1, x_2) \stackrel{\text{def}}{=} x_1 < x_2$ is not. In [VBNB14], a semi-decision procedure was provided to compute an equivalent Boolean combination of unary formulae, from a given quantifier-free formula over a decidable background theory.

An ESFA $\mathcal{A}$ is *monadic* if for each rule $(q_1, \ell, \psi, q_2)$ of $\mathcal{A}$ such that $\ell > 0$, $\psi$ is monadic.

**Theorem 19** ([DV15])**.** *The following results hold for ESFAs:*

- *Cartesian ESFAs, monadic ESFAs and SFAs are expressively equivalent, moreover, this also holds for the deterministic case.*

- *The membership and nonemptiness problems of ESFAs are decidable, but the universality, language inclusion and equivalence problems of ESFAs are undecidable.*
- *For each $\ell \in \mathbb{N} \setminus \{0\}$, ESFAs with lookahead $\ell + 1$ are more expressive than ESFAs with lookahead $\ell$.*
- *ESFAs are closed under union, but not closed under intersection or complementation. Moreover, checking whether there exists an input word accepted by two ESFAs $\mathcal{A}$ and $\mathcal{A}'$ with lookahead 2 over quantifier free successor arithmetic and tuples is undecidable.*

In [DV15], the last result in Theorem 19 was shown by reducing the reachability problem of Minsky machines to the problem checking whether there exists an input word accepted by two ESFAs $\mathcal{A}$ and $\mathcal{A}'$ with lookahead 2 over quantifier free successor arithmetic and tuples from $\mathbb{N}^3$.

*Remark 2.* In the definition of ESFAs, when the reading head is in the position $i$ and a transition with $\ell \geq 2$ lookahead is used, then after the transition, the reading head will be moved to the position $i + \ell$, instead of the next position to the right of $i$, that is, $i + 1$. This special semantics of lookaheads in ESFAs is essential for the decidability of nonemptiness problem. (Otherwise, we are already be able to reduce the reachability problem of Minsky machines to the nonemptiness problem of ESFAs.)

### 7.2 Symbolic Transducers

Similar to symbolic automata, symbolic transducers are introduced as extensions of finite-state transducers, where the input letters are replaced by formulae over an infinite data domain and the output letters are replaced by terms.

For the definition of symbolic transducers, we introduce the concept of background theories and label theories. Intuitively, background theories are many-sorted Boolean algebra satisfying the additional constraint that the set of formulae is closed under substitutions. Label theories extend background theories further by adding inequalities of terms into the set of formulae.

**Definition 21 (Background theories).** *A background theory $\Upsilon$ is a tuple $(\Omega, \|\circ\|, \Psi)$ satisfying the following constraints:*

- *$\Omega = (\mathfrak{S}, \mathfrak{F}, \mathfrak{P})$ is a signature satisfying that each of $\mathfrak{S}, \mathfrak{F}, \mathfrak{P}$ is a recursively enumerable set.*
- *$\|\circ\|$ is an $\Omega$-interpretation such that for each $s \in \mathfrak{S}$, $\|s\|$ is a recursively enumerable set (denoted by $\mathbb{D}_s$).*
- *$\Psi = \bigcup_{\boldsymbol{s} \in \mathfrak{S}^+} \Psi^{(\boldsymbol{s})}$ such that for each $\boldsymbol{s} = (s_1, \ldots, s_i) \in \mathfrak{S}^+$, $\Psi^{(\boldsymbol{s})}$ is a recursively enumerable set of $\Omega$-formulae of arity $s_1 \times \cdots \times s_i$ closed under Boolean connectives $\vee, \wedge, \neg$. In addition, $\Psi$ is closed under substitutions, that is, for each $\boldsymbol{s}/\boldsymbol{s}'$-term $\boldsymbol{f}$ and $\psi(\boldsymbol{x}) \in \Psi^{(\boldsymbol{s}')}$, we have $\psi[\boldsymbol{f}/\boldsymbol{x}] \in \Psi^{(\boldsymbol{s})}$. For each $\psi(\boldsymbol{x}) \in \Psi$, we use $\|\psi\|$ to denote the set $\{\eta(\boldsymbol{x}) \mid \eta$ is an $\Omega$-assignment, and $\|\circ\| \models_\eta \psi(\boldsymbol{x})\}$. Elements of $\|\psi\|$ are called the* witnesses *of $\psi$.*

The notion $\mathsf{isSat}(\psi)$ for $\psi \in \Psi$ and the decidability of $\Upsilon$ can be defined similarly as effective Boolean algebra.

**Definition 22 (Label theories).** *A label theory $\Upsilon$ with the input sort $s_{\mathsf{in}}$ and output sort $s_{\mathsf{out}}$ is a tuple $(\Omega, \|\circ\|, \Psi, \Psi')$ satisfying the following constraints:*

- *$(\Omega, \|\circ\|, \Psi)$ is a background theory such that $\Omega = (\mathfrak{S}, \mathfrak{F}, \mathfrak{P})$ and $s_{\mathsf{in}}, s_{\mathsf{out}} \in \mathfrak{S}$,*
- *$\Psi' = \bigcup_{i \in \mathbb{N}\setminus\{0\}} (\Psi')^{(s_{\mathsf{in}}^i)}$ such that for each $i \in \mathbb{N} \setminus \{0\}$, $(\Psi')^{(s_{\mathsf{in}}^i)}$ comprises the formulae of the form $\psi(\boldsymbol{x}) \wedge f(\boldsymbol{x}) \neq g(\boldsymbol{x})$, where $\psi(\boldsymbol{x}) \in \Psi^{(s_{\mathsf{in}}^i)}$ and $f, g$ are $s_{\mathsf{in}}^i/s_{\mathsf{out}}$-terms.*

*A label theory is decidable if it is decidable to check $\mathsf{isSat}(\psi)$ for $\psi \in \Psi \cup \Psi'$.*

Given a formula $\psi(\boldsymbol{x}) \in (\Psi)^{(s_{\mathsf{in}}^i)}$ and two $s_{\mathsf{in}}^i/s_{\mathsf{out}}$-terms $f(\boldsymbol{x}), g(\boldsymbol{x})$, $f$ and $g$ are *equivalent up to* $\psi$, denoted by $f \simeq_\psi g$, if $\mathsf{isSat}(\psi(\boldsymbol{x}) \wedge f(\boldsymbol{x}) \neq g(\boldsymbol{x}))$ does not hold. Two sequences of $s_{\mathsf{in}}^i/s_{\mathsf{out}}$-terms $\boldsymbol{f} = f_1 ... f_n$ and $\boldsymbol{g} = g_1 ... g_m$ are *equivalent up to* $\psi$, denoted by $\boldsymbol{f} \simeq_\psi \boldsymbol{g}$, iff $n = m$ and for every $j \in [n]$, $f_j \simeq_\psi g_j$.

Given a $s_{\mathsf{in}}^i/s_{\mathsf{out}}$-term $\boldsymbol{f} = f_1 ... f_n$ and a sequence of data values $\boldsymbol{d} = (d_1, \ldots, d_i) \in (\mathbb{D}_{s_{\mathsf{in}}})^i$, let $\|\boldsymbol{f}\|(\boldsymbol{d})$ denote the sequence $\|f_1\|(\boldsymbol{d}) ... \|f_n\|(\boldsymbol{d})$, that is, a data word of sort $s_{\mathsf{out}}$.

**Definition 23 (Symbolic finite-state transducers).** *A* symbolic finite-state transducer *(SFT) is a tuple $\mathcal{A} = (Q, \Upsilon, s_{\mathsf{in}}, s_{\mathsf{out}}, q_0, \delta, F)$, where:*

- *$Q$, $q_0$ and $F$ are defined as those for SFAs,*
- *$\Upsilon = (\mathfrak{S}, \mathfrak{F}, \mathfrak{P})$ is a decidable label theory with the input sort $s_{\mathsf{in}}$ and output sort $s_{\mathsf{out}}$,*
- *$\delta$ is a finite set of symbolic transitions $(q, \psi, \boldsymbol{f}, q')$ such that $q, q' \in Q$, $\psi \in \Psi^{s_{\mathsf{in}}}$ and $\boldsymbol{f}$ is a sequence of $s_{\mathsf{in}}/s_{\mathsf{out}}$-terms.*

Let $\mathcal{A} = (Q, \Upsilon, s_{\mathsf{in}}, s_{\mathsf{out}}, q_0, \delta, F)$ be an SFT. Then $\mathcal{A}$ is *deterministic* if for all $(q_1, \psi, \boldsymbol{f}, q_2), (q_1, \psi', \boldsymbol{f'}, q_2') \in \delta$, if $\mathsf{isSat}(\psi \wedge \psi')$, then $q_2 = q_2'$ and $\boldsymbol{f} \simeq_{\psi \wedge \psi'} \boldsymbol{f'}$.

*Semantics of SFTs.* Similar to SFAs, a symbolic transition $t = (q_1, \psi, \boldsymbol{f}, q_2) \in \delta$ in the SFT $\mathcal{A}$ can be concretised into a potentially infinite set $\|t\|$ of *concrete* transitions $\rightarrow \subseteq Q \times \mathbb{D}_{s_{\mathsf{in}}} \times (\mathbb{D}_{s_{\mathsf{out}}})^* \times Q$, where $q_1 \xrightarrow{d/w} q_2 \in \|t\|$ iff $d \in \|\psi\|$ and $w = \|\boldsymbol{f}\|(d)$. Intuitively, suppose $\mathcal{A}$ is at the state $q_1$ and reading the input data value $d \in \mathbb{D}_{s_{\mathsf{in}}}$, if there is a transition $(q_1, \psi, \boldsymbol{f}, q_2) \in \delta$ such that $d \in \|\psi\|$, then $\mathcal{A}$ can move from the state $q_1$ to the state $q_2$ after reading $d$, moreover it produces a data word $w \in (\mathbb{D}_{s_{\mathsf{out}}})^*$.

Given a data word $u = d_1 ... d_n \in (\mathbb{D}_{s_{\mathsf{in}}})^*$, $q_1 \overset{u/w}{\twoheadrightarrow} q_{n+1}$ if there exist states $q_2, ..., q_n \in Q$ and data words $w_1, ..., w_n \in (\mathbb{D}_{s_{\mathsf{out}}})^*$ such that $w = w_1 ... w_n$ and for each $i \in [n]$, $q_i \xrightarrow{d_i/w_i} q_{i+1}$. The *transduction* $\mathcal{T}_\mathcal{A}$ defined by the SFT $\mathcal{A}$ is a relation $\mathcal{T}_\mathcal{A} \subseteq (\mathbb{D}_{s_{\mathsf{in}}})^* \times (\mathbb{D}_{s_{\mathsf{out}}})^*$ defined as follows: For each $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$ and $w \in (\mathbb{D}_{s_{\mathsf{out}}})^*$, $(u, w) \in \mathcal{T}_\mathcal{A}$ iff there exists $q' \in F$ such that $q_0 \overset{u/w}{\twoheadrightarrow} q'$. For each

$u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$, define $\mathcal{T}_{\mathcal{A}}(u) = \{w \in (\mathbb{D}_{s_{\mathsf{out}}})^* \mid (u, w) \in \mathcal{T}_{\mathcal{A}}\}$. The SFT $\mathcal{A}$ is *single-valued* if for all $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$, $|\mathcal{T}_{\mathcal{A}}(u)| \leq 1$. The SFT $\mathcal{A}$ is *finite-valued* if there exists a bound $K \geq 0$ such that for all $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$, $|\mathcal{T}_{\mathcal{A}}(u)| \leq K$.

*Example 13.* Let us consider the simple SFT

$$\mathcal{A}_{\mathbf{xor}} = (\{q_0\}, \Upsilon, BV^2, BV, q_0, \{(q_0, \mathsf{true}, f, q_0)\}, \{q_0\}),$$

where $BV^2$ and $BV$ are respectively bit vectors with length 2 and 1, the function $f \stackrel{\text{def}}{=} \lambda b_0, b_1.\ b_0\ \mathbf{xor}\ b_1$ (**xor** is the bitwise exclusive or operator). The SFT $\mathcal{A}_{\mathbf{xor}}$ transforms each sequence of bit pairs $(b_0^1, b_1^1)...(b_0^n, b_1^n)$ into a sequence of bits $b^1...b^n$ such that for all $i \in [n]$, $b^i = b_0^i\ \mathbf{xor}\ b_1^i$.

Let $\Upsilon_1 = (\Omega_1, \| \circ \|_1, \Psi_1, \Psi_1')$ be a label theory with input sort $s_1$ and output sort $s_2$, and $\Upsilon_2 = (\Omega_2, \| \circ \|_2, \Psi_2, \Psi_2')$ be a label theory with input sort $s_2$ and output sort $s_3$. Then $\Upsilon_1$ and $\Upsilon_2$ are said to be *composable* if the following constraints hold: Let $\Omega_1 = (\mathfrak{S}_1, \mathfrak{F}_1, \mathfrak{P}_1)$ and $\Omega_2 = (\mathfrak{S}_2, \mathfrak{F}_2, \mathfrak{P}_2)$, then

- $\mathfrak{S}_1 \cap \mathfrak{S}_2 = \{s_2\}$,
- for each $i, j \in \mathbb{N} \setminus \{0\}$, the set of functions from $\mathfrak{F}_1$ of arity $s_2^i \to s_2^j$ is the same as the set of functions from $\mathfrak{F}_2$ of arity $s_2^i \to s_2^j$, moreover, for each such function $f$, $\|f\|_1 = \|f\|_2$, finally, all these function symbols are the only ones shared by $\mathfrak{F}_1$ and $\mathfrak{F}_2$,
- for each $i \in \mathbb{N} \setminus \{0\}$, the set of predicates from $\mathfrak{P}_1$ of arity $s_2^i$ is the same as the set of predicates from $\mathfrak{P}_2$ of arity $s_2^i$, moreover, for each such predicate $p$, $\|p\|_1 = \|p\|_2$, finally, all these predicate symbols are the only ones shared by $\mathfrak{P}_1$ and $\mathfrak{P}_2$.

From two composable label theories $\Upsilon_1$ and $\Upsilon_2$, a label theory $\Upsilon = (\Omega, \|\circ\|, \Psi, \Psi')$, called the *composition* of $\Upsilon_1$ and $\Upsilon_2$, can be defined as follows.

- the input sort and output sort of $\Upsilon$ are $s_1$ and $s_3$ respectively,
- $\Omega = (\mathfrak{S}_1 \cup \mathfrak{S}_2, \mathfrak{F}_1 \cup \mathfrak{F}_2, \mathfrak{P}_1 \cup \mathfrak{P}_2)$.
- The $\| \circ \|$-interpretations of sorts, function symbols, and predicate symbols from $\Omega_1 \cap \Omega_2$ are those of $\| \circ \|_1$. On the other hand, the $\| \circ \|$-interpretations of sorts, function symbols, and predicate symbols from $(\Omega_1 \setminus \Omega_2) \cup (\Omega_2 \setminus \Omega_1)$ inherit from $\| \circ \|_1$ or $\| \circ \|_2$.
- $\Psi$ is closure of $\Psi_1 \cup \Psi_2$ under Boolean connectives and substitutions (i.e. the minimum set of formulae that subsumes $\Psi_1 \cup \Psi_2$ and is closed under Boolean connectives and substitutions).
- $\Psi' = \bigcup_{i \in \mathbb{N} \setminus \{0\}} (\Psi')^{(s_1^i)}$ such that for each $i \in \mathbb{N} \setminus \{0\}$, $(\Psi')^{(s_1^i)}$ comprises the formulae of the form $\psi(\boldsymbol{x}) \wedge f(\boldsymbol{x}) \neq g(\boldsymbol{x})$, where $\psi(\boldsymbol{x}) \in \Psi^{(s_1^i)}$ and $f, g$ are $s_1^i/s_3$-terms.

SFTs are said to be *closed under composition* if for each pair of SFTs $\mathcal{A}_1$ with the input/output sort $s_1/s_2$, and $\mathcal{A}_2$ with the input/output sort $s_2/s_3$, there is an SFT $\mathcal{A}$ such that for each data word $w \in (\mathbb{D}_{s_1})^*$, it holds that

$\mathcal{T}_{\mathcal{A}}(w) = \mathcal{T}_{\mathcal{A}_2}(\mathcal{T}_{\mathcal{A}_1}(w))$. Two SFTs $\mathcal{A}_1$ and $\mathcal{A}_2$ with the input/output sort $s_1/s_2$ are *equivalent* if for each $w \in (\mathbb{D}_{s_1})^*$, $\mathcal{T}_{\mathcal{A}_1}(w) = \mathcal{T}_{\mathcal{A}_2}(w)$. The equivalence problem of SFTs is to decide the equivalence of two given SFTs with the same input/output sorts.

**Theorem 20** ([vNG01, VHL+12, VB16]). *The following results hold for SFTs:*

- *SFTs are closed under composition if their label theories are composable.*
- *The equivalence problem of finite-valued SFTs is decidable.*

We would like to mention that the equivalence problem of finite state transducers (hence for SFTs) is undecidable [FV98].

Similarly to the extension of SFAs into ESFAs, SFTs can be naturally generalised into extended symbolic finite-state transducers (ESFTs).

**Definition 24 (Extended symbolic finite-state transducers).** *An* extended symbolic finite-state transducer *(ESFT) $\mathcal{A}$ is a tuple $(Q, \Upsilon, s_{\mathsf{in}}, s_{\mathsf{out}}, q_0, \delta, F)$, where $Q$, $\Upsilon, s_{\mathsf{in}}, s_{\mathsf{out}}$ and $q_0 \in Q$ are defined as those for SFTs, and $\delta$ is a finite set of transition rules of the form $t = (q_1, \ell, \psi, \boldsymbol{f}, q_2)$, where:*

- *$q_1 \in Q \setminus \{q_f\}$ and $q_2 \in Q$ are respectively the source and target states of $t$,*
- *$\ell \in \mathbb{N} \setminus \{0\}$ is the* lookahead *of $t$,*
- *$\psi \in \Psi^{(s_{\mathsf{in}}^{\ell})}$,*
- *$\boldsymbol{f}$ is a sequence of $s_{\mathsf{in}}^{\ell}/s_{\mathsf{out}}$-terms, each of them representing a function from $(\mathbb{D}_{s_{\mathsf{in}}})^i$ to $\mathbb{D}_{s_{\mathsf{out}}}$,*

*and $F$ is a set of* final rules *$t = (q_1, \ell, \psi, \boldsymbol{f}, q_f)$ such that if $\ell > 0$, then $t$ satisfies the same constraints as transition rules, otherwise (i.e. $\ell = 0$), $\psi = \mathsf{true}$.*

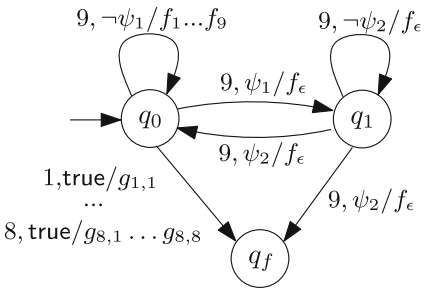
The lookahead of an ESFT is defined similarly as for ESFAs.

*Semantics of ESFTs.* Let $\mathcal{A} = (Q, \Upsilon, s_{\mathsf{in}}, s_{\mathsf{out}}, q_0, \delta, F)$ be an ESFT. The semantics of rules $t = (q_1, \ell, \psi, \boldsymbol{f}, q_2)$ of $\mathcal{A}$ is defined as follows:

$$\|t\| = \{q_1 \xrightarrow{u/w} q_2 \mid u \in \|\psi\|, w \in \|\boldsymbol{f}\|(u)\}.$$

Intuitively, the transition $t = (q_1, \ell, \psi, \boldsymbol{f}, q_2)$ reads $\ell$ adjacent input data values $u$ that satisfies $\psi$, then produces a sequence of data values $w \in \|\boldsymbol{f}\|(u)$.

Given a data word $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$, $q_1 \overset{u/w}{\twoheadrightarrow} q_{n+1}$ if there exist states $q_2, ..., q_n \in Q$, words $u_1, ..., u_n \in (\mathbb{D}_{s_{\mathsf{in}}})^*$ and words $w_1, ..., w_n \in (\mathbb{D}_{s_{\mathsf{out}}})^*$ such that $u = u_1 ... u_n$, $w = w_1 ... w_n$ and for each $i \in [n]$, $q_i \xrightarrow{u_i/w_i} q_{i+1}$. The *transduction* $\mathcal{T}_{\mathcal{A}}$ defined by $\mathcal{A}$ is a relation on $(\mathbb{D}_{s_{\mathsf{in}}})^* \times (\mathbb{D}_{s_{\mathsf{out}}})^*$ defined as follows: For each $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$ and $w \in (\mathbb{D}_{s_{\mathsf{out}}})^*$, $(u, w) \in \mathcal{T}_{\mathcal{A}}$ iff $q_0 \overset{u/w}{\twoheadrightarrow} q_f$. In addition, we use $\mathcal{T}_{\mathcal{A}}(u)$ to denote the set $\{w \in (\mathbb{D}_{s_{\mathsf{out}}})^* \mid (u, w) \in \mathcal{T}_{\mathcal{A}}\}$.




$$\psi_1 \stackrel{\text{def}}{=} x_1 \ldots x_9 = \langle script \rangle *$$
$$\psi_2 \stackrel{\text{def}}{=} x_1 \ldots x_9 = \langle /script \rangle$$
$$f_\epsilon \stackrel{\text{def}}{=} \lambda x_1, \ldots, x_9.\ \epsilon$$
$$\forall i \in [9]:\ f_i \stackrel{\text{def}}{=} \lambda x_1, \ldots, x_9.\ x_i$$
$$\forall i \in [8], j \in [i]:\ g_{i,j} \stackrel{\text{def}}{=} \lambda x_1, \ldots, x_i.\ x_j$$

**Fig. 7.** The ESFT $\mathcal{A}_{script}$

*Example 14.* Let us consider the ESFT $\mathcal{A}_{script} = (\{q_0, q_1, q_f\}, \Upsilon, s, s, q_0, \delta, \{q_0, q_1\})$, where $\Upsilon$ is the theory of UTF-8 characters where all the function symbols are constants and the set of predicate symbols is empty, $s$ is the sort of UTF-8 characters, $\delta$ is shown in Fig. 7. $\mathcal{A}_{script}$ removes all the non-empty data subwords following each occurrence of $\langle script \rangle$ until $\langle /script \rangle$ occurs.

An ESFT $\mathcal{A} = (Q, \Upsilon, s_{\mathsf{in}}, s_{\mathsf{out}}, q_0, \delta, F)$ is *deterministic* if for all rules $(q_1, \ell, \psi, \boldsymbol{f}, q_2)$, $(q_1, \ell', \psi', \boldsymbol{f'}, q_2') \in \delta \cup F$:

- if $q_2,\ q_2' \in Q \setminus \{q_f\}$ and $\mathsf{isSat}(\psi \wedge \psi')$, then $q_2 = q_2'$, $\ell = \ell'$ and $\boldsymbol{f} \simeq_{\psi \wedge \psi'} \boldsymbol{f'}$,
- if $q_2 = q_2' = q_f$, $\mathsf{isSat}(\psi \wedge \psi')$ and $\ell = \ell'$, then $\boldsymbol{f} \simeq_{\psi \wedge \psi'} \boldsymbol{f'}$,
- if $q_2 \in Q \setminus \{q_f\}, q_2' = q_f$ and $\mathsf{isSat}(\psi \wedge \psi')$, then $\ell > \ell'$.

An ESFT $\mathcal{A}$ is *single-valued* if $|\mathcal{T}_\mathcal{A}(u)| \leq 1$ for all $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$. An ESFT $\mathcal{A}$ is *finite-valued* if there exists $K \geq 0$ such that $|\mathcal{T}_\mathcal{A}(u)| \leq K$ for all $u \in (\mathbb{D}_{s_{\mathsf{in}}})^*$. Cartesian and monadic ESFTs are defined similarly as for ESFAs.

**Theorem 21 ([DV15]).** *The following results hold for ESFTs:*

- *Cartesian ESFTs, monadic ESFTs, and SFTs are expressively equivalent, moreover, this fact holds in the deterministic case.*
- *ESFTs with lookahead $\ell+1$ are more expressive than ESFTs with lookahead $\ell$.*
- *ESFTs are not closed under composition (even if the label theories are composable).*
- *The equivalence problem of single-valued ESFTs over quantifier free successor arithmetic and tuples is undecidable, but is decidable for single-valued Cartesian ESFTs.*

It is open whether the equivalence problem of finite-valued Cartesian ESFTs is decidable or not.

*Further Reading.* Symbolic visibly pushdown automata (SVPA) were investigated in [DA14]. Another extension of SFAs, called symbolic finite-state automata with registers (SRA), was also investigated in [DV15]. It turns out that adding registers into SFAs entails undecidability, even for the nonemptiness problem, since Minsky machines can be easily simulated by SRAs. In addition,

symbolic finite-state tree automata (SFTAs) were investigated in [VB11a, VB15, VD16]. It was shown that SVPAs and SFTAs preserve all the desirable properties of visibly pushdown automata and tree automata respectively. Symbolic tree transducers (STT) were also investigated. It was shown in [FV14] that symbolic tree transducers are not closed under compositions, which corrected an incorrect claim in [VB11b].

## 8 Formalisms with Data Constraints for the Verification of Programs Manipulating Dynamic Data Structures

Dynamic data structures, or heaps, are widely used in system software, e.g., operating systems and device drivers. Formal analysis and verification of programs manipulating dynamic data structures are notoriously difficult. For instance, the sizes of dynamic data structures are unbounded, their shapes may change during the execution of the program, and their nodes may contain data values from an infinite domain, or even worse, there may be pointer arithmetics applied to the pointer variables. Researchers have proposed various approaches to reason about dynamic data structures, e.g., shape analysis [SRW02], separation logic [Rey02], and forest automata [HHR+12]. Noteworthily most work focuses on the shape properties, e.g., whether the data structure is a list, or a binary tree, but disregards data and size constraints, e.g., whether the lists and trees are sorted or the trees are balanced.

### 8.1 Separation Logic with Inductive Definitions and Data Constraints

Separation logic (SL) is an extension of Hoare logic. Since its introduction, SL has become a widely used formalism for analysing and verifying heap-manipulating programs [BCO05, DOY06, CDOY11]. As an assertion language, SL can express how data structures are laid out in memory in a succinct way. In a nutshell, this language features: (i) a spatial conjunction operator that decomposes the heap into disjoint regions, each of which can be reasoned about independently, and (ii) inductive predicates that describe the shape of unbounded linked data structures such as lists, trees, etc. We shall present a version of separation logic with data constraints, which may include pure constraints on data values and capture desired properties of structural heaps such as the size, height, sortedness and even near-balanced tree properties.

As in Sect. 7, we consider a data domain $\mathbb{D}$, but this time we have an explicit logical language to specify (much) more involved properties over $\mathbb{D}$. As a general framework, we are a bit abstract here and assume a theory $(\mathbb{D}, \mathcal{L})$ where $\mathcal{L}$ is a suitable logical structure interpreted over $\mathbb{D}$. Typical cases include Presburger arithmetic (in which $(\mathbb{D}, \mathcal{L}) = (\mathbb{N}, +, \leq, 0, 1)$), logical theories supported by modern SMT solvers, or even logical theories on sets or multisets. As a convention, *data variables* are typically denoted by $\mathsf{DVars}$, ranged over by lowercase letters $x, y, \cdots$.

To define a *separation logic with data constraints*, we further assume an infinite set $\mathbb{L}$ of locations. As a convention, $l, l', \cdots \in \mathbb{L}$ denote locations. Accordingly, we introduce a set of *location variables* $\mathsf{LVars}$ ranged over by uppercase letters $E, F, X, Y, \cdots$. We further consider two kinds of *fields*, i.e., location fields from $\mathcal{F}$ and data fields from $\mathcal{D}$. Each field $f \in \mathcal{F}$ (resp. $d \in \mathcal{D}$) is associated with $\mathbb{L}$ (resp. $\mathbb{D}$). A *term* is either a variable from $\mathsf{DVars} \cup \mathsf{LVars}$, or the constant symbol $\mathsf{nil}$. We usually use $t$ and $\boldsymbol{t}$ to denote a term and a tuple of terms.

Logic formulae may contain a set of (user-defined) inductive predicates, which are collected in $\mathcal{P}$ and will defined momentarily. In the following, the logic is denoted by $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$.

*Syntax.* $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$ formulae comprise three types of formulae: *pure formulae* $\Pi$, *data formulae* $\Delta$, and *spatial formulae* $\Sigma$, which are defined by the following rules:

$$\begin{array}{rrr} \Pi \stackrel{\text{def}}{=} & E = F \mid E \neq F \mid \Pi \wedge \Pi & \text{(pure formulae)} \\ \Sigma \stackrel{\text{def}}{=} & \mathtt{emp} \mid E \mapsto \rho \mid P(\boldsymbol{t}) \mid \Sigma * \Sigma & \text{(spatial formulae)} \\ \rho \stackrel{\text{def}}{=} & (f, X) \mid (d, x) \mid \rho, \rho & \\ \Delta \stackrel{\text{def}}{=} & \text{formulae from } \mathcal{L} & \text{(data formulae)} \end{array}$$

where $P \in \mathcal{P}$, $f \in \mathcal{F}$, and $d \in \mathcal{D}$. For spatial formulae $\Sigma$, formulae of the form $\mathtt{emp}$, $E \mapsto \rho$, or $P(\boldsymbol{t})$ are called *spatial atoms*. In particular, formulae of the form $E \mapsto \rho$ and $P(\boldsymbol{t})$ are called *points-to atoms* and *predicate atoms* respectively. Each predicate $P \in \mathcal{P}$ has a fixed arity, and is of the form

$$P(\boldsymbol{t}) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} \exists \boldsymbol{w}_i.(\Pi_i \wedge \Delta_i \wedge \Sigma_i),$$

We call $\exists \boldsymbol{w}_i.(\Pi_i \wedge \Delta_i \wedge \Sigma_i)$ the *rule* of $P(\boldsymbol{t})$. In addition, if in a rule $\exists \boldsymbol{w}_i.(\Pi_i \wedge \Delta_i \wedge \Sigma_i)$, $\Sigma_i$ contains predicate atoms, the rule is called an *inductive rule*; otherwise, it is called a *base rule*.

*Remark 3.* Separation logic, as an extension of first-order logic, usually encompasses two connectives: the separating conjunction ($*$) and its adjoin (the separating implication $-\!\!*$, aka the *magic wand*). It turns out that the magic wand is so powerful that, adding it to the logic would make the logic undecidable immediately (with only very few exceptions). Moreover, although very interesting from a theoretical perspective, its importance in program verification is debatable, since in many cases, the use of the magic wand can be avoided. In light of this, we exclude this connective in our logic.

*Semantics.* Formulae of $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$ are interpreted on the *(memory) states*. Formally, a *state* is a pair $(s, h)$, where

- $s$ is a *stack*, which is a partial function from $\mathsf{LVars} \cup \mathsf{DVars}$ to $\mathbb{L} \cup \mathbb{D}$ such that $dom(s)$ is finite and $s$ respects the data type,

– $h$ is a *heap*, which is a partial function from $\mathbb{L} \times (\mathcal{F} \cup \mathcal{D})$ to $\mathbb{L} \cup \mathbb{D}$ such that
  - $h$ respects the data type of fields, that is, for each $l \in \mathbb{L}$ and $f \in \mathcal{F}$ (resp. $l \in \mathbb{L}$ and $d \in \mathcal{D}$), if $h(l, f)$ (resp. $h(l, d)$) is defined, then $h(l, f) \in \mathbb{L}$ (resp. $h(l, d) \in \mathbb{D}$); and
  - $h$ is field-consistent, i.e. every location in $h$ possess the same set of fields.

For a heap $h$, we use $\mathsf{ldom}(h)$ to denote the set of locations $l \in \mathbb{L}$ such that $h(l, f)$ or $h(l, d)$ is defined for some $f \in \mathcal{F}$ and $d \in \mathcal{D}$. Moreover, we use $\mathrm{Flds}(h)$ to denote the set of fields $f \in \mathcal{F}$ or $d \in \mathcal{D}$ such that $h(l, f)$ or $h(l, d)$ is defined for some $l \in \mathbb{L}$. Two heaps $h_1$ and $h_2$ are said to be *field-compatible* if $\mathrm{Flds}(h_1) = \mathrm{Flds}(h_2)$. We write $h_1 \# h_2$ if $\mathsf{ldom}(h_1) \cap \mathsf{ldom}(h_2) = \varnothing$. Moreover, we write $h_1 \uplus h_2$ for the disjoint union of two field-compatible fields $h_1$ and $h_2$ (this implies that $h_1 \# h_2$).

Let $(s, h)$ be a state and $\varphi$ be an $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$ formula. The semantics of $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$ formulae is defined as follows,

– $(s, h) \vDash E = F$ if $s(E) = s(F)$,
– $(s, h) \vDash E \neq F$ if $s(E) \neq s(F)$,
– $(s, h) \vDash \Pi_1 \wedge \Pi_2$ if $(s, h) \vDash \Pi_1$ and $(s, h) \vDash \Pi_2$,
– $(s, h) \vDash \mathsf{emp}$ if $\mathsf{ldom}(h) = \varnothing$,
– $(s, h) \vDash E \mapsto \rho$ if $\mathsf{ldom}(h) = s(E)$, and for each $(f, X) \in \rho$, $h(s(E), f) = s(X)$, and for each $(d, x) \in \rho$, $h(s(E), d) = s(x)$,
– $(s, h) \vDash P(\boldsymbol{t})$ if $(s, h) \in [\![P(\boldsymbol{t})]\!]$,
– $(s, h) \vDash \Sigma_1 * \Sigma_2$ if there are $h_1, h_2$ such that $h = h_1 \uplus h_2$, $(s, h_1) \vDash \Sigma_1$ and $(s, h_2) \vDash \Sigma_2$.

where the semantics of predicates $[\![P(\boldsymbol{t})]\!]$ is given by the least fixpoint of a monotone operator constructed from the body of rules for $P$ in a standard way, as in [BFGP14].

*Example 15.* Linked list segments are defined by the inductive predicate $\mathsf{ls}(E, F)$,

$$\mathsf{ls}(E, F) \stackrel{\text{def}}{=} (E = F \wedge \mathsf{emp}) \vee (\exists X.\ E \mapsto (\mathtt{next}, X) * \mathsf{ls}(X, F)).$$

In addition, acyclic list segments are defined by the inductive predicate $\mathsf{als}(E, F)$ whose definition is obtained from that of $\mathsf{ls}(E, F)$ by adding $E \neq F$ to the inductive rule. Sorted list segments are defined by the inductive predicate $\mathsf{sls}(E, F, x)$,

$$\begin{aligned}\mathsf{sls}(E, F, x) \stackrel{\text{def}}{=}\ & (E = F \wedge \mathsf{emp}) \vee (\exists X, x'.\ x \leq x' \wedge \\ & E \mapsto ((\mathtt{next}, X), (\mathtt{data}, x)) * \mathsf{sls}(X, F, x')).\end{aligned}$$

And sorted acyclic list segments are defined by the inductive predicate $\mathsf{asls}(E, F, x)$ whose definition is obtained from that of $\mathsf{sls}(E, F, x)$ by adding $E \neq F$ to the inductive rule. Linked list segments with consecutive data values are defined by

$$\begin{aligned}\mathsf{pls}(E, F, x) \stackrel{\text{def}}{=}\ & (E = F \wedge \mathsf{emp}) \vee (\exists X, x'.\ x' = x + 1 \wedge \\ & E \mapsto ((\mathtt{next}, X), (\mathtt{data}, x)) * \mathsf{pls}(X, F, x')).\end{aligned}$$

For the purpose of program verification, the following two decision problems play a vital role, which are the subjects of much of the current research in this area.

- Satisfiability: Given an $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$ formula $\varphi$, decide whether $[\![\varphi]\!]$ is empty.
- Entailment: Given two $\mathsf{SLID}[\mathcal{P}, \mathcal{L}]$ formulae $\varphi, \psi$ such that $\mathsf{Vars}(\psi) \subseteq \mathsf{Vars}(\varphi)$, decide whether $\varphi \vDash \psi$ holds.

We comment that the former problem is fundamental in studying logics, and usually serves as the first task in developing (automated) tool support. The latter question enables automated verification of programs with SL assertions in a Hoare logic style.

Not surprisingly, these questions are challenging, since in general the entailment problem of separation logic with inductive predicates (even without $\mathsf{data}$ constraints) is already undecidable [AGH+14]. Over the past ten years, researchers have developed various techniques to tackle the challenges, by considering different fragments, or utilizing incomplete decision procedures (in particular, by considering heuristics).

*Linearly Compositional Fragment.* In [GCW16], Gu *et al.* defined a *linearly compositional* fragment, where the inductive predicates, as well as the data constraints, must obey certain restrictions. A predicate $P \in \mathcal{P}$ is *linearly compositional* if

- the parameters of $P$ can be divided into three categories: source parameters $E, \boldsymbol{\alpha}$, destination parameters $F, \boldsymbol{\beta}$, and static parameters $\boldsymbol{\xi}$, such that $E, \boldsymbol{\alpha}$ and $F, \boldsymbol{\beta}$ are symmetric, in the sense that the two vectors of parameters are of the same length, and the two parameters in the same positions of the two vectors are of the same data type, in addition, $E, F$ are location variables,
- the inductive definition of $P$ is given by

$$\begin{aligned} P(E, \boldsymbol{\alpha}; F, \boldsymbol{\beta}; \boldsymbol{\xi}) \stackrel{\text{def}}{=} & (E = F \wedge \boldsymbol{\alpha} = \boldsymbol{\beta} \wedge \mathtt{emp}) && (R_0) \\ & \vee (\exists \boldsymbol{X} \exists \boldsymbol{x}.\ \Delta \wedge E \mapsto \rho * P(Y, \boldsymbol{\gamma}; F, \boldsymbol{\beta}; \boldsymbol{\xi})) && (R_1) \end{aligned}$$

The term "linearly compositional" reflects that: (1) $P(E, \boldsymbol{\alpha}; F, \boldsymbol{\beta}; \boldsymbol{\xi})$ can only define linear data structures, for instance, singly or doubly linked lists, lists with tail pointers, (2) $P(E, \boldsymbol{\alpha}; F, \boldsymbol{\beta}; \boldsymbol{\xi})$ satisfies the so-called *composition lemma* $P(E_1, \boldsymbol{\alpha_1}; E_2, \boldsymbol{\alpha_2}; \boldsymbol{\xi}) * P(E_2, \boldsymbol{\alpha_2}; E_3, \boldsymbol{\alpha_3}; \boldsymbol{\xi}) \Rightarrow P(E_1, \boldsymbol{\alpha_1}; E_3, \boldsymbol{\alpha_3}; \boldsymbol{\xi})$, which is essential for deciding the entailment problem by extending the procedure based on graph homomorphism introduced in [CHO+11].

Furthermore, the data formulae are defined as:

$$\Delta \stackrel{\text{def}}{=} \mathsf{true} \mid x \ \mathtt{o} \ c \mid x \ \mathtt{o} \ y + c \mid \Delta \wedge \Delta$$

where $\mathtt{o} \in \{=, \leq, \geq\}$ and $c$ is an integer constant.

We have the following constraints on the inductive rule $(R_1)$:

1. None of the variables from $F, \boldsymbol{\beta}$ occur elsewhere in $R_1$, that is, in $\Delta$, or $E \mapsto \rho$.
2. Each conjunct of $\Delta$ is of the form $\alpha_i \mathbin{\mathfrak{o}} c$, $\alpha_i \mathbin{\mathfrak{o}} \xi_j$, or $\alpha_i \mathbin{\mathfrak{o}} \gamma_i + c$ for $\mathfrak{o} \in \{=, \leq, \geq\}$, $1 \leq i \leq |\boldsymbol{\alpha}| = |\boldsymbol{\gamma}|$, $1 \leq j \leq |\boldsymbol{\xi}|$, and $c \in \mathbb{Z}$.
3. For each $1 \leq i \leq |\boldsymbol{\alpha}|$ such that $\alpha_i$ is a data variable, either $\alpha_i$ occurs in $\rho$, or $\Delta$ contains $\alpha_i = \gamma_i + c$ for some $c \in \mathbb{Z}$.
4. Each variable occurs in $P(Y, \boldsymbol{\gamma}; F, \boldsymbol{\beta}; \boldsymbol{\xi})$ (resp. $\rho$) at most once.
5. All location variables from $\boldsymbol{\alpha} \cup \boldsymbol{\xi} \cup \boldsymbol{X}$ occur in $\rho$.
6. $Y \in \boldsymbol{X}$ and $\boldsymbol{\gamma} \subseteq \{E\} \cup \boldsymbol{X} \cup \boldsymbol{x}$.

The first constraint on $(R_1)$ above is essential to guarantee that $P(E, \boldsymbol{\alpha}; F, \boldsymbol{\beta}; \boldsymbol{\xi})$ satisfies the composition lemma (cf. [ESW15]). We will use $\mathrm{Flds}(P)$ to denote the set of fields occurring in the inductive rule $(R_1)$ of $P$ and $\mathrm{PLFld}(P)$ to denote the unique location field $f$ such that $(f, Y)$ occurs in the inductive rule $(R_1)$ of $P$ (the uniqueness of $f$ is due to the aforementioned 4-th constraint of $(R_1)$).

We write $\mathsf{SLID}_{\mathsf{LC}}[\mathcal{P}]$ for the collection of separation logic formulae $\varphi = \Pi \wedge \Delta \wedge \Sigma$ satisfying the following constraints,

- **linearly compositional predicates**: all predicates from $\mathcal{P}$ are linearly compositional,
- **domination of principal location field**: for each pair of predicates $P_1, P_2 \in \mathcal{P}$, if $\mathrm{Flds}(P_1) = \mathrm{Flds}(P_2)$, then $\mathrm{PLFld}(P_1) = \mathrm{PLFld}(P_2)$,
- **uniqueness of predicates**: there is $P \in \mathcal{P}$ such that each predicate atom of $\Sigma$ is of the form $P(-)$, and for each points-to atom occurring in $\Sigma$, the set of fields of this atom is $\mathrm{Flds}(P)$.

*Example 16.* The inductive predicate $\mathsf{ls}(E, F)$ in Example 15 is linearly compositional, while all the others therein are not. For instance, $\mathsf{als}$ does not satisfy the constraint that the source parameter $F$ occurs only once in the inductive rule, $\mathsf{sls}(E, F, x)$ does not satisfy that the source parameters and destination parameters are symmetric. Nevertheless, the predicates $\mathsf{sls}(E, F, x)$ and $\mathsf{pls}(E, F, x)$ can be adapted into linearly compositional predicates $\mathsf{sls}(F, x; F, x')$ and $\mathsf{pls}'(E, x; F, x')$ by adding one extra destination parameter $x'$,

$$\begin{aligned}\mathsf{sls}'(E, x; F, x') \stackrel{\text{def}}{=}\ & (E = F \wedge x = x' \wedge \mathtt{emp}) \vee (\exists X, x_1.\ x \leq x_1 \wedge \\ & E \mapsto ((\mathtt{next}, X), (\mathtt{data}, x)) * \mathsf{sls}'(X, x_1; F, x')),\end{aligned}$$

$$\begin{aligned}\mathsf{pls}'(E, x; F, x') \stackrel{\text{def}}{=}\ & (E = F \wedge x = x' \wedge \mathtt{emp}) \vee (\exists X, x_1'.\ x_1 = x + 1 \wedge \\ & E \mapsto ((\mathtt{next}, X), (\mathtt{data}, x)) * \mathsf{pls}'(X, x_1; F, x')).\end{aligned}$$

**Theorem 22.** *The following facts hold for* $\mathsf{SLID}_{\mathsf{LC}}[\mathcal{P}]$.

- *The satisfiability problem of* $\mathsf{SLID}_{\mathsf{LC}}[\mathcal{P}]$ *is in NP.*
- *The entailment problem of* $\mathsf{SLID}_{\mathsf{LC}}[\mathcal{P}]$ *formulae is in* $\Pi_3^{\mathrm{P}}$.

It is an interesting open problem to extend the results in Theorem 22 to compositional inductive predicates that are capable of defining non-linear data structures, e.g. trees.

*Semi-decision Procedures for Separation Logic with Inductive Definitions and Data Constriants.* Bouajjani et al. considered a fragment of separation logic with the $\mathsf{ls}$ predicate and data constraints, called SLD (Singly-linked list with Data Logic), where data constraints are specified by universal quantifiers over index variables [BDES12]. They showed that the entailment problem of SLD is undecidable in general, but provided a sound—but incomplete—decision procedure. In addition, they identified a decidable fragment of SLD. The logic SLD in [BDES12] focuses on singly linked lists, and it is unclear how to extend to other linear structures such as doubly linked lists. The decision procedure in [BDES12] is incomplete for fragments that can express list segments where the data values are consecutive. (Note that this can be expressed in the logic $\mathsf{SLID}_{\mathsf{LC}}[\mathcal{P}]$ aforementioned; see $\mathsf{pls}'(E, x; F, x')$ in Example 16.)

In [CDNQ12], Chin *et al.* proposed an entailment checking procedure that can handle *well-founded* predicates (that may be recursively defined) using unfold/fold reasoning. In [LSC16], the authors present a semi-decision procedure for a fragment of separation logic with inductive predicates and Presburger arithmetic. The authors present S2SAT, a decision procedure combining under-approximation and over-approximation for simultaneously checking SAT and UNSAT properties for a sound and complete theory augmented with inductive predicates. To check the *satisfiability* (but not entailment) of a formula, the procedure iteratively unfolds the formula and examines the derived disjuncts. In each iteration, it searches for a proof of either satisfiability or unsatisfiability. They also identify a syntactically restricted fragment of the logic for which the procedure is terminating and thus complete.

*Other Work on Decision Procedures for First-Order Separation Logic with Data Constraints.* Bansal et al. considered first-order separation logic on lists with ordered data and identified the decidability frontier of the satisfiability problem [BBL09]. Very recently, Reynolds et al. proposed a decision procedure for the quantifier-free fragment of first-order separation logic interpreted over heap graphs with data elements ranging over a parametric multi-sorted (possibly infinite) domain [RISK16].

### 8.2 GRASS: Logic of Graph Reachability and Stratified Sets

GRASS stands for logic of **G**raph **R**eachability **A**nd **S**tratified **S**ets, which was introduced by Piskac *et al.* [PWZ13,PWZ14]. The main motivation of these logics is to encode separation logic with inductive predicates into decidable fragments of many-sorted first-order logic, where inductive predicates (e.g. singly linked lists) are encoded by reachability predicates without relying on induction and separating conjunction is encoded by set constraints, and thus offering an SMT-based decision procedure and tool support for separation logic with inductive definitions. An appealing feature of this approach is that the translation into many-sorted first-order logic offers a convenient way to combine shape properties and data constraints, by utilising the Nelson-Oppen framework [NO79].

In the following, we first define the logic GRASS.

**Definition 25 (GRASS logic, [PWZ14]).** *The GRASS logic can be defined as many-sorted first-order logic with the signature* $\Omega_{\mathrm{GS}} = (\mathfrak{S}_{\mathrm{GS}}, \mathfrak{F}_{\mathrm{GS}}, \mathfrak{P}_{\mathrm{GS}})$*, where*

- $\mathfrak{S}_{\mathrm{GS}} = \{\mathsf{node}, \mathsf{field}, \mathsf{set}\}$*;*
- $\mathfrak{F}_{\mathrm{GS}}$ *consists of null* : $\mathsf{node}$, $\mathsf{read} : \mathsf{field} \times \mathsf{node} \rightarrow \mathsf{node}$, $\mathsf{write} : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \rightarrow \mathsf{node}$*, and a countable infinite set of constant symbols for each sort in* $\mathfrak{S}_{\mathrm{GS}}$*;*
- $\mathfrak{P}_{\mathrm{GS}}$ *consists of* $B : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \times \mathsf{node}$ *and* $\in$: $\mathsf{node} \times \mathsf{set}$.

*Semantics.* The semantics of GRASS formulae is defined with respect to a theory $\mathsf{Th}(\mathcal{I}_{\mathrm{GS}})$, where $\mathcal{I}_{\mathrm{GS}}$ is a set of $\Omega_{\mathrm{GS}}$-interpretations such that an $\Omega_{\mathrm{GS}}$-interpretation $I$ is in $\mathcal{I}_{\mathrm{GS}}$ if $I$ satisfies the following conditions.

- $I$ interprets the sort $\mathsf{node}$ as a finite set $\mathsf{node}^I$.
- The sort $\mathsf{field}$ is interpreted as the set of all functions $\mathsf{node}^I \rightarrow \mathsf{node}^I$.
- The sort $\mathsf{set}$ is interpreted as the set of all subsets of $\mathsf{node}^I$.
- The function symbols $\mathsf{read}$ and $\mathsf{write}$ represent field look-up and field update. They must satisfy the following properties,
  - $\forall u \in \mathsf{node}^I$, $f \in \mathsf{field}^I$, $\mathsf{read}^I(f, u) = f(u)$,
  - $\forall u, v \in \mathsf{node}^I$, $f \in \mathsf{field}^I$, $\mathsf{write}^I(f, u, v)$ is the function $f' \in \mathsf{field}^I$ such that for each $w \in \mathsf{node}^I$, if $w = u$, then $f'(w) = u$, otherwise, $f'(w) = f(w)$.
- The between predicate $B(f, x, y, z)$ denotes that $x$ reaches $z$ via an $f$-path that must go though $y$. Formally, $B(f, x, y, z)$ satisfies that for each $(f, u, v, w) \in \mathsf{field}^I \times \mathsf{node}^I \times \mathsf{node}^I \times \mathsf{node}^I$, $B^I(f, u, v, w)$ holds iff $(u, w) \in f^* \wedge (u, v) \in (\{(u_1, f(u_1)) \mid u_1 \in \mathsf{node}^I \wedge u_1 \neq w\})^*$, where $f^*$ is the reflexive and transitive closure of $f$, similarly for $(\{(u_1, f(u_1)) \mid u_1 \in \mathsf{node}^I \wedge u_1 \neq w\})^*$.
- Finally, $\in^I$, the interpretation of $\in$ in $I$, is the set membership relation, that is, for each $u \in \mathsf{node}^I$ and $S \in \mathsf{set}$, $u \in^I S$ holds iff $u$ is an element of $\mathfrak{S}$.

We will use $R(f, x, y)$ as a short-hand for $B(f, x, y, y)$, which intuitively means that there is an $f$-path from $x$ to $y$.

Although the satisfiability problem of GRASS is undecidable in general, decidable fragments have been considered in [PWZ13,PWZ14]. In the following, we will use the fragment of GRASS in [PWZ13] to illustrate the idea, where the specialisation of GRASS to lists was considered and it was shown how to translate separation logic formulae over lists into GRASS. The interested reader can refer to [PWZ14] for the fragment GRIT which is devoted to tree structures.

Let us call the fragment of GRASS in [PWZ13] as $\mathrm{GRASS}_{list}$.

**Definition 26 ($\mathrm{GRASS}_{list}$, [PWZ13]).** *We assume that* $\mathcal{X}$ *is a countably infinite set of variables of sorts* $\mathsf{node}$ *and* $\mathsf{set}$*. We use the lower-case symbols* $x, y \in \mathcal{X}$ *for variables of sort* $\mathsf{node}$ *and upper-case symbols* $X, Y \in \mathcal{X}$ *for variables of sort* $\mathsf{set}$*. In addition, we assume that* $\mathtt{next} \in \mathsf{field}$ *is used to denote the next-pointers between locations in lists. Then the syntax of* $\mathrm{GRASS}_{list}$ *is defined by the following rules,*

$T_L \stackrel{def}{=} x \mid \texttt{next}(T_L)$, *where* $x \in \mathcal{X}$,
$A \stackrel{def}{=} T_L = T_L \mid T_L \xrightarrow{\texttt{next}\setminus T_L} T_L$,
$U \stackrel{def}{=} A \mid \neg U \mid U \wedge U \mid U \vee U$,
$T_S \stackrel{def}{=} X \mid \varnothing \mid T_S \setminus T_S \mid T_S \cup T_S \mid T_S \cap T_S \mid \{x.U\}$, *s.t.* $\texttt{next}(x)$ *does not occur in* $U$,
$B \stackrel{def}{=} T_S = T_S \mid T_L \in T_S$,
$F \stackrel{def}{=} A \mid B \mid \neg F \mid F \wedge F \mid F \vee F$.

*A GRASS formula is a propositional combination of atoms. There are two types of atoms.*

- *Atoms of type A are either equalities between terms of type* $T_L$ *and* reachability predicates. *The terms of type* $T_L$ *represent nodes in the graph. They are associated with the sort* node *and are constructed from variables and application of* next. *Reachability predicates* $t_1 \xrightarrow{\texttt{next}\setminus t_3} t_2$ *intuitively means that there is a path in the graph that from* $t_1$ *to* $t_2$ without *going through* $t_3$.
- *Atoms of type B are equalities between terms of sort* set *and membership tests. Terms of type* set *represent stratified sets*[1], *i.e., their elements are interpreted as nodes in the graph. Terms of sort* set *include* set comprehensions *of the form* $\{x.U\}$, *where* $U$ *is a Boolean combination of atoms of type A.*

We will use $t_1 \xrightarrow{\texttt{next}} t_2$ as an abbreviation of $t_1 \xrightarrow{\texttt{next}\setminus t_2} t_2$, which intuitively means that $t_2$ is reachable from $t_1$ by following the field next. In addition, we use $t_1 \neq t_2$ as an abbreviation of $\neg(t_1 = t_2)$. For a variable $x \in \mathcal{X}$, we use $\{x\}$ to denote the singleton set $\{y.\ y = x\}$. The side condition that $\texttt{next}(x)$ does not occur in $U$ in the terms $\{x.\ U\}$ is important to ensure the decidability of the logic.

Note that GRASS$_{list}$ includes some syntactic sugar that is not in GRASS defined above. We will illustrate how this syntactic sugar can be casted into the original definition of GRASS.

- $\texttt{next}(t) \equiv \texttt{read}(\texttt{next}, t)$,
- $t_1 \xrightarrow{\texttt{next}\setminus t_3} t_2 \equiv R(\texttt{next}, t_1, t_2) \wedge \forall x.\ (B(\texttt{next}, t_1, x, t_2) \wedge x \neq t_2) \rightarrow x \neq t_3$,
- Set operations can be reformulated into GRASS as well. For instance, $(X_1 \setminus X_2) \cup X_3 = Y \equiv \forall x.\ ((x \in X_1 \wedge \neg x \in X_2) \vee x \in X_3) \leftrightarrow x \in Y$, and $X = \{x\} \equiv x \in X \wedge \forall y.\ y \in X \leftrightarrow y = x$.

We use $X = Y \uplus Z$ as an abbreviation of the formula $X = Y \cup Z \wedge Y \cap Z = \varnothing$, which intuitively means that $X$ is the disjoint union of $Y$ and $Z$.

*Example 17.* Consider the formula $F \equiv Y = \{x.\ x \xrightarrow{\texttt{next}} y\} \wedge Z = \{x.\ x \xrightarrow{\texttt{next}} z\} \wedge X = Y \uplus Z$. This formula expresses that the subgraph of the heap graph induced by the set of nodes $X$ comprises two disjoint connected components, one in which all nodes reach $y$, and one in which all nodes reach $z$.

[1] The notion of stratified sets comes from [Zar03].

**Theorem 23** ([PWZ13]). *The satisfiability problem of $GRASS_{list}$ is NP-complete.*

In the following, we first illustrate how SLID[als], i.e. the logic SLID with only a single inductive predicate $\mathsf{als}(x, y)$ and no data constraints (cf. Example 15 for the definition of als, adapted slightly by replacing $E, F$ with $x, y$), can be translated into $\text{GRASS}_{list}$. Specifically, SLID[als] formulae $\Pi \wedge \Sigma$ are defined by the following rules,

$$\Pi \stackrel{\text{def}}{=} x = y \mid x \neq y \mid \Pi \wedge \Pi, \qquad \Sigma \stackrel{\text{def}}{=} x \mapsto (\mathtt{next}, y) \mid \mathsf{als}(x, y) \mid \Sigma * \Sigma,$$

where $\Pi$ and $\Sigma$ are called the pure and spatial formulae respectively.

The translation is done by induction the syntax of SLID[als] formulae. Since the translation of pure formulae is trivial (the identity translation), we only describe the translation of spatial formulae, denoted by $tr_X(\Sigma)$ (where $X$ denotes the set of locations), below.

- $tr_X(\mathtt{emp}) \stackrel{\text{def}}{=} X = \varnothing$,
- $tr_X(x \mapsto (\mathtt{next}, y)) \stackrel{\text{def}}{=} X = \{x\} \wedge \mathtt{next}(x) = y$,
- $tr_X(\mathsf{als}(x, y)) \stackrel{\text{def}}{=} x \xrightarrow{\mathtt{next}} y \wedge X = \{z.\ x \xrightarrow{f \setminus y} z \wedge z \neq y\}$,
- $tr_X(\Sigma_1 * \Sigma_2) \stackrel{\text{def}}{=} X = X_1 \uplus X_2 \wedge tr_{X_1}(\Sigma_1) \wedge tr_{X_2}(\Sigma_2)$, where $X_1$ and $X_2$ are two fresh variables of sort set.

As a matter of fact, we can even translate the Boolean combination of SLID[ls] formulae into $\text{GRASS}_{list}$ since $\text{GRASS}_{list}$ is closed under negations.

*Extension with Data Constraints.* One notable feature of this translation of separation logic formulae into $\text{GRASS}_{list}$ is that it offers a convenient way to specify and reason about data constraints in dynamic data structures, by using the Nelson-Oppen combination framework. To support reasoning about data constraints, we extend the signature of $\text{GRASS}_{list}$ with an additional sort data for data values, data fields interpreted as the functions from $\mathsf{node}^I$ to $\mathsf{data}^I$, and sets with data elements. The read and write functions are extended accordingly. In the following, we assume that there is a unique data field $d$ and we use $d(x)$ to denote the value of a node $x$ corresponding to field $d$.

We can combine $\text{GRASS}_{list}$ with any decidable quantifier-free first-order theory that is signature-disjoint from $\text{GRASS}_{list}$ and stably-infinite to interpret the data sort. The extensions that we discuss build on such quantifier-free combinations. [PWZ14] considers three categories of extensions with data: (1) monadic predicates on the data value of one node, (2) binary predicates between the data values of two distinct nodes, and (3) constraints on the content of data structures, that is, sets of data values occurring in data structures.

- Monadic predicates. These predicates are able to express properties such as upper and lower bounds on the values contained in a tree. Such formulae have the following form: $\forall x.x \in X \rightarrow Q(d(x))$ where $Q$ is a monadic predicate over data and $X$ a variable of sort set. This class of formulae also forms a

so-called **$\Psi$-local theory extension** [IJS08]. Then we can slightly adapt the decision procedure for $\text{GRASS}_{list}$ to obtain a complete decision procedure for the extension of $\text{GRASS}_{list}$ with monadic-predicate data constraints.
- Binary predicates. These predicates are introduced to define, for instance, a sorted linked list, in which we need to relate data values in two distinct nodes. To ensure completeness of the decision procedure, the binary predicates must satisfy that the expressed binary relations are *transitive* as well as some other constraints. (They are too technical to be stated here clearly. Those who are interested can read Sect. 7 of [PWZ14] for these additional constraints) One typical transitive binary predicate is the order relation between data values. The transitivity requirement prevents us from expressing data constraints involving counting, e.g., length constraints or multiset constraints. With binary predicates, we can express the sortedness property as follows: $\forall x, y \in X.\ x \xrightarrow{\texttt{next}} y \rightarrow d(x) \leq d(y)$.
- Set constraints. This class of extensions enables reasoning about functional correctness properties. Essentially a way of referring to the content of lists is needed. While one can define the content of a list whose footprint is $X$ as $C(X) = \{z \mid \exists x \in X.\ z = x.d\}$. This definition goes beyond $\text{GRASS}_{list}$, due to the existential quantifier appearing inside the set comprehension. In [PWZ13], the authors proposed a solution by adding a witness function that maps a data value back to a node in the graph which stores the data value. They define the witness function in an axiomatic way, any show that the axioms still give a $\Psi$-local theory extension.

*Limitations of This Approach.* Unfortunately, there is no precise characterization of the limit of extensions that preserve the property of local theory extensions on which the decision procedure is built. However, not all extensions are local, in particular, the constraints involving counting, e.g. length constraints and multiset constraints.

**Other Works on First-/Second-Order Logics Combining Shape Properties and Data Constraints.** Bouajjani et al. proposed a fragment of many-sorted first-order logic with reachability predicates, called CSL (Composite Structure Logic [BDES09]), to reason about programs manipulating composite dynamic data structures. The formulae in CSL allow a limited form of alternation between existential and universal quantifiers and they can express constraints on reachability between positions in the heap following some pointer fields, linear constraints on the lengths of the lists, as well as constraints on the data values attached to these positions. For data constraints, the logic CSL is parameterized by a first-order logic over the associated data domain. They proved that the satisfiability problem of CSL is decidable whenever the underlying data logic is decidable. In addition, Madhusudan et al. defined a fragment of monadic second-order logic, called STRAND (STRucture ANd Data), to reason about both shape properties and data constraints, in tree structures [MPQ11]. While the satisfiability of STRAND logic is undecidable in general, several decidable fragments were identified in [MPQ11].

### 8.3 Streaming Transducers

In [AC11], Alur and Cerny proposed *streaming transducers* to show that for a class of single-pass list processing programs, the equivalence problem of the programs in this class is decidable. The intuition of streaming transducers is to model linked lists as data words, use a set of data word variables to store some intermediate information for the outputs, and at the same time use a set of registers ranging over an ordered data domain to guide the control flow of transducers. In the following, we present the definition of streaming transducers and some basic facts known for streaming transducers.

Let $\mathbb{D}$ be an infinite set of data values. We will use $<$ to denote the strict total order over $\mathbb{D}$. Examples of $(\mathbb{D}, <)$ include $(\mathbb{Z}, <)$, the set of integers with the order relation, and $(\mathbb{Q}, <)$, the set of rational numbers with the order relation. As for NRAs in Sect. 3, let $R$ be a set of registers and $\mathsf{cur} \notin R$ be a distinguished register to denote the data value in the current position, in addition, let $R^{\copyright}$ denote $R \cup \{\mathsf{cur}\}$. A *guard* formula over $R$ is defined by the rules $g \stackrel{\text{def}}{=} \mathsf{true} \mid \mathsf{false} \mid \mathsf{cur}\ \mathfrak{o}\ r \mid g \wedge g \wedge g \vee g$, where $r \in R$ and $\mathfrak{o} \in \{=, \neq, <, >\}$. Let $\mathsf{G}_R$ denote the set of guards over $R$. Let $\rho$ be a valuation that assigns each $r \in R$ a data value from $\mathbb{D}$, and $d \in \mathbb{D}$. Then $\rho[d/\mathsf{cur}]$ satisfies a guard $g$, denoted by $\rho[d/\mathsf{cur}] \models g$, is defined as follows:

- $\rho[d/\mathsf{cur}] \models \mathsf{cur} = r$ if $d = \rho(r)$, similarly for $\rho[d/\mathsf{cur}] \models \mathsf{cur} \neq r$, $\rho[d/\mathsf{cur}] \models \mathsf{cur} < r$, and $\rho[d/\mathsf{cur}] \models \mathsf{cur} > r$,
- $\rho[d/\mathsf{cur}] \models g \wedge g$ and $\rho[d/\mathsf{cur}] \models g \vee g$ are defined in a standard way.

**Definition 27 (Streaming transducers).** *A streaming transducer (ST) $\mathcal{S}$ is a tuple $(Q, \Sigma, \Gamma, R, X, q_0, \tau_0, \delta, O)$, where:*

- *$Q$ is a finite set of states,*
- *$R$ is a finite set of registers,*
- *$X$ is a finite set of data word variables,*
- *$q_0 \in Q$ is the initial state,*
- *$\tau_0 : R \to \mathbb{D}$ assigns each register an initial data value,*
- *$\delta$ is a finite set of transitions comprising the tuples $(q, \sigma, g, q', \alpha)$, where $q, q' \in Q$, $\sigma \in \Sigma$, $g$ is a guard on $R$, $\alpha$ is a* **function** *(instead of a partial function) which assigns each $r \in R$ a variable $r' \in R^{\copyright}$, and assigns each $x \in X$ a sequence from $((\Gamma \times R^{\copyright}) \cup X)^*$,*
- *$O$ is a partial output function from $Q$ to $((\Gamma \times R) \cup X)^*$.*

*In addition, $\mathcal{S}$ satisfies the following constraints.*

- **deterministic***: for each pair of distinct transitions $(q, \sigma, g_1, q_1, \alpha_1), (q, \sigma, g_2, q_2, \alpha) \in \delta$, it holds that $g_1 \wedge g_2$ is unsatisfiable,*
- **copyless***: for each $q \in Q$ and $x \in X$, there is at most one occurrence of $x$ in $O(q)$, in addition, for each $x \in X$ and $(q, \sigma, g, q', \alpha) \in \delta$, there is at most one occurrence of $x$ in the set of words $\{\alpha(y) \mid y \in X\}$.*

*Semantics of STs.* Given a data word $w = (\sigma_1, d_1)\dots(\sigma_n, d_n)$ and an ST $\mathcal{S} = (Q, \Sigma, \Gamma, R, X, q_0, \delta, O)$, a *configuration* of $\mathcal{S}$ on $w$, is a pair $(i, \rho)$, where $\rho$ is a *valuation* $\rho$ on $R \cup X$, that is, a function which assigns each $r \in R$ a data value from $\mathbb{D}$, and assigns each $x \in X \cap \mathsf{dom}(\rho)$ a data word over the alphabet $\Gamma$. The initial configuration is $(q_0, \rho_0)$ where $\rho_0(r) = \tau_0(r)$ for each $r \in R$, and $\rho_0(x) = \varepsilon$ for each $x \in X$. A configuration $(q', \rho')$ is said to be a successor of another configuration $(q, \rho)$, denoted by $(q, \rho) \longrightarrow (q', \rho')$, if there are $d \in \mathbb{D}$ and a transition $(q, \sigma, g, q', \alpha) \in \delta$ such that $\rho[d/\mathsf{cur}] \models g$ and for each $r \in R$, $\rho'(r) = (\rho[d/\mathsf{cur}])(\alpha(r))$, and for each $x \in X$, $\rho'(x) = (\rho[d/\mathsf{cur}])(\alpha(x))$, where $(\rho[d/\mathsf{cur}])(\alpha(x))$ is obtained from $\alpha(x)$ by replacing each occurrence of $y \in R \cup X$ in $\alpha(x)$ with $(\rho[d/\mathsf{cur}])(y)$. A *run* of $\mathcal{S}$ on $w$ is a sequence of configurations $(q_0, \rho_0)(q_1, \rho_1)\dots(q_n, \rho_n)$ such that $(q, \rho_i) \longrightarrow (q, \rho_{i+1})$ for each $i : 0 \leq i < n$. Note that since $\mathcal{S}$ is deterministic, there is at most one run of $\mathcal{S}$ on $w$. The *output* of $\mathcal{S}$ on $w$, denoted by $\mathcal{S}(w)$, is defined as $\rho_n(O(q_n))$, if there is a run of $\mathcal{S}$ on $w$, say $(q_0, \rho_0)(q_1, \rho_1)\dots(q_n, \rho_n)$, such that $O(q_n)$ is defined, otherwise, $\mathcal{S}(w)$ is undefined.

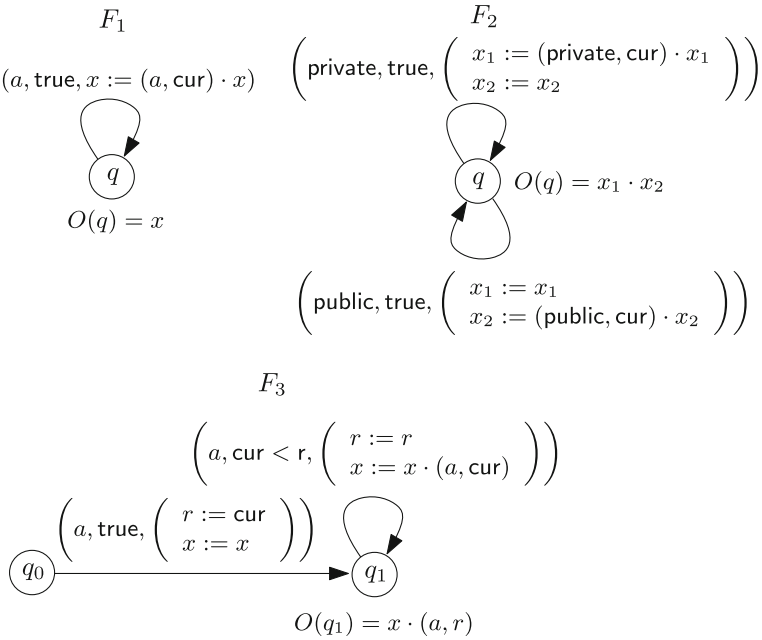



**Fig. 8.** Examples of streaming transducers

*Example 18.* Here are a few examples of streaming transducers (see Fig. 8).

– Let $\Sigma = \{a\}$. Let $F_1$ be the transduction that reverses a data word. Then $F_1$ is defined by an ST $\mathcal{S}_1 = (\{q\}, \Sigma, \Sigma, R = \varnothing, X = \{x\}, q, \delta, O)$, where
  - $\delta = \{(q, a, \mathsf{true}, q, \alpha)\}$ such that $\alpha(x) = (a, \mathsf{cur}) \cdot x$,
  - $O(q) = x$.

- Let $\Sigma = \{\mathsf{private}, \mathsf{public}\}$. Let $F_2$ be the transduction that outputs $w_1 \cdot w_2$ from $w$, where $w_1$ and $w_2$ are the subsequences of $w$ that contain the $\mathsf{private}$ and $\mathsf{public}$ entries respectively. Then $F_2$ is defined by an ST $\mathcal{S}_2 = (\{q\}, \Sigma, \Sigma, R = \varnothing, X = \{x_1, x_2\}, q, \delta, O)$, where
  - $\delta = \{(q, \mathsf{private}, \mathsf{true}, q, \alpha_1), (q, \mathsf{public}, \mathsf{true}, q, \alpha_2)\}$ such that $\alpha_1(x_1) = x_1 \cdot (\mathsf{private}, \mathsf{cur})$, $\alpha_1(x_2) = x_2$, $\alpha_2(x_1) = x_1$, and $\alpha_2(x_2) = x_2 \cdot (\mathsf{public}, \mathsf{cur})$,
  - $O(q) = x_1 \cdot x_2$.
- Let $\Sigma = \{a\}$. Let $F_3$ be the transduction to move the data value in the first position to the last position, provided that the sequence of data values in the data word, except the data value in the first position, is sorted, in addition, all these data values are less than the data value in the first position. Then $F_3$ is defined by an ST $\mathcal{S}_3 = (\{q_0, q_1\}, \Sigma, \Sigma, R = \{r\}, X = \{x\}, q_0, \delta, O)$, where:
  - $\delta = \{(q_0, \mathsf{true}, q_1, \alpha_1), (q_1, \mathsf{cur} < r, q_1, \alpha_2)\}$ such that $\alpha_1(r) = \mathsf{cur}$, $\alpha_1(x) = x$, $\alpha_2(r) = r$, and $\alpha_2(x) = x \cdot (a, \mathsf{cur})$.
  - $O(q_1) = x \cdot (a, r)$.

It is easy to check that each of $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ defined above satisfies the copyless constraint.

STs are said to be closed under composition if for each pair of STs $\mathcal{S}_1$ with the input/output alphabet $\Sigma/\Gamma$, and $\mathcal{S}_2$ with the input/output alphabet $\Gamma/\Pi$, there is an ST $\mathcal{S}$ such that for each data word $w$ over the alphabet $\Sigma$, it holds that $\mathcal{S}(w) = \mathcal{S}_2(\mathcal{S}_1(w))$.

The equivalence problem of SNTs: Given two STs $\mathcal{S}_1$ and $\mathcal{S}_2$, decide whether they are equivalent, in the sense that for each data word $w$, $\mathcal{S}_1(w) = \mathcal{S}_2(w)$.

**Theorem 24** ([AC11]). *The following results hold for streaming transducers:*

- *STs are not closed under composition.*
- *The equivalence problem of STs is PSPACE-complete.*
- *The equivalence problem of the two-way extension of STs is undecidable.*

The PSPACE-hardness of the equivalence problem follows from the fact that the equivalence of DRAs is PSPACE-hard (cf. Theorem 2).

At last, we would like to remark that since its introduction, most of the work on streaming transducers focus on finite alphabets, see e.g. [AC10, AD12, ADGT13].

*Other Automata Models to Reason About Dynamic Data Structures with Data Constraints.* Forest automata were also extended with order constraints to reason about the behaviour of programs manipulating dynamic data structures, where a sound but incomplete procedure was proposed to decide the language inclusion problem of two forest automata [AHJ+13].

## 9 Formalisms for Analysing Programs in the MapReduce Framework

The MapReduce framework is a popular programming model proposed by Dean and Ghemawat from Google Inc. for data-parallel computations [DG04]. Since its introduction, various data-parallel computing platforms based on the MapReduce framework, e.g. Apache Hadoop[2], Apache SPARK[3], Microsoft SCOPE [CJL+08], Yahoo! Pig Latin [ORS+08], and Facebook Hive [TSJ+09], have appeared and a huge number of big-data processing jobs are executed on these platforms daily.

In the MapReduce framework, the *reducer* produces an output from a list of inputs. Due to the scheduling policy of the platform, the inputs may arrive at the reducers in different order. The *commutativity problem* of reducers asks if the output of a reducer is independent of the order of its inputs. A formal analysis of the commutativity problem of reducers in the MapReduce framework was first considered in [CHSW15], where it was shown that (1) the commutativity problem is undecidable in general, if multiplication operators are available, and (2) if the data domain is a finite set, then the commutativity problem is decidable and reduced to the equivalence problem of two-way finite-state automata.

Very recently, Chen et al. proposed a model of reducers, called streaming numerical transducers (SNTs), and extended the decidability result in [CHSW15] to the infinite data domain [CSW16]. The model of SNTs originates from the observation that in practice MapReduce programs are usually used for data analytics and thus require very simple control flow. By exploiting this simplicity, in SNTs, the control and data flow of programs are separated and arithmetic operations are disallowed in the control flow. The design of SNTs is inspired by streaming transducers [AC11] (see Sect. 8.3). Nevertheless, the two models are intrinsically different since the outputs of SNTs are integers while those of streaming transducers are data words.

In this section, as in symbolic automata, we assume data words are elements of $\mathbb{D}^*$. In addition, we assume this data domain is the integer domain $\mathbb{Z}$. An SNT scans a data word $w = d_1 \dots d_n$ from left to right, records and aggregates information in variables, and outputs an integer when it finishes reading the data word.

Let $Z$ be a set of variables. Then an *expression* over $Z$ is defined recursively by the following rules: $e \in \mathsf{E}_Z \stackrel{\text{def}}{=} c \mid z \mid (e + e) \mid (e - e)$, where $z \in Z$ and $c \in \mathbb{Z}$. We use $\mathsf{E}_Z$ to denote the set of all possible expressions over $Z$. For an expression $e$, let $\mathsf{var}(e)$ denote the set of variables in $e$. Given a set of expressions $E$, we also use $\mathsf{var}(E)$ to denote the set of all variables appeared in $E$, i.e., $\mathsf{var}(E) = \bigcup_{e \in E} \mathsf{var}(e)$. A *guard* over $Z$ is defined recursively by the following rules: $g \in \mathsf{G}_Z \stackrel{\text{def}}{=} \mathsf{true} \mid v < v \mid v = v \mid v > v \mid g \wedge g$, where $v \in Z \cup \mathbb{Z}$. We use $\mathsf{G}_Z$ to denote the set of guards over $Z$. A *guarded expression* over $Z$ is a pair $(g, e) \in \mathsf{G}_Z \times \mathsf{E}_Z$.

[2] http://hadoop.apache.com.
[3] http://spark.apache.com.

A *valuation* $\rho$ of $Z$ is a function from $Z$ to $\mathbb{Z}$. The value of an expression $e \in \mathsf{E}_Z$ under a valuation $\rho$ over $Z$, denoted by $[\![e]\!]_\rho$, is defined recursively in the standard way. Let $\rho$ be a valuation of $Z$ and $g$ be a guard in $\mathsf{G}_Z$. Then $\rho$ satisfies $g$, denoted by $\rho \models g$, iff $g$ is evaluated to $\mathsf{true}$ under $\rho$. We say that a guard $g$ is *satisfiable* if there exists a valuation $\rho$ satisfying $g$.

**Definition 28 (Streaming numerical transducers).** *A streaming numerical transducer (SNT) $\mathcal{S}$ is a tuple $(Q, R, X, \delta, q_0, O)$, where $Q$ is a finite set of states, $R$ is a finite set of* control registers*, $X$ is a finite set of* data variables*, $\delta$ is the set of transitions, $q_0 \in Q$ is the initial state, $O$ is the output function, which is a total function from $Q$ to $2^{\mathsf{G}_R \times \mathsf{E}_{R \cup X}}$, i.e. $O(q)$ for $q \in Q$ is a finite set of guarded expressions over $X \cup Y$ where the guards only put constraints on $R$. In addition, a distinguished register $\mathsf{cur} \notin R$ is used to denote the data value in the current position. For convenience, let $R^{\copyright}$ denote $R \cup \{\mathsf{cur}\}$.*

*The set of transitions $\delta$ comprises the tuples $(q, g, \eta, q')$, where $q, q' \in Q$, $g$ is a* guard *over $R^{\copyright}$, and $\eta$ is an assignment function which is a partial function from $R \cup X$ to $E_{R^{\copyright} \cup X}$ such that for each $r \in \mathsf{dom}(\eta) \cap R$, $\eta(r) \in R^{\copyright}$. Informally, $\eta$ maps a data variable to an expression over $R^{\copyright} \cup X$ and a control register to either $\mathsf{cur}$ or another control register. We write $q \xrightarrow{(g,\eta)} q'$ to denote $(q, g, \eta, q') \in \delta$ for convenience. Moreover, we assume that an SNT $\mathcal{S}$ is* deterministic*. That is, (1) for each pair of distinct transitions originating from $q$, say $(q, g_1, \eta_1, q_1')$ and $(q, g_2, \eta_2, q_2')$, it holds that $g_1 \wedge g_2$ is unsatisfiable, (2) for any state $q \in Q$ and each pair of distinct guarded expressions $(g_1, e_1)$ and $(g_2, e_2)$ in $O(q)$, it holds that $g_1 \wedge g_2$ is unsatisfiable.*

*Semantics of SNTs.* The semantics of an SNT $\mathcal{S}$ is defined as follows. A *configuration* of $\mathcal{S}$ is a pair $(q, \rho)$, where $q \in Q$ and $\rho$ is a valuation of $R \cup X$. An *initial* configuration of $\mathcal{S}$ is $(q_0, \rho_0)$, where $\rho_0$ assigns zero to all variables in $R \cup X$. A sequence of configurations $\mathcal{R} = (q_0, \rho_0)(q_1, \rho_1) \dots (q_n, \rho_n)$ is a *run* of $\mathcal{S}$ over a data word $w = d_1 \dots d_n$ iff there exists a *path* (sequence of transitions) $P = q_0 \xrightarrow{(g_1,\eta_1)} q_1 \xrightarrow{(g_2,\eta_2)} q_2 \dots q_{n-1} \xrightarrow{(g_n,\eta_n)} q_n$ such that for each $i \in [n+1]$, $\rho_{i-1}[d_i/\mathsf{cur}] \models g_i$, and $\rho_i$ is obtained from $\rho_{i-1}$ as follows: (1) For each $r \in R$, if $r \in \mathsf{dom}(\eta_i)$, then $\rho_i(r) = [\![\eta_i(r)]\!]_{\rho_{i-1}[d_i/\mathsf{cur}]}$, otherwise $\rho_i(r) = \rho_{i-1}(r)$. (2) For each $x \in X$, if $x \in \mathsf{dom}(\eta_i)$, then $\rho_i(x) = [\![\eta_i(x)]\!]_{\rho_{i-1}[d_i/\mathsf{cur}]}$, otherwise, $\rho_i(x) = \rho_{i-1}(x)$. We call $(q_n, \rho_n)$ the *final configuration* of the run. In this case, we also say that the run $\mathcal{R}$ follows the path $P$. We say that a path $P$ in $\mathcal{S}$ is *feasible* iff there exists a run of $\mathcal{S}$ following $P$. Given a data word $w = d_1 \dots d_n$, if there is a run of $\mathcal{S}$ over $w$ from $(q_0, \rho_0)$ to $(q_n, \rho_n)$ and there exists a guarded expression $(g, e) \in O(q_n)$ such that $\rho_n \models g$, then the output of $\mathcal{S}$ over $w$, denoted by $\mathcal{S}(w)$, is $[\![e]\!]_{\rho_n}$. Otherwise, $\mathcal{S}(w)$ is undefined, denoted by $\perp$.

*Example 19 (SNT for max).* The SNT $\mathcal{S}_{\max}$ for computing the maximum value of an input data word is defined as $(\{q_0, q_1, q_2\}, \{\mathsf{max}\}, \varnothing, \delta, q_0, O)$, where the set of transitions $\delta$ and the output function $O$ are illustrated in Fig. 9 (here $R = \{\mathsf{max}\}$, $X = \varnothing$, and $\mathsf{max} := \mathsf{cur}$ denotes the assignment of $\mathsf{cur}$ to the variable $\mathsf{max}$).

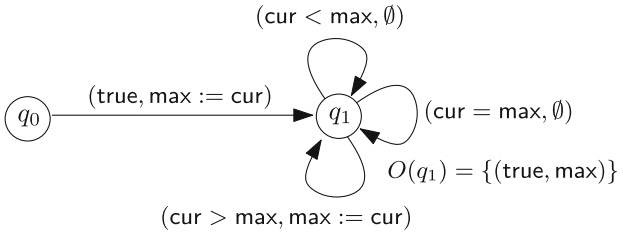



**Fig. 9.** The SNT $\mathcal{S}_{\max}$ for computing the maximum value

We focus on three decision problems of SNTs defined as follows: (1) *Commutativity*: Given an SNT $\mathcal{S}$, decide whether $\mathcal{S}$ is commutative, that is, whether for each data word $w$ and each permutation $w'$ of $w$, $\mathcal{S}(w) = \mathcal{S}(w')$. (2) *Equivalence*: Given two SNTs $\mathcal{S}, \mathcal{S}'$, decide whether $\mathcal{S}$ and $\mathcal{S}'$ are equivalent, that is, whether over each data word $w$, $\mathcal{S}(w) = \mathcal{S}'(w)$. (3) *Non-zero output*: Given an SNT $\mathcal{S}$, decide whether $\mathcal{S}$ has a non-zero output, that is, whether there exists a data word $w$ such that $\mathcal{S}(w) \notin \{\bot, 0\}$.

**Theorem 25 ([CSW16, CLTW16]).** *The commutativity, equivalence, and non-zero output problem of SNTs can be decided in exponential time.*

In [CSW16, CLTW16], Theorem 25 was proved as follows:

1. The commutativity problem of SNTs is reduced to the equivalence problem of SNTs in polynomial time, which can be further reduced to the non-zero output problem of SNTs in polynomial time.
2. Then it is shown that the non-zero output problem of SNTs can be decided in exponential time, by extending Karr's algorithm for computing affine relationships in affine programs [MS04].

*Further Reading.* Recently, Neven et al. proposed variants of register automata and transducers as formal models for the distributed evaluation of relational algebra on relational databases in MapReduce framework [NSST15]. They introduced three models and investigated the expressibility issues.

## 10 Conclusion

This chapter has provided a tutorial and survey on the state of the art of automata models and logics to reason about the behaviour of software systems which embrace data values from an infinite domain. We have presented the models with different mechanisms to deal with infinite data values, register automata (and related logics), data automata (and related logics), pebble automata, and symbolic automata and transducers. In addition, we included two application-oriented sections, on formal models to reason about programs manipulating dynamic data structures and for the static analysis of data-parallel

programs respectively. For these two sections, we presented separation logic with data constraints, logic of graph reachability and stratified sets, streaming transducers, and streaming numerical transducers. For each model, we introduced the basic definitions, used some examples to illustrate the model, and stated the main theoretical properties of the model.

For the perspectives of this field, in our opinion, researchers should strengthen the connections of the models with applications to better motivate, or to achieve greater impact of, their work. In particular, symbolic automata and transducers, separation logic with data constraints, and streaming numerical transducers are the formalisms that are better motivated by applications. These formalisms are still the research focus in the verification and database community. In addition, in order to produce practical tools to solve industrial-scale problems, there are still various challenges, and interested readers are encouraged to work on, and contribute to, this promising field.

**Acknowledgments.** Taolue Chen is partially supported by EPSRC grant (EP/P00430X/1), European CHIST-ERA project SUCCESS, ARC Discovery Project (DP160101652), Singapore MoE AcRF Tier 2 grant (MOE2015-T2-1-137), NSFC grant (No. 61662035), and an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2014A14). Fu Song is partially supported by Shanghai Pujiang Program (No. 14PJ1403200), Shanghai ChenGuang Program (No. 13CG21), and NSFC Projects (Nos. 61402179, 61532019 and 91418203). Zhilin Wu is partially supported by the NSFC projects (Nos. 61100062, 61272135, 61472474, and 61572478).

## References

[AC10] Alur, R., Cerný, P.: Expressiveness of streaming string transducers. In: Proceedings of the 30th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, pp. 1–12 (2010)

[AC11] Alur, R., Cerny, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 599–610 (2011)

[ACW12] Alur, R., Cerný, P., Weinstein, S.: Algorithmic analysis of array-accessing programs. ACM Trans. Comput. Log. **13**(3), 27 (2012)

[AD12] Alur, R., D'Antoni, L.: Streaming tree transducers. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 42–53. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31585-5_8

[ADGT13] Alur, R., Durand-Gasselin, A., Trivedi, A.: From monadic second-order definable string transformations to transducers. In: Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 458–467 (2013)

[AGH+14] Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 411–425. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54830-7_27

[AHJ+13] Abdulla, P.A., Holik, L., Jonsson, B., Lengal, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. In: Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 224–239 (2013)

[BBL09] Bansal, K., Brochenin, R., Lozes, E.: Beyond shapes: lists with ordered data. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 425–439. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00596-1_30

[BCO05] Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005). doi:10.1007/11575467_5

[BDES09] Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04081-8_13

[BDES12] Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 167–182. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33386-6_14

[BDM+11] Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Logic **12**(4), 27 (2011)

[BFGP14] Brotherston, J., Fuhs, C., Gorogiannis, J.N., Perez, A.N.: A decision procedure for satisfiability in separation logic with inductive predicates. In: Proceedings of the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (2014)

[BKLT13] Bojańczyk, M., Klin, B., Lasota, S., Toruńczyk, S.: Turing machines with atoms. In: Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 183–192 (2013)

[BL12] Bojanczyk, M., Lasota, S.: An extension of data automata that captures XPath. Log. Methods Comput. Sci. **8**(1), 1–28 (2012)

[BMS+06] Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS), pp. 7–16 (2006)

[Boj13] Bojańczyk, M.: Modelling infinite structures with atoms. In: Libkin, L., Kohlenbach, U., de Queiroz, R. (eds.) WoLLIC 2013. LNCS, vol. 8071, pp. 13–28. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39992-3_3

[BS10] Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. **411**(4–5), 702–715 (2010)

[BSSS06] Bojańczyk, M., Samuelides, M., Schwentick, T., Segoufin, L.: Expressive power of pebble automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 157–168. Springer, Heidelberg (2006). doi:10.1007/11786986_15

[Büc60] Büchi, J.R.: Weak second-order arithmetic and finite automata. Z. Math. Log. Grundl. Math. **6**, 66–92 (1960)

[Büc62] Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proceedings of the 1960 International Congress for Logic, Methodology and Philosophy of Science, pp. 1–11. Stanford University Press (1962)

[CDNQ12] Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)

[CDOY11] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011)
[CHO+11] Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23217-6_16
[CHSW15] Chen, Y.-F., Hong, C.-D., Sinha, N., Wang, B.-Y.: Commutativity of reducers. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 131–146. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_9
[CJL+08] Chaiken, R., Jenkins, B., Larson, P.Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. PVLDB **1**(2), 1265–1276 (2008)
[CK98] Cheng, E.Y.C., Kaminski, M.: Context-free languages over infinite alphabets. Acta Inf. **35**(3), 245–267 (1998)
[CLTW16] Chen, Y.-F., Lengal, O., Tan, T., Wu, Z.: Equivalence of streaming numerical transducers (2016). (manuscript)
[CSW16] Chen, Y.-F., Song, L., Wu, Z.: The commutativity problem of the MapReduce framework: a transducer-based approach. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 91–111. Springer, Cham (2016). doi:10.1007/978-3-319-41540-6_6
[D'A12] D'Antoni, L.: In the maze of data languages. CoRR, abs/1208.5980 (2012)
[DA14] D'Antoni, L., Alur, R.: Symbolic visibly pushdown automata. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 209–225. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_14
[DG04] Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI), pp. 137–150 (2004)
[DHLT14] Decker, N., Habermehl, P., Leucker, M., Thoma, D.: Ordered navigation on multi-attributed data words. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 497–511. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44584-6_34
[DL09] Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. **10**(3), 16:1–16:30 (2009)
[DOY06] Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006). doi:10.1007/11691372_19
[DV14] D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 541–554 (2014)
[DV15] D'Antoni, L., Veanes, M.: Extended symbolic finite automata and transducers. Form. Methods Syst. Des. **47**(1), 93–119 (2015)
[Elg61] Elgot, C.: Decision problems of finite automata design and related arithmetic. Trans. Am. Math. Soc. **98**, 21–52 (1961)
[ESW15] Enea, C., Sighireanu, M., Wu, Z.: On automated lemma generation for separation logic with inductive definitions. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 80–96. Springer, Cham (2015). doi:10.1007/978-3-319-24953-7_7
[Fig12] Figueira, D.: Alternating register automata on finite words and trees. Log. Methods Comput. Sci. **8**(1), 1–43 (2012)

[FV98] Fülöp, Z., Vogler, H.: Syntax-Directed Semantics - Formal Models Based on Tree Transducers. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (1998)

[FV14] Fülöp, Z., Vogler, H.: Forward and backward application of symbolic tree transducers. Acta Inf. **51**(5), 297–325 (2014)

[Gal85] Gallier, J.H.: Logic for Computer Science: Foundations of Automatic Theorem Proving. Harper & Row Publishers, Inc., New York (1985)

[GCW16] Gu, X., Chen, T., Wu, Z.: A complete decision procedure for linearly compositional separation logic with data constraints. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 532–549. Springer, Cham (2016). doi:10.1007/978-3-319-40229-1_36

[GKS10] Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13089-2_47

[GKS12] Grumberg, O., Kupferman, O., Sheinvald, S.: Model checking systems and specifications with parameterized atomic propositions. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 122–136. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33386-6_11

[GKS13] Grumberg, O., Kupferman, O., Sheinvald, S.: An automata-theoretic approach to reasoning about parameterized systems and specifications. In: Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 397–411. Springer, Heidelberg (2013). doi:10.1007/978-3-319-02444-8_28

[GKS14] Grumberg, O., Kupferman, O., Sheinvald, S.: A game-theoretic approach to simulation of data-parameterized systems. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 348–363. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11936-6_25

[GS92] German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)

[HHR+12] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Form. Methods Syst. Des. **41**(1), 83–106 (2012)

[HLL+16] Hofman, P., Lasota, S., Lazić, R., Leroux, J., Schmitz, S., Totzke, P.: Coverability trees for petri nets with unordered data. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 445–461. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49630-5_26

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)

[IJS08] Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_19

[Kar16] Kara, A.: Logics on data words: expressivity, satisfiability, model checking. Ph.D. thesis, TU Dortmund University (2016). https://eldorado.tu-dortmund.de/bitstream/2003/35216/1/Dissertation.pdf

[KF94] Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994)

[KST12] Kara, A., Schwentick, T., Tan, T.: Feasible automata for two-variable logic with successor on data words. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 351–362. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28332-1_30

[KT06] Kaminski, M., Tan, T.: Regular expressions for languages over infinite alphabets. Fundam. Inform. **69**(3), 301–318 (2006)
[KT10] Kaminski, M., Tan, T.: A note on two-pebble automata over infinite alphabets. Fundam. Inform. **98**(4), 379–390 (2010)
[LSC16] Le, Q.L., Sun, J., Chin, W.-N.: Satisfiability modulo heap-based programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 382–404. Springer, Cham (2016). doi:10.1007/978-3-319-41528-4_21
[LTV15] Libkin, L., Tan, T., Vrgoc, D.: Regular expressions for data words. J. Comput. Syst. Sci. **81**(7), 1278–1297 (2015)
[MP71] McNaughton, R., Papert, S.: Counter-Free Automata. MIT Press, Cambridge (1971)
[MPQ11] Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 611–622 (2011)
[MR11a] Manuel, A., Ramanujam, R.: Class counting automata on datawords. Int. J. Found. Comput. Sci. **22**(4), 863–882 (2011)
[MR11b] Mens, I.-E., Rahonis, G.: Variable tree automata over infinite ranked alphabets. In: Winkler, F. (ed.) CAI 2011. LNCS, vol. 6742, pp. 247–260. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21493-6_16
[MRT14] Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Reachability in pushdown register automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) MFCS 2014. LNCS, vol. 8634, pp. 464–473. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44522-8_39
[MS04] Müller-Olm, M., Seidl, H.: A note on Karr's algorithm. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1016–1028. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27836-8_85
[NO79] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
[NSST15] Neven, F., Schweikardt, N., Servais, F., Tan, T.: Distributed streaming with finite memory. In: Proceedings of the 18th International Conference on Database Theory (ICDT), pp. 324–341 (2015)
[NSV01] Neven, F., Schwentick, T., Vianu, V.: Towards regular languages over infinite alphabets. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 560–572. Springer, Heidelberg (2001). doi:10.1007/3-540-44683-4_49
[NSV04] Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Log. **5**(3), 403–435 (2004)
[ORS+08] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1099–1110 (2008)
[PWZ13] Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_54
[PWZ14] Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_47

[Rey02] Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS), pp. 55–74 (2002)
[RISK16] Reynolds, A., Iosif, R., Serban, C., King, T.: A decision procedure for separation logic in SMT. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 244–261. Springer, Cham (2016). doi:10.1007/978-3-319-46520-3_16
[Sch65] Schützenberger, M.P.: On finite monoids having only trivial subgroups. Inf. Control **8**(2), 190–194 (1965)
[Seg06] Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006). doi:10.1007/11874683_3
[SG87] Sistla, A.P., German, S.M.: Reasoning with many processes. In: Proceedings of the 2nd Symposium on Logic in Computer Science (LICS), pp. 138–152 (1987)
[SRW02] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
[SW14] Song, F., Wu, Z.: Extending temporal logics with data variable quantifications. In: Proceedings of the 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS), pp. 253–265 (2014)
[SW16] Song, F., Zhilin, W.: On temporal logics with data variable quantifications: decidability and complexity. Inf. Comput. **251**, 104–139 (2016)
[Tan10] Tan, T.: On pebble automata for data languages with decidable emptiness problem. J. Comput. Syst. Sci. **76**(8), 778–791 (2010)
[Tan13] Tan, T.: Graph reachability and pebble automata over infinite alphabets. ACM Trans. Comput. Log. **14**(3), 19 (2013)
[Tan14] Tan, T.: Extending two-variable logic on data trees with order on data values and its automata. ACM Trans. Comput. Log. **15**(1), 8 (2014)
[TSJ+09] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive - a warehousing solution over a map-reduce framework. PVLDB **2**(2), 1626–1629 (2009)
[TW68] Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Theory Comput. Syst. **2**, 57–81 (1968)
[VB11a] Veanes, M., Bjørner, N.: Foundations of finite symbolic tree transducers. Bull. EATCS **105**, 141–173 (2011)
[VB11b] Veanes, M., Bjørner, N.: Symbolic tree transducers. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 377–393. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29709-0_32
[VB15] Veanes, M., Bjørner, N.: Symbolic tree automata. Inf. Process. Lett. **115**(3), 418–424 (2015)
[VB16] Veanes, M., Bjørner, N.: Equivalence of finite-valued symbolic finite transducers. In: Mazzara, M., Voronkov, A. (eds.) PSI 2015. LNCS, vol. 9609, pp. 276–290. Springer, Cham (2016). doi:10.1007/978-3-319-41579-6_21
[VBNB14] Veanes, M., Bjørner, N., Nachmanson, L., Bereg, S.: Monadic decomposition. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 628–645. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_42
[VD16] Veanes, M., D'Antoni, L.: Minimization of symbolic tree automata. In: Proceedings of the 30th IEEE Symposium on Logic in Computer Science (LICS) (2016)

[Vea13] Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 16–23. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39274-0_3

[VHL+12] Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: algorithms and applications. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 137–150 (2012)

[Via09] Vianu, V.: Automatic verification of database-driven systems: a new frontier. In: Proceedings of the 12th International Conference on Database Theory (ICDT), pp. 1–13 (2009)

[vL90] van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B. Elsevier and MIT Press, Amsterdam and Cambridge (1990)

[vNG01] van Noord, G., Gerdemann, D.: Finite state transducers with predicates and identities. Grammars **4**(3), 263–286 (2001)

[VW86] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the 1st Symposium on Logic in Computer Science (LICS), pp. 332–344 (1986)

[Wat96] Watson, B.W.: Implementing and using finite automata toolkits. Nat. Lang. Eng. **2**(4), 295–302 (1996)

[Wu11] Wu, Z.: A decidable extension of data automata. In: Proceedings of the 2nd International Symposium on Games, Automata, Logics and Formal Verification (GandALF), pp. 116–130 (2011)

[Wu12] Wu, Z.: Commutative data automata. In: Proceedings of the 26th International Workshop, 21st Annual Conference on Computer Science Logic (CSL), pp. 528–542 (2012)

[WVS83] Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS), pp. 185–194 (1983)

[Zar03] Zarba, C.G.: Combining sets with elements. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 762–782. Springer, Heidelberg (2003). doi:10.1007/978-3-540-39910-0_33