Chapter 1

Automata over infinite alphabets

Amaldev Manuel and R. Ramanujam
Institute of Mathematical Sciences, C.I.T campus,
Taramani, Chennai - 600113.

In many contexts such as validation of XML data, software model checking and parametrized verification, the systems studied are naturally abstracted as finite state automata, but whose input alphabet is infinite. While use of such automata for verification requires that the non-emptiness problem be decidable, ensuring this is non-trivial, since the space of configurations of such automata is infinite. We describe some recent attempts in this direction, and suggest that an entire theoretical framework awaits development.

1.1. Motivation

The theory of finite state automata over (finite) words is an arena that is rich in concepts and results, offering interesting connections between computability theory, algebra, logic and complexity theory. Moreover, finite state automata provide an excellent abstraction for many real world applications, such as string matching in lexical analysis [1, 2], model checking finite state systems [3] etc.

Considering that finite state machines have only bounded memory, it is *a priori* reasonable that their input alphabet is finite. If the input alphabet were infinite, it is hardly clear how such a machine can tell infinitely many elements apart. And yet, there are many good reasons to consider mechanisms that achieve precisely this.

Abstract considerations first: consider the set of all finite sequences of natural numbers (given in binary) separated by hashes. A word of this language, for example, is 100#11#1101#100#10101. Now consider the subset L containing all sequences with some number repeating in it. It is easily seen that L is not regular, it is not even context-free. The problem with L has little to do with the representation of the input sequence. If we were given a bound on the numbers occurring in any sequence, we could easily build a finite state automaton recognizing L. The difficulty arises precisely because we don't have such a bound or because we have 'unbounded data'. It is not difficult to find instances of languages like L occurring naturally in the computing world. For example consider the sequences of all *nonces* used in a security protocol run. Ideally this language should be $\overline{L}$. The question is how to recognize such languages, and whether there is any hope of describing regular collections of this sort.

Note that we could simply take the set of binary numbers as the alphabet in the example above: $D = \{\#, 0, 1, 10, 11, \ldots\})$. Now, $L = \{w = b_0\#b_1\# \ldots b_n \mid w \in D^*, \exists i, j.b_i = b_j\}$. Note further that D itself is a regular language over the alphabet $\{\#, 0, 1\}$.

There are more concrete considerations that lead to infinite alphabets as well, arising from two strands of computation theory: one from attempts to extend classical model checking techniques to *infinite state systems*, and the other is the realm of *databases*. Systems like software programs, protocols (communication, cryptography, ...), web services and alike are typically infinite state, with many different sources of unbounded data: program data, recursion, parameters, time, communication media, etc. Thus, model checking techniques are confronted with infinite alphabets. In databases, the XML standard format of *semi-structured data* consists of labelled trees whose nodes carry data values. The trees are constrained by schemes describing the tree structure, and restrictions on data values are specified through data constraints. Here again we have either trees or paths in trees whose nodes are labelled by elements of an infinite alphabet.

Building theoretical foundations for studies of such systems leads us to the question of how far we can extend finite state methods and techniques to infinite state systems. The attractiveness of finite state machines can mainly be attributed to the easiness of several decision problems on them. They are robust, in the sense of invariance under nondeterminism, alternation etc. and characterizations by a plurality of formalisms such as Kleene expressions, monadic second order logic, and finite semigroups. Regular languages are logically well behaved (closed under boolean operations, homomorphisms, projections, and so on). What we would like to do is to introduce mechanisms for unbounded data in finite state machines in such a way that we can retain as many of these nice properties as possible.

In the last decade, there have been several answers to this question. We make no attempt at presenting a comprehensive account of all these, but point to some interesting automata theory that has been developed in this direction. Again, while many theorems can be discussed, we concentrate only on one question, that of emptiness checking, guided by concerns of system verification referred to above.

The material discussed here covers the work of several researchers, and much of it can be found in [4–8]. Our own contribution is limited to the material in 1.5.

1.2. Languages of data words

Notation: Let $k > 0$; we use $[k]$ to denote the set $\{1, 2, \ldots k\}$. When we say $[k]_0$, we mean the set $\{0\} \cup [k]$. By $\mathbb{N}$ we mean the set of natural numbers $\{0, 1, \ldots\}$. When $f : A \to B$, $(a, b) \in (A \times B)$, by $f \oplus (a, b)$, we mean the function $f' : A \to B$, where $f'(a') = f(a')$ for all $a' \in A, a' \neq a$, and $f'(a) = b$.

Before we consider automaton mechanisms, we discuss languages over infinite alphabets. We will look only at languages of words but it is easily seen that similar notions can be defined for languages of *trees*, whose nodes are labelled from an infinite alphabet. We will use the terminology of database theory, and refer to languages over infinite alphabets as data languages. However, it should be noted that at least in the context of database theory, data trees (as in XML) are more natural than data words, but as it turns out, the questions discussed here turn out to be considerably harder for tree languages than for word languages.

Customarily, the infinite alphabet is split into two parts: it is of the form $\Sigma \times D$, where Σ is a finite set, and D is a countably infinite set. Usually, Σ is called the *letter alphabet* and D is called the *data alphabet.* Elements of D are referred to as *data values.* We use letters a, b etc to denote elements of Σ and use d, d' to denote elements of D.

The letter alphabet is a way to provide the data values 'contexts'. In the case of XML, Σ consists of tags, and D consists of data values. Consider the XML description: `<name> ''Tagore''</name>`: the tag `<name>` can occur along with different strings; so also, the string `''Tagore''` can occur as the value associated with different tags. As another example, consider a system of unbounded processes with states $\{b, w\}$ for 'busy' and 'wait'. When we work with the traces of such a system, each observation records the state of a process denoted by its process identifier (a number). A word in this case will be, for example, $(b, d_1)(w, d_2)(w, d_1)(b, d_2)$.

A **data word** w is an element of $(\Sigma \times D)^*$. A collection of data words $L \subseteq (\Sigma \times D)^*$ is called a *data language.* In this article, by default, we refer to data words simply as words and data languages as languages. As usual, by $|w|$ we denote the length of w.

Let $w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ be a data word. The *string projection* of w, denoted as $str(w) = a_1 a_2 \ldots a_n$, the projection of w to its Σ components. Let $i \in [n] = |w|$. The data class of d_i in w is the set $\{j \in [n] \mid d_i = d_j\}$. A subset of $[n]$ is called a data class of w if it is the data class of some d_i, $i \in [n]$. Note that the set of data classes of w form a partition of $[|w|]$.

We introduce some example data languages which we will keep referring to in the course of our discussion; these are over the alphabet $\Sigma = \{a, b\}, D = \mathbb{N}$.

- $L_{\exists n}$ is the set of all words in $(\Sigma \times D)^*$ in which at least n distinct data values occur.
- $L_{<n}$ is the set of all data words in which every data value occurs at most n times.
- $L_{a^*b^*}$ is the set of all data words whose string projections are in the set a^*b^*.
- $L_{\forall a \exists b}$ is the set of all data words where every data value occurring under a occurs under b also.
- $L_{fd(a)}$ is the collection of all data words in which all the data values in

context a are distinct. ($fd(a)$ stands for functional dependency on a.)

Let $\cdot$ denote concatenation on data words. For $L \subseteq (\Sigma \times D)^*$, consider the Myhill - Nerode equivalence on $(\Sigma \times D)^*$ induced by L: $w_1 \sim_L w_2$ iff $\forall w. w_1 \cdot w \in L \Leftrightarrow w_2 \cdot w \in L$. L is said to be regular when $\sim_L$ is of finite index. A classical theorem of automata theory equates the class of regular languages with those recognized by finite state automata, in the context of languages over finite alphabets.

It is easily seen that $\sim_{L_{fd(a)}}$ is not of finite index, since each singleton data word (a, d) is distinguished from (a, d'), for $d \neq d'$. Hence we cannot expect a classical finite state automaton to accept $L_{fd(a)}$; we need to look for another device, perhaps an infinite state machine.

Indeed, for most data languages, the associated equivalence relation is of infinite index. Is there a notion of *recognizability* that can be defined meaningfully over such languages and yet corresponds (in some way) to finite memory? This is the central question addressed in this article.

1.3. Formulating an automaton mechanism

The notion of regularity on languages over infinite alphabets can be approached using different mechanisms: *descriptive* ones like logics or rational expressions, or *operational* ones like automata models. There are two reasons for our discussing only automata here: one, machine models are closer to our algorithmic intuition about language behaviour, and enable us to compare the computational power of different machines; two, there are relatively fewer results to discuss on the descriptive side.

The first challenge in formulating an automaton mechanism is the question of 'finite representability'. It is essential for a machine model that the automaton is presented in a finite fashion. In particular, we need implicit finite representations of the data alphabet. An immediate implication is that we need algorithms that work with such implicit representations. Towards this, from now on, *we consider only data alphabets D in which membership and equality are decidable.*

Automata for words over finite alphabets are usually presented as working on a read-only finite tape, with a *tape head* under finite state control. One detail which is often taken for granted is the complexity of the tape head. Since we can recognize a finite language (which is the alphabet!) by a constant-sized circuit the computing power of the tapehead is far inferior to that of the automaton.

In the case of infinite alphabets, the situation is different, and our assumption about decidable membership and equality in D makes sense when we consider the complexity of the tape head. For example, if we consider the alphabet as the encodings of all halting Turing machines, the tapehead has to be a Σ^0_1 machine, which is obviously hard to conceive of as a machine model relevant to software verification. Therefore, we see that our assumption needs tightening and we should require the membership and equality checking in the alphabet to be *computationally feasible.* In fact, we should also ensure that the language accepted by the automaton, when restricted to a finite subset of the infinite alphabet, remains regular.

One obvious way of implementing finite presentations is by insisting that the automaton uses only *finitely many* data values in its transition relation. This seems reasonable, since any property of data that we wish to specify can refer explicitly only to finitely many data values. However, when the only allowed operation on data is checking for equality of data values, such an assumption becomes drastic: it is easily seen that having infinite data alphabets is superfluous in such automata. Every such machine is equivalent to a finite state machine over a finite alphabet.

Thus we note that infinite alphabets naturally lead us to infinite state systems, whose space of configurations is infinite. The theory of computation is rich in such models: pushdown systems, Petri nets, vector addition systems, Turing machines etc. In particular, we are led to models in which we equip the automaton with some additional mechanism to enable it to have infinitely many configurations.

This takes us to a striking idea from the 1960's: "automata theory is the study of memory structures". These are structures that allow us to fold infinitely many actions into finitely many instructions for manipulating memory, which can be part of the automaton definition. These are storage mechanisms which provide specific tools for manipulating and accessing data. Obvious memory mechanisms are *registers* (which act like scratch pads, for memorizing specific data values encountered), *stacks*, *queues* etc.

One obvious memory structure is the input tape, which can be 'upgraded' to an unbounded sequential read-write memory. But then it is easily noted that a finite state machine equipped with such a tape is Turing-complete. On the other hand, if the tape is read-only, the machine accepts all datawords whose string projections belong to the letter language (subset of Σ^*) defined by the underlying automaton. Clearly this machine is also not very interesting. We therefore look for structures that keep us in between: those with infinitely many configurations, but for which reachability is yet decidable. Note that such ambition is not unrealistic, since Petri nets and pushdown systems are systems of this kind.

1.4. Automata with registers

The simplest form of memory is a finite random access read-write storage device, traditionally called *register*. In register automata [4], the machine is equipped with finitely many registers, each of which can be used to store one data value. Every automaton transition includes access to the registers, reading them before the transition and writing to them after the transition. The new state after the transition depends on the current state, the input letter and whether or not the input data value is already stored in any of the registers. If the data value is not stored in any of the registers, the automaton can choose to write it in a register. The transition may also depend on which register contains the encountered data value.

Below, fix an alphabet $\Sigma \times D$, with Σ finite and D countable. Let $D_\perp = D \cup \{\perp\}$, where $\perp \notin D$ is a special symbol.

Definition 1.1. A ***k*-register finite memory automaton** is given by $R_A = (Q, \Sigma, \Delta, \tau_0, U, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. τ_0 is the initial register configuration given by $\tau_0 : [k] \to D_\perp$, and U is a partial *update* function: $(Q \times \Sigma) \to [k]$. The transition relation is $\Delta \subseteq (Q \times \Sigma \times [k] \times Q)$. We assume that for all $i, j \in [k]$ if $i \neq j$ and $\tau_0(i), \tau_0(j) \in D$ then $\tau_0(i) \neq \tau_0(j)$ (that is, registers initially contain distinct data values).

Note that the description of R_A is finite, D appears only in the specification of the initial register configuration. It is reasonable to suppose that bounded precision suffices for this purpose. Thus the infinite alphabet plays only an implicit (albeit critical) role in the behaviour of R_A.

$\perp$ is used above to denote an uninitialized register. The working of the automaton is as follows. Suppose that R_A is in state p, with each of the registers r_i holding data value d_i, $i \in [k]$, and its input is of the form (a, d) where $a \in \Sigma$ and $d \in D$. Now there are two cases:

- If $d \neq d_i$ for all i, then a register update is enabled. If $U(p, a)$ is undefined, this is deemed to be an error, and the machine halts. Otherwise the value d is put into r_j where $U(p, a) = j$; other registers are left untouched, and the state does not change.
- Suppose that $d = d_i$, for some $i \in [k]$, and $(p, a, i, q) \in \Delta$. Then this transition is enabled, and when applying the transition, the registers are left untouched.

This is formalized as follows. A *configuration* of R_A is of the form (q, τ) where $q \in Q$ and $\tau : [k] \to D_\perp$. Let C_A denote the set of all configurations of R_A. Δ and U define a transition $\Delta(C_A) \subseteq (C_A \times \Sigma \times D \times C_A)$ as follows: there is a transition from (q, τ) to (q', τ') in $\Delta(C_A)$ on (a, d) iff (a) $\tau = \tau'$, $d = \tau(i)$, $(i \in [k])$, $(q, a, i, q') \in \Delta$, or (b) for all $i \in [k]$, $d \neq \tau(i)$, $\tau' = \tau \oplus (U(q, a), d)$, $(q, a, U(q, a), q') \in \Delta$.

A *run* of R_A on a data word $w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ is a sequence $\gamma = (q_0, \tau_0)(q_1, \tau_1) \ldots (q_n, \tau_n)$, where (q_0, τ_0) is the initial configuration of R_A, and for every $i \in [n]$, there is a transition from (q_{i-1}, τ_{i-1}) to (q_i, τ_i) on (a_i, d_i) in $\Delta(C_A)$. γ is *accepting* if $q_n \in F$. The language accepted by R_A, denoted $L(R_A) = \{w \in (\Sigma \times D)^* \mid R_A$ has an accepting run on $w\}$.

Example 1.1. Recall the language $L_{fd(a)}$ mentioned earlier: it is the set of all data words in which all the data values in context a are distinct. Formally:

$$L_{fd(a)} = \left\{ w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n) \;\middle|\; \begin{array}{c} w \in (\Sigma \times D)^*, \\ \forall ij.a_i = a_j = a \implies d_i \neq d_j \end{array} \right\}$$

The language $\overline{L_{fd(a)}}$, can be accepted by a 2-register finite memory automaton $A = (Q, q_0, \Delta, \tau_0, U, F)$, where

- $Q = \{q_0, q_1, q_f\}$
- $\tau_0 = (\perp, \perp)$
- $U(q_0, \Sigma) = 1, U(q_1, \Sigma) = U(q_f, \Sigma) = 2$[*]
- $F = \{q_f\}$
- Δ consists of,
  - $(q_0, \Sigma, 1, q_0)$
  - $(q_0, a, 1, q_1)$
  - $(q_1, \Sigma, 2, q_1)$
  - $(q_1, a, 1, q_f)$
  - $(q_f, \Sigma, \{1, 2\}, q_f)$

A works as follows. Initially A is in state q_0 and stores new input data in the first register. When reading the data value with label a, which appears twice, A changes the state to q_1 and waits there storing the new data in the second register. When the data value stored in the first register appears the second time with label a, A changes state to q_f and continues to be there.

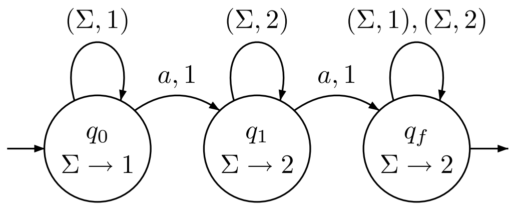

Fig. 1.1. Automaton in the Example 1.1

Note that a finite memory automaton uses only finitely many registers to deal with infinitely many symbols, and hence we get something analogous to the pumping lemma for regular languages which asserts that a finite state automaton which accepts sufficiently long words also accepts infinitely many words. Suppose there are k registers and the automaton sees $k+1$ data values; since the only places where it can store these data values are in the registers, it is bound to forget one of the data values. This is made precise by the following lemma.

Lemma 1.1. *If a k-register automaton A accepts any word at all, then it accepts a word containing at most k distinct data values.*

Proof. Let $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ be a data word accepted by A and $(q_0, \tau_0)(q_1, \tau_1) \dots (q_n, \tau_n)$ be an accepting run of A on w. Let $U_{min}(w) = j$ be the least integer such that $d_j \notin range(\tau_{j-1})$ and $\tau_{j-1}(U(q_{j-1}, a_j)) \neq \perp$ (∞, if it is not defined). The previous condition says that j is the first position in the run, where A is going to replace a data value in the register which is not $\perp$. If such a j does

[*]We abuse the notation here, $U(q_0, \Sigma) = 1$, denotes that for all $a \in \Sigma$, $U(q_0, a) = 1$. We use similar shorthands later on whenever they are handy and intuitive, for instance in writing the transition relation of an automaton.

not exist, then the register values are never replaced (other than $\perp$'s) and therefore w contains at most k distinct data values. If $j \leq n$, then we will construct a data word w' of length n accepted by A, with $j < U_{min}(w')$. This implies that by finite iteration (at most n times) of the process described below, we can construct a word with at most k distinct symbols. Let $u = U(q_{j-1}, a_j)$ and let $d_u = \tau_{j-1}(u)$.

Consider

$$w' = \begin{matrix} a_1\, a_2 \ldots a_{j-1} \\ d_1\, d_2 \ldots d_{j-1} \end{matrix} \left\{ \begin{matrix} a_j\, a_{j+1} \ldots a_n \\ d_j\, d_{j+1} \ldots d_n \end{matrix} \right\} [d_j \mid d_u, d_u \mid d_j]$$

which is got by replacing all the occurrences of d_j with d_u and replacing d_u by d_j in the string, after the position j. It is clear that, when the automaton reaches the position j, the data value is present in the register $U(u)$, and therefore no replacement will be done. Hence $j < U_{min}(w')$.

However, in order to show that w' is accepted by A, it remains to be proved that $(q_0, \tau_0)(q_1, \tau_1) \ldots (q'_j, \tau'_j)(q'_{j+1}, \tau'_{j+1}) \ldots (q'_n, \tau'_n)$ is an accepting run for w', where, $q'_i = q_i$ and $\tau'_i = \tau_i [d_j \mid \tau_{j-1}(u), \tau_{j-1}(u) \mid d_j]$. We prove this using induction on the length of the modified part of the run. Since in the original run d_j is stored in the register u and $(u, q_{j-1}, a_j, q_j) \in \Delta$ it follows that there is a transition from (q_{j-1}, τ_{j-1}) to (q'_j, τ'_j) on $(a_j, \tau_{j-1}(u))$. Now for the inductive step, assume that $(q_0, \tau_0)(q_1, \tau_1) \ldots (q'_h, \tau'_h)$ is the modified run corresponding to $(q_0, \tau_0)(q_1, \tau_1) \ldots (q_h, \tau_h)$. We have to prove that there is a transition from (q'_h, τ'_h) to (q'_{h+1}, τ'_{h+1}) on (a_h, d'_h), where d'_h stands for $d_h\, [d_j \mid \tau_{j-1}(u), \tau_{j-1}(u) \mid d_j]$.

- If for some $t \in [k]$, $d_h = \tau_h(t)$ then $(t, q_h, a_h, q_{h+1}) \in \Delta$ and $\tau_{h+1} = \tau_h$. Therefore d'_h appears in the register t. Hence there is a transition from (q'_h, τ'_h) to (q'_{h+1}, τ'_{h+1}) on (a_h, d'_h) also.
- If $d_h \notin \mathit{range}(\tau_h)$, then τ_{h+1} is obtained from τ_h by replacing the contents of the register $U(q_h, a_h)$ with d_h. Since $d'_h \notin \mathit{range}(\tau'_h)$, τ'_{h+1} is obtained from τ'_h by replacing the contents of register $U(q_h, a_h)$. Again there is a transition from (q'_h, τ'_h) to (q'_{h+1}, τ'_{h+1}) on (a_h, d'_h).

This completes the proof. □

Note that the language $L_{fd(a)}$ requires unboundedly many data values to occur with a, and hence by the above lemma, it cannot be recognized by any register automaton. On the other hand, since $\overline{L_{fd(a)}}$ can accepted by a register automaton, we see that languages recognized by register automata are not closed under complementation. As this suggests, non-deterministic register automata are more powerful than deterministic ones.

While the lemma demonstrates a limitation of register machines in terms of computational power, it also shows the way for algorithms on these machines.

Theorem 1.1. *Emptiness checking of register automata is decidable.*

Proof. Let A be a register automaton with k registers, which we want to check for emptiness. Let $range(\tau_0) \subseteq D' \subseteq D, |D'| = k$ be a subset of D containing k different symbols including those in the register's initialization. We claim that $L(A) \neq \emptyset$ if and only if $L(A) \cap (\Sigma \times D')^* \neq \emptyset$. The if direction is trivial. The other direction follows from the preceding lemma. Thus a classical finite state automaton working on a finite alphabet can be employed for checking emptiness of A. □

The emptiness problem for register automata is in NP, since we can guess a word of length polynomial in the size of the automaton and verify that it is accepted. It has also been shown that the problem is complete for NP in [9]. The problem is no less hard for the deterministic subclass of these automata. Though, as we mentioned earlier, register automata are not closed under complementation, they are closed under intersection, union, Kleene iteration and homomorphisms.

There are many extensions of the register automaton model. An obvious one is to consider *two-way* machines: interestingly, this adds considerable computational power and the emptiness problem becomes undecidable [4, 5, 10].

1.5. Automata with counters

Automata with counters have a long history. It is well known that a finite state automaton equipped with two counters is Turing-complete, whereas with only one counter, it has the computational power of a pushdown automaton. Can the counter mechanism be employed in the context of unbounded data? While a register mechanism is used to note down data values, counters are used to record the number of occurrences of some (pre-determined) events.

When automata with counters are considered on words over finite alphabets, one typical use of counters is to note the multiplicity of a letter in the input word. On infinite alphabets, we could similarly count the number of occurrences of data values, or letter - value pairs, or these subject to constraints. But then, each such 'event type' needs a counter for itself, which implies the need for unboundedly many counters. On the other hand, as we observed above, two counter machines are already Turing-complete. This suggests restraint on the allowed counter operations. While there are many possible restrictions, a natural one is to consider *monotone* counters, which can be incremented, reset and compared against constants, but never decremented.

A model with such characteristics is presented below. The automaton includes a bag of infinitely many monotone counters, one for each possible data value. When it encounters a letter - data pair, say (a, d), the multiplicity of d is checked against a given constraint, and accordingly updated, the transition causing a change of state, as well as possible updates for other data as well. We can think of the bag as a hash table, with elements of D as keys, and counters as hash values. Transitions depend only on hash values (subject to constraints) and not keys.

Such counting is used in many infinite state systems. For instance, in systems of unbounded processes, a task enters the system with low priority which increases in the course of a computation until the task is scheduled, resetting the priority. In the context of web services, a server not only offers data dependent services, but also needs information on how many clients are at any time using a particular service. For instance, the loan requests granted by a server depend on how many clients are requesting low credit and how many need high credit at that time.

A **constraint** is a pair $c = (op, e)$, where $op \in \{<, =, \neq, >\}$ and $e \in \mathbb{N}$. When $v \in \mathbb{N}$, we say $v \models c$ if $v \, \mathsf{op} \, e$ holds. Let $\mathbb{C}$ denote the set of all constraints. Define a *bag* to be a map $h : \mathbb{D} \to \mathbb{N}$. Let $\mathbb{B}$ denote the set of bags.

Below, let $Inst = \{\uparrow^+, \downarrow\}$ stand for the set of *instructions*: $\uparrow^+$ tells the automaton to increment the counter, whereas $\downarrow$ asks for a reset.

Definition 1.2. A **class counting automaton**, abbreviated as CC_A, is a tuple $CC_A = (Q, \Delta, I, F)$, where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states. The transition relation is given by: $\Delta \subseteq (Q \times \Sigma \times C \times Inst \times U \times Q)$, where C is a finite subset of $\mathbb{C}$ and U is a finite subset of $\mathbb{N}$.

Let A be a CC_A. A configuration of A is a pair (q, h), where $q \in Q$ and $h \in \mathbb{B}$. The initial configuration of A is given by (q_0, h_0), where $\forall d \in \mathbb{D}, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \dots (a_n, d_n)$, a run of A on w is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ such that $q_0 \in I$ and for all $i, 0 \leq i < n$, there exists a transition $t_i = (q, a, c, \pi, n, q') \in \Delta$ such that $q = q_i$, $q' = q_{i+1}$, $a = a_{i+1}$ and:

- $h_i(d_{i+1}) \models c$.
- h_{i+1} is given by:

$$h_{i+1} = \begin{cases} h_i \oplus (d, n') & \text{if } \pi = \uparrow^+, n' = h_i(d) + n \\ h_i \oplus (d, n) & \text{if } \pi = \downarrow \end{cases}$$

γ is an accepting run above if $q_n \in F$. The language accepted by A is given by $\mathcal{L}(A) = \{w \in \Sigma \times \mathbb{D}^* \mid A \text{ has an accepting run on } w\}$. $\mathcal{L} \subseteq (\Sigma \times \mathbb{D})^*$ is said to be recognizable if there exists a CC_A A such that $\mathcal{L} = \mathcal{L}(A)$. Note that the counters are either incremented or reset to fixed values.

Note that the instruction $(\uparrow^+, 0)$ says that we do not wish to make any update, and $(\uparrow^+, 1)$ causes a unit increment; we use the notation [0] and [+1] for these instructions below.

Example 1.2. The language $L_{fd(a)}$ = *"Data values under a are all distinct"* is accepted by a CC_A. The CC_A accepting this language is the automaton $A = (Q, \Delta, q_0, F)$ where

$Q = \{q_0, q_1\}$, q_0 is the only initial state and $F = \{q_0\}$. Δ consists of:

- $(q_0, a, (=, 0), q_0, [+1])$

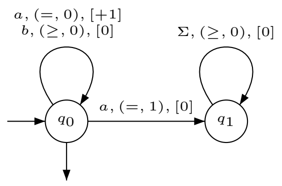

Fig. 1.2. Automaton in the Example 1.2

- $(q_0, a, (=, 1), q_1, [0])$
- $(q_0, b, (\geq, 0), q_0, [0])$
- $(q_1, \Sigma, (\geq, 0), q_1, [0])$

Since the automaton above is deterministic, by complementing it, that is, setting $F = \{q_1\}$, we can accept the language $\overline{L_{fd(a)}}$ = *"There exists a data value appearing at least twice under a"*.

It is easily seen that by a similar argument, the language $L_{\forall a, \leq\ n}$ = *"All data values under a occur at most n times"* can be accepted by a CC_A. So also is the language $L_{\exists a, \geq\ n}$ = *" There exists a data value appearing under a occurring more than n times"*.

Given a CC_A $A = (Q, \Delta, q_0, F)$ let m be the maximum constant used in Δ. We define the following equivalence relation on $\mathbb{N}$, $c \simeq_{m+1} c'$ iff $c < (m+1) \vee c' < (m+1) \Rightarrow c = c'$. Note that if $c \simeq_{m+1} c'$ then a transition is enabled at c if and only if it is enabled at c'. We can extend this equivalence to configurations of the CC_A as follows. Let $(q_1, h_1) \simeq_{m+1} (q_2, h_2)$ iff $q_1 = q_2$ and $\forall d \in \mathbb{D}, h_1(d) \simeq_{m+1} h_2(d)$.

Lemma 1.2. *If C_1, C_2 are two configurations of the CC_A such that $C_1 \simeq_{m+1} C_2$, then $\forall w \in (\Sigma \times \mathbb{D})^*$, $C_1 \vdash_w^* C_1' \implies \exists C_2', C_2 \vdash_w^* C_2'$ and $C_1' \simeq_{m+1} C_2'$.*

Proof. Proof by induction on the length of w. For the base case observe that any transition enabled at C_1 is enabled at C_2 and the counter updates respects the equivalence. For the inductive case consider the word $w.a$. By induction hypothesis $C_1 \vdash_w^* C_1' \implies \exists C_2', C_2 \vdash_w^* C_2'$ and $C_1' \simeq_{m+1} C_2'$. If $C_1' \vdash_a C_1''$ then using the above argument there exists C_2'' such that $C_2' \vdash_a C_2''$ and $C_1'' \simeq_{m+1} C_2''$. □

In fact the lemma holds for any $N \geq m+1$, where m is the maximum constant used in Δ. This observation paves the way for the route to decidability of the emptiness problem.

Before we proceed to discuss decidability, we observe that the model admits many extensions. For instance, instead of working with one bag of counters, the automaton can use several bags of counters, much as multiple registers are used in the register automaton. Another strengthening involves checking for the presence of *any* counter satisfying a given constraint and updating it. Moreover, the language of constraints can be strengthened: any syntax that can specify a finite or co-finite subset of $\mathbb{N}$ will do. We do not pursue these generalizations here, but merely remark that the theory extends straightforwardly.

Theorem 1.2. *The emptiness problem of class counting automata is decidable.*

Proof. We reduce the emptiness problem of CC_A to the covering problem on Petri nets. Recall that in the case of nondeterministic finite state automata over finite alphabets, for checking emptiness, we can omit the labels on transitions and reduce the problem to reachability in graphs. In the context of CC_A, the similar simplification is to omit $\Sigma \times D$ labels from the configuration graph; we are then left with counter behaviour. However reachability is no longer trivial, since we have unboundedly many counters, leading us to the realm of vector addition systems.

Definition 1.3. An ω-counter machine B is a tuple (Q, Δ, q_0) where Q is a finite set of states, $q_0 \in Q$ is the initial state and $\Delta \subseteq (Q \times C \times Inst \times U \times Q)$, where C is a finite subset of $\mathbb{C}$ and U is a finite subset of $\mathbb{N}$.

A configuration of B is a pair (q, h), where $q \in Q$ and $h : \mathbb{N} \to \mathbb{N}$. The initial configuration of B is (q_0, h_0) where $h_0(i) = 0$ for all i in $\mathbb{N}$. A run of B is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ such that for all i such that $0 \leq i < n$, there exists a transition $t_i = (p, c, \pi, n, q) \in \Delta$ such that $p = q_i$, $q = q_{i+1}$ and there exists j such that $h(j) \models c$, and the counters are updated in a similar fashion to that of CC_A.

The reachability problem for B asks, given $q \in Q$, whether there exists a run of B from (q_0, h_0) ending in (q, h) for some h ("Can B reach q?").

Lemma 1.3. *Checking emptiness for CC_A can be reduced to checking reachability for ω-counter machines.*

Proof. It suffices to show, given a CC_A, $A = (Q, \Delta, q_0, F)$, where $F = \{q\}$, that there exists a counter machine $B_A = (Q, \Delta', q_0)$ such that A has an accepting run on some data word exactly when B_A can reach q. (When F is not a singleton, we simply repeat the construction.) Δ' is obtained from Δ by converting every transition (p, a, c, π, n, q) to (p, c, π, n, q). Now, let $L(A) \neq \emptyset$. Then there exists a data word w and an accepting run $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ of A on w, with $q_n = q$. Let $g : \mathbb{N} \to D$ be an enumeration of data values. It is easy to see that $\gamma' = (q_1, h_0 \circ g)(q_1, h_1 \circ g) \dots (q_n, h_n \circ g)$ is a run of B_A reaching q.

($\Leftarrow$) Suppose that B_A has a run $\eta = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$, $q_n = q$. It can be seen that $\eta' = (q_0, h_0 \circ g^{-1})(q_1, h_1 \circ g^{-1}) \dots (q_n, h_n \circ g^{-1})$ is an accepting run of A on $w = (a_1, d_1) \dots (a_n, d_n)$ where w satisfies the following. Let (p, c, π, n, q) be the transition of B_A taken in the configuration (q_i, h_i) on counter m. Then by the definition of B_A there exists a transition (p, a, c, π, n, q) in Δ. Then it should be the case that $a_{i+1} = a$ and $d_{i+1} = g(m)$. □

Proposition 1.1. *Checking non-emptiness of ω-counter machines is decidable.*

Let $s \subseteq \mathbb{N}$, and c a constraint. We say $s \models c$, if for all $n \in s$, $n \models c$.

We define the following partial function Bnd on all finite and cofinite subsets of $\mathbb{N}$. Given $s \subseteq_{fin} \mathbb{N}$, $Bnd(s)$ is defined to be the least number greater than all the elements in s. Given $s \subseteq_{cofinite} \mathbb{N}$, $Bnd(s)$ is defined to be $Bnd(\mathbb{N}\backslash s)$. Given an ω-counter machine $B = (Q, \Delta, q_0)$ let $m_B = max\{Bnd(s) \mid s \models c, c \text{ is used in } \Delta\}$.

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = Q \cup \{i \mid i \in \mathbb{N}, 1 \leq i \leq m_B\}$.
- T is defined according to Δ as follows. Let $(p, c, \pi, n, q) \in \Delta$ and let i be such that $0 \leq i \leq m_B$ and $i \models c$. Then we add a transition t such that ${}^\bullet t = \{p, i\}$ and $t^\bullet = \{q, i'\}$, where (i) if π is $\uparrow^+$ then $i' = min\{m_B, i + n\}$, and (ii) if π is $\downarrow$ then $i' = n$.
- The flow relation F is defined according to ${}^\bullet t$ and $t^\bullet$ for each $t \in T$.
- The initial marking is defined as follows. $M_0(q_0) = 1$ and for all p in S, if $p \neq q_0$ then $M_0(p) = 0$.

The construction above glosses over some detail: Note that elements of these sets can be zero, in which case we add edges only for the places in $[m_B]$ and ignore the elements which are zero.

Let M be any marking of N_B. We say that M is a *state marking* if there exists $q \in Q$ such that $M(q) = 1$ and $\forall p \in Q$ such that $p \neq q$, $M(p) = 0$. When M is a state marking, and $M(q) = 1$, we speak of q as the state marked by M. For $q \in Q$, define $M_f(q)$ to be set of state markings that mark q. It can be shown, from the construction of N_B, that in any reachable marking M of N_B, if there exists $q \in Q$ such that $M(q) > 0$, then M is a state marking, and q is the state marked by M.

We now show that the counter machine B can reach a state q iff N_B has a reachable marking which covers a marking in $M_f(q)$. We define the following equivalence relation on $\mathbb{N}$, $m \simeq_{m_B} n$ iff $(m < m_B) \vee (n < m_B) \Rightarrow m = n$. We can lift this to the hash functions (in ω-counters) in the natural way: $h \simeq_{m_B} h'$ iff $\forall i\, (h(i) < m_B) \vee (h'(i) < m_B) \Rightarrow h(i) = h'(i)$. It can be easily shown that if $h \simeq_{m_B} h'$ then a transition is enabled at h if and only if it is enabled at h'.

Let μ be a mapping B-configurations to N_B-configurations as follows: given $\chi = (q, h)$, define $\mu(\chi) = M_\chi$, where

$$M_\chi(p) = \left\{ \begin{array}{ll} 1 & \text{iff } p = q \\ 0 & \text{iff } p \in Q\backslash\{q\} \\ |[p]| & \text{iff } p \in P\backslash Q, p \neq 0 \end{array} \right\}$$

Above $[p]$ denotes the equivalence class of p under $\simeq_{m_B}$ on $\mathbb{N}$ in h. Now suppose that B reaches q. Let the resulting configuration be $\chi = (q, h)$. We claim that the marking $\mu(\chi)$ of N_B is reachable (from M_0) and covers $M_f(q)$. Conversely if a reachable marking M of N_B covers $M_f(q)$, for some $q \in Q$, then there exists a reachable configuration $\chi = (q, h)$ of B such that $\mu(\chi) = M$. This is proved by a simple induction on the length of the run.

Since the covering problem for Petri nets is decidable, so is reachability for ω-counter machines and hence emptiness checking for CC_A is decidable. $\square$

The decision procedure above runs in EXPSPACE, and thus we have elementary decidability, though CC_A configurations form an infinite state system. The problem is complete for EXPSPACE by an easy reduction from covering problem of Petri nets. CC_A are not closed under complementation, but they are closed under union and intersection. The details can be found in [11].

1.6. Automata with hash tables

A hash table is a data structure containing 'keys' and 'values'. It provides random access to the stored 'value' corresponding to 'key'. In the case of infinite data, we can employ a hash table with the elements of D as the keys. The values have to be from a finite set, since a finite state automaton can only distinguish only finitely many symbols in the transition relation. Thus, the hash values impose an equivalence relation of finite index on data.

The main idea is as follows. On reading a (a, d), the automaton reads the table entry corresponding to d and makes a transition dependent on the table entry, the input letter a and the current state. The transition causes a change of state as well as updating of the table entry. Such a model has been termed a class memory automaton [6].

Definition 1.4. A **class memory automaton** is a tuple $CM_A = (Q, \Sigma, \Delta, q_0, F_\ell, F_g)$ where Q is a finite set of states, q_0 is the initial state and $F_g \subseteq F_\ell \subseteq Q$ are the sets of **global** and **local** accepting states respectively. The transition relation is $\Delta \subseteq (Q \times \Sigma \times (Q \cup \{\bot\}) \times Q)$.

The working of the automaton is as follows. The finite set of hash values is simply the set of automaton states. A transition of the form (p, a, s, q) on input (a, d) stands for the state transition of the automaton from p to q as well as the updating of the hash value for d from s to q. The acceptance condition has two parts. The global acceptance set F_g is as usual: after reading the input the automaton state should be in F_g. The local acceptance condition refers to the state of the hash table: the image of the hash function should be contained in F_ℓ. Thus acceptance depends on the memory of the data encountered.

Formally, a hash function is a map $h : D \to (Q \cup \{\bot\})$ such that $h(d) = \bot$ for all but finitely many data values. h holds the hash value (the state) which is assigned to the data value d when it was read the last time. A configuration of the automaton is of the form (q, h) where h is a hash function. The initial configuration of the automaton is (q_0, h_0) where $h_0(d) = \bot$ for all $d \in D$.

Transition on configurations is defined as follows: a transition from a configuration (p, h) on input (a, d) to (q, h') is enabled if $(p, a, h(d), q) \in \Delta$ $h' = h \oplus (d, q)$.

A *run* of CM_A on a data word $w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ is, as usual, a sequence $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$, where h_0 is the initial configuration of CM_A, and for every $i \in [n]$, there is a transition from (q_{i-1}, h_{i-1}) to (q_i, h_i) on

(a_i, d_i) in $\Delta(C_A)$. γ is *accepting* if $q_n \in F_g$ and for all $d \in D$, $f(d) \in F_l \cup \{\perp\}$. The language accepted by CM_A, denoted $L(CM_A) = \{w \in (\Sigma \times D)^* \mid CM_A$ has an accepting run on $w\}$.

Example 1.3. The language $L_{fd(a)}$ can be accepted by the following class memory automaton $A = (Q, \Sigma, \Delta, q_0, F_l, F_g)$ where $Q = \{q_0, q_a, q_b\}$ and Δ contains the tuples $\{(p, a, \perp, q_a), (p, b, \perp, q_b), (p, b, q_a, q_a), (p, b, q_b, q_b), (p, a, q_b, q_a) \mid p \in \{q_0, q_a, q_b\}\}$. F_l is the set $\{q_b\}$ and F_g is the set Q.

Theorem 1.3. *The emptiness problem for CM_A is decidable.*

Proof. Let $A = (Q, \Sigma, \Delta, q_0, F_l, F_g)$ be a given CM_A. We construct a Petri net N_A and a set of configurations M_A such that A accepts a string if and only if N_A can reach any of M_A.

Define $N_A = (S, T, F)$ where $S = Q \cup \{q^c \mid q \in Q\}$, and the transition relation T is as follows. For each $\delta = (p, a, s, q)$ where $s \neq \perp$ we add a new transition t_δ such that ${}^\bullet t_\delta = \{p, s^c\}$ and $t_\delta^\bullet = \{q, q^c\}$. For each $\delta = (p, a, \perp, q)$ where we add a new transition t_δ such that ${}^\bullet t_\delta = \{p\}$ and $t_\delta^\bullet = \{q, q^c\}$. We add additional transitions $t_{(p,q)}$ for each $p \in F_g, q \in F_l$ such that ${}^\bullet t_{(p,q)} = \{p, q^c\}$ and $t_{(p,q)}^\bullet = \{p\}$. The flow relation is defined accordingly.

The initial marking of the net is M_0 where q_0 has a single token and all other places are empty. M_A is the set of configurations in which exactly one of $q \in F_g$ has a single token and all other places are empty.

The details are routine. The place q^c keeps track of the number of data values with state q. Using induction it can be easily shown that a run of the automata gives a firing sequence in the net and vice versa. Finally when we reach a global state we can use the additional transitions to pump out all the tokens in the local final states. The only subtlety is that the additional transitions in the net can be used even before reaching an accepting configuration in the net, in which case it amounts to abandoning certain data classes in the run of the automaton (these are data values which are not going to be used again). $\square$

Thus emptiness for CM_A is reduced to reachability in Petri nets. As it happens, it is also as hard as Petri net reachability [7]. Since the latter problem is not even known to be elementary, we need to look for subclasses with better complexity. CM_A are not closed under complementation, but they are closed under union, intersection, homomorphisms. It also happens that they admit a natural logical characterization to which we will return later.

1.7. Automata with stacks

Another memory structure that has played a significant role in theory of computation is the stack, or pushdown mechanism. In automata theory, use of the pushdown mechanism is related to acceptance of context free languages. The main idea is that we can remember unbounded information but can access the memory only in a limited fashion. This gives the same power as a finite state machine with a single counter which can be incremented, decremented and checked for zero. In the context of data words, such a mechanism can be employed either to remember data values, or positions in the word, (denoting data classes), or both.

An elegant way of modelling such memory of positions is the concept of a pebble: whenever we wish to remember a position (perhaps one where we say a particular data value), we place a pebble on it. How many pebbles we can use, and when several pebbles have been placed, which one can be accessed first, determines the memory structure. Below, we consider a model where a stack discipline is used to access the pebbles.

1.7.1. *Pebble automata*

Below, let $Ins = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. These are instructions to the automaton: while $\leftarrow$ and $\rightarrow$ tell the machine to move right or left along the data word, $\uparrow$ and $\downarrow$ are for pushing on to or popping up the stack.

Definition 1.5. A pebble automaton [5] P_A is a system $(Q, \Sigma, k, \Delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states; $k > 0$ is called the *stack height*, and $\Delta : (Q \times \Sigma \times [k] \times 2^{[k]} \times 2^{[k]}) \rightarrow (Q \times Ins)$.

When a transition is of the form $\alpha \rightarrow \beta$ where $\alpha = (p, a, i, P, V)$ and $\beta = (q, \pi)$ where $\pi \in Ins$, the automaton is in state p, reading letter a, and depending on the "pointer stack" (i, P, V) transits to state q, moving the control head or operating on the stack according to Ins. The stack information, which will be clearer as we study configurations, is as follows: i is the current height of the stack, P is used to collect the pointers at a position, whereas V collects pointers having the same data value.

Let $m \geq 0$. The set of *m-configurations* of P_A is given by $C_m = (Q \times [k] \times ([k] \rightarrow [m]_0))$. Given a string $w \in (\Sigma \times D)^*$, we call $\chi \in C_{|w|}$ a w-configuration. $\chi_0 = (q, i, \theta) \in C_0$ is said to be *initial* if $q = q_0$, $i = 1$ and $\theta(1) = 1$. A configuration (q, i, θ) is said to be *final* if $q \in F$ and $\theta(i) = |w|$.

Consider a data word $(a_1, d_1) \dots (a_m, d_m)$. A transition $(p, i, a, P, V) \rightarrow (q, \pi)$ *is enabled* at a w-configuration (q, j, θ) if $p = q$, $i = j$, $a_{\theta(i)} = a$, $P = \{\ell < i \mid \theta(\ell) = \theta(i)\}$, and $V = \{\ell < i \mid d_{\theta(\ell)} = d_{\theta(i)}\}$. Thus P is the set of all pointers pointing to the position pointed by $\theta(i)$, and V is the set of all positions which contains the same data value pointed by $\theta(i)$.

P_A can go from a w-configuration (p, i, θ) to a w-configuration (q, i', θ') if there is a transition $\alpha \rightarrow (q, \pi)$ enabled at (p, i, θ) such that $\theta'(j) = \theta(i)$ for all $j < i$ and:

(1) if $\pi = \rightarrow$ then $i = i'$ and $\theta'(i) = \theta(i) + 1$.
(2) if $\pi = \leftarrow$ then $\theta(i) > 1$, $i = i'$ and $\theta'(i) = \theta(i) - 1$.
(3) if $\pi = \downarrow$, then $i < k$, $i' = i + 1$ and $\theta'(i') = \theta(i)$.

(4) if $\pi = \uparrow$, then $i > 1$, and $i' = i - 1$.

Above, whenever the side conditions are not met (as for instance, when it tries to pop the empty stack), the machine halts. As one may expect, the pebble mechanism adds considerable computational power.

Example 1.4. The language $\overline{L_{fd(a)}}$, can be accepted by a pebble automaton $A = (Q, \Sigma, 2, q_0, F, \Delta)$, where $Q = \{q_0, q_1, q_\rightarrow, q_f\}$ and $F = \{q_f\}$. Δ consists of the following transitions: $\Delta = \{(q_0, \Sigma, 1, \emptyset, \emptyset) \rightarrow (q_0, \rightarrow), (q_0, a, 1, \emptyset, \emptyset) \rightarrow (q_\rightarrow, \downarrow), (q_\rightarrow, a, 2, \{1\}, \{1\}) \rightarrow (q_1, \rightarrow), (q_1, \Sigma, 2, \emptyset, \emptyset) \rightarrow (q_1, \rightarrow), (q_1, a, 2, \emptyset, \{1\}) \rightarrow (q_f, \rightarrow), (q_f, \Sigma, 2, *, *) \rightarrow (q_f, \rightarrow)\}$

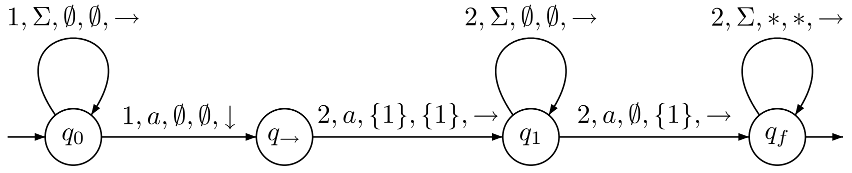
$1, \Sigma, \emptyset, \emptyset, \rightarrow$
$2, \Sigma, \emptyset, \emptyset, \rightarrow$
$2, \Sigma, *, *, \rightarrow$
q_0
$1, a, \emptyset, \emptyset, \downarrow$
$q_\rightarrow$
$2, a, \{1\}, \{1\}, \rightarrow$
q_1
$2, a, \emptyset, \{1\}, \rightarrow$
q_f

Fig. 1.3. Pebble automaton in the Example 1.4

A works in the following way. It stays in state q_0 while moving to the right and non-deterministically places a new pebble at a position with label a. After placing the new pebble the automaton moves one position to the right. A continues moving to the right and after reaching a position with label a and having the same data value (as that under the first pebble) it enters the final state.

Example 1.5. The language $L_{\forall a \exists b}$, can be accepted by a pebble automaton. The automaton works the following way. A starts in state q_0, it continues moving to the right, whenever it sees an a, it drops a new pebble at the first position and enters the state $q_\rightarrow$. In the state $q_\rightarrow$ the automaton goes all the way to the left and from there starts searching for a position with a label b and having the same data value as the data value under the first pebble. Once it sees that, it lifts the pebble, continues moving right in state q_0 until it reaches the end of the input.

Indeed, the pebble automaton is too powerful, as the following theorem demonstrates.

Theorem 1.4. *Emptiness checking is undecidable for deterministic pebble automata.*

Proof. We reduce the Post's Correspondence Problem to emptiness checking deterministic pebble automata. The pebble automaton first checks whether the input is of desired form and then accepts the input if it is a solution of the PCP instance.

An instance of PCP consists of a finite number of pairs (u_i, v_i), $u_i, v_i \in \Sigma^*$. The question is whether there exists a non-empty, finite sequence $i_0, i_i, \ldots i_n$ such that:

$$w = u_{i_0} u_{i_1} \ldots u_{i_n} = v_{i_0} v_{i_1} \ldots v_{i_n}$$

This is done as follows. We take $\Sigma' = \Sigma \cup \overline{\Sigma}$, two disjoint copies of Σ, as our letter alphabet and D as the data alphabet. The words over Σ are denoted by u, v etc. and the words over $\overline{\Sigma}$ are represented as $\overline{u}, \overline{v}$ etc. The solution for the PCP can be represented in $(\Sigma' \times D)^*$ as:

$$\hat{w}_d = \begin{array}{cccccccc} a_1^{i_0} & a_2^{i_0} & .. & a_{|u_{i_0}|}^{i_0} & \overline{a_1^{i_0}} & \overline{a_2^{i_0}} & .. & \overline{a_{|v_{i_0}|}^{i_0}} \\ d_1 & d_2 & .. & d_{|u_{i_0}|} & d_1 & d_2 & .. & d_{|v_{i_0}|} \end{array} ..$$

$$\begin{array}{cccccccc} a_1^{i_1} & a_2^{i_1} & .. & a_{|u_{i_1}|}^{i_1} & \overline{a_1^{i_1}} & \overline{a_2^{i_1}} & .. & \overline{a_{|v_{i_1}|}^{i_1}} \\ d_{|u_{i_0}|+1} & d_{|u_{i_0}|+2} & .. & d_{|u_{i_0}|+|u_{i_1}|} & d_{|v_{i_0}|+1} & d_{|v_{i_0}|+2} & .. & d_{|v_{i_0}|+|v_{i_1}|} \end{array} ..$$

The string projection of $\hat{w}_d$ is $str(\hat{w}_d) = u_{i_0}\overline{v_{i_0}}u_{i_1}\overline{v_{i_1}} \ldots u_{i_n}\overline{v_{i_n}}$. Every data value appearing in the word appears exactly twice, once associated with a label in Σ and once with the corresponding label in $\overline{\Sigma}$.

We can verify these two properties by a pebble automaton A which has subroutines to check the following,

- the string projection belongs to $\{u_i\overline{v_i} \mid 1 \leq i \leq k\}^+$. In fact, this can be done by a finite automaton over a finite alphabet.
- each data value occurs exactly twice, once labelled with a letter from Σ and once with the corresponding letter in Σ'. This can be done by checking, for each a, w belongs to the languages $L_{\forall a \exists \overline{a}}$, $L_{\forall \overline{a} \exists a}$.
- the sequence of data values are the same in w and $\overline{w}$. This can be done by checking every consecutive pair occurring in the Σ-labelled part occurs in the Σ'-labelled part, and vice versa.

Thus the automaton A, by checking each property above, can verify the solution of the PCP instance. Now it is clear that if there is a data word w such that w is accepted by the automaton A then there is a solution for the PCP instance and vice versa.

Therefore emptiness of deterministic pebble automata is undecidable. □

Undecidability suggests that the machine model under consideration has strong computational power, which is reinforced by the fact that pebble automata are robust. The class is closed under logical operations (complementation, union, intersection) and its expressive power is invariant under nondeterminism and alternation.

1.8. Logics for data languages

The celebrated theorem of Büchi [12] asserts the equivalence of languages recognized by finite state automata and those defined by sentences of Monadic Second Order Logic (MSO). We consider the language of First order logic with a single binary relation $<$ and a unary relation Q_a for each $a \in \Sigma$, extended by set quantifiers. The idea is that such formulas are interpreted over words in Σ^*, with first order variables denoting positions in words, and second order (monadic) variables denoting sets of positions. MSO logics have been extensively studied not only over finite and infinite words (over finite alphabets) but also over trees and graphs.

Considering the variety of automata models we have been discussing for languages of words over infinite alphabets, a natural criterion for assessing such models is in terms of how they relate to such logical presentations. This is an extensive area of research, and we merely sketch the main ideas here.

In the same spirit as Σ-words are defined as Σ-labels over positions, we can consider data words over $(\Sigma \times D)$ as well. Guided by our decisions to consider data values only implicitly and restricting operations on data to only checking for equality, we are led to an equivalence relation on word positions: two positions are equivalent if they carry the same data value. Thus, we can consider data words to be structures with positions labelled by Σ and an equivalence relation on word positions.

This discussion suggests that we consider enriching the standard syntax of MSO with equivalence on first order terms. The syntax of $MSO(\Sigma, <, +1, \sim)$ is given by:

$$\varphi ::= a(x) \mid x = y \mid x = y + 1 \mid x < y \mid x \sim y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

Note that $<$ and $+1$ are inter-definable in the logic. The First order fragment of this logic is denoted $\mathrm{FO}(\Sigma, <, +1, \sim)$.

Let FV denote the collection of first order variables, and SV that of set variables. Given a word $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ and an interpretation $I = (I_f, I_s)$ of the variables $I_f : FV \to [n]$ and $I_s : SV \to 2^{[n]}$, we can define the semantics of formulas as follows.

$$\begin{array}{ll}
w, I \models a(x) & \text{if } a_{I_f(x)} = a \\
w, I \models x = y & \text{if } I_f(x) = I_f(y) \\
w, I \models x = y + 1 & \text{if } I_f(x) = I_f(y) + 1 \\
w, I \models x < y & \text{if } I_f(x) < I_f(y) \\
w, I \models x \in X & \text{if } I_f(x) \in I_s(X) \\
w, I \models x \sim y & \text{if } d_{I_f(x)} = d_{I_f(y)} \\
w, I \models \neg\varphi & \text{if } w, I \not\models \varphi \\
w, I \models \varphi_1 \vee \varphi_1 & \text{if } w, I \models \varphi_1 \text{ or } w, I \models \varphi_2 \\
w, I \models \exists x.\varphi & \text{if there is an } i \text{ in } [n] \text{ s.t. } w, (I_f[x \to i], I_s) \models \varphi \\
w, I \models \exists X.\varphi & \text{if there exists } J \subseteq [n] \text{ s.t. } w, (I_f, I_s[X \to J]) \models \varphi
\end{array}$$

A sentence σ is a formula with no free variables, and it is easily seen that for any w, either $w \models \sigma$ or $w \models \neg\sigma$ independent of any interpretation I. Let $L(\sigma) = \{w \mid w \models \sigma\}$.

Let $L \subseteq (\Sigma \times D)^*$. We say L is definable when there exists a sentence σ such that $L = L(\sigma)$. We are interesting in relating definable languages of data words with those recognized by automata models.

The sentence $\forall x \forall y. x \neq y \rightarrow x \not\sim y$ defines the language of data words in which no data value repeats. Similarly $\forall x \forall y. a(x) \wedge a(y) \wedge x \neq y \rightarrow x \not\sim y$ defines the language $L_{fd(a)}$. The sentence $\forall x \exists y. a(x) \rightarrow b(y) \wedge x \not\sim y$ defines the language $L_{\forall a \exists b}$.

The above examples show that definability is computationally powerful, as asserted by the following proposition.

Proposition 1.2. *The satisfiability problem for* $\mathrm{FO}(\Sigma, <, +1, \sim)$ *is undecidable [7].*

For a proof of the above claim, note that the PCP coding which we used in the undecidability of pebble automata can in fact be carried out in $\mathrm{FO}(\Sigma, <, +1, \sim)$.

The above proposition suggests that for restricting definability to implementability by devices, we should look for some fragment of the logic. There are many ways to obtain decidable fragments of quantificational logic: for example, by specifying the form of quantifier prefixes allowed [13], restricting the number of variables which can used in the formula [14], or restricting the syntactic structure of formulas, such as in Guarded fragments ([15, 16]). The study of decidable fragments of first order logics over special classes of structures is an interesting area of study in its own right ([13]).

The restriction by number of variables used turns out to be especially interesting for infinite alphabets. A careful look at the undecidability proof above shows that *three variables* suffice. On the other hand, the example languages we have been working with are all defined above using only *two variables.* This provides sufficient motivation to focus attention on the two variable fragment of the logic above.

$\mathrm{FO}^2(\Sigma, <, +1, \sim)$, is the set of all formulas in the language of $\mathrm{FO}(\Sigma, <, +1, \sim)$ using at most two variables. As the following theorem shows, such a restriction pays off.

Theorem 1.5. *Satisfiability of* $\mathrm{FO}^2(\Sigma, <, +1, \sim)$ *is decidable [7].*

Proof. Here we only sketch the proof, the details are intricate and require a more elaborate presentation. We refer to the logic simply as FO^2.

We denote by α, β etc unary quantifier free formulas (which are called *types*). It is easy to see that FO^2 can express the following properties.

- data-blind properties, i.e. properties not using the predicate $\sim$. These are the word languages over Σ expressible by FO^2.
- Each class contains at most one occurrence of α: This can be expressed by the formula $\forall xy. \alpha(x) \wedge \alpha(y) \wedge x \sim y \rightarrow x = y$.

- In each class every α occurs before every β: This can be expressed by the formula $\forall xy.\alpha(x) \wedge \beta(y) \wedge x \sim y \rightarrow x < y$.
- Each class with an α has also a β: This can be expressed by the formula $\forall x \exists y.\alpha(x) \rightarrow \beta(y) \wedge x \sim y$.
- If a position is in a different class than its successor then it has type α: This can be expressed by $\forall xy.x \not\sim y \wedge y = x + 1 \rightarrow \alpha(x)$.

The idea behind the proof is to convert any given formula to a disjunction of conjunction of formulas of the above kinds (which is called Data Normal Form). In order to do this we first convert the given formula to the so-called Scott Normal Form:

$$\left(\forall xy.\chi \wedge \bigwedge_i \forall x \exists y.\chi_i\right)$$

A formula in this form is then converted to the form

$$\bigwedge_i \theta_i$$

where each θ_i belongs to one of the five kinds.

The crucial next step is the fact that each of these formulas can be recognized by a Class Memory automaton. The existential predicates correspond to nondeterminism in the automaton. Since CM_A are closed under union, intersection and projection we can compose automata corresponding to the subformulas to get an automaton corresponding to a formula. Hence the satisfiability of the formula reduces to non-emptiness problem of the automaton, which is decidable. $\square$

This connection can be strengthened as the following corollary shows.

Corollary 1.1. *Languages recognized by CM_A are the languages expressed by* $\mathrm{EMSO}^2(\Sigma, <, +1, \sim, \oplus)$.

Here EMSO^2 stands for existential monadic second order formulas with their first order fragment containing only two variables and $\oplus$ stands for the *class successor* relation.

1.8.1. *Temporal logics for data languages*

If Büchi's theorem relates MSO definability and recognizability by automata, another celebrated theorem due to Kamp [17] relates FO definability to temporal logics. These are modal logics with modalities such as *eventually, until, since* etc. The success of formal methods in system verification has to do with the ease of temporal logics both in terms of algorithmic tools and reasoning.

Hence, when we consider description of data languages, temporal logic is a natural candidate. We need modalities to mark positions, and recall marks when needed. A temporal logic developed by [8] adds freeze operators to propositional linear time temporal logic. These come in two forms: unary modalities $\downarrow_i$, $i > 0$, and atomic formulas $\uparrow_i$. Informally, the semantics is as follows. $\downarrow_i$ stores the current data value in register i and $\uparrow_i$ checks whether the current data value equals to the data stored in the register i.

The syntax of the logic is given by:

$$\varphi ::= \top \mid a \mid \uparrow_i \mid \varphi \wedge \varphi \mid \neg\varphi \mid X\varphi \mid \varphi U \varphi \mid \downarrow_i \varphi$$

The semantics (over finite words) is defined in the obvious manner.

The formula $F \downarrow_1 XF \uparrow_1$ expresses the set of data words in which at least one data value repeats. Hence its negation expresses the language of all data words in which no data value repeats. $G(a \rightarrow \downarrow_1 XF(\uparrow_1 \wedge b)$ expresses the set of data words in which in every class, if an a occurs there is a b occurring later.

Again, the logic is too powerful: the satisfiability problem for LTL with freeze operators is undecidable. However satisfiability of LTL-freeze with one register is decidable [8]. Also, there is a natural definition of register automata corresponding to LTL-freeze, though this definition is not equivalent to the register machines we discussed earlier.

LTL-freeze cannot express the language $L_{\forall a \exists b}$, and this is precisely because of the absence of past operators. Similarly FO^2 cannot express the following language, which LTL with freeze can: between two a of the same class there is an a which belongs to some other class. This is due to the shortage of variables in FO^2. Hence in general FO^2 and LTL-freeze are incomparable.

1.9. Related models and logics

Though register automata are relatively weak, they possess an interesting theory, and the algebraic and rational aspects of register automata have been studied. As an extension of recognizability by register automata, in [18] a Myhill-Nerode theorem for data strings is shown. [19] proposes a notion of regular expressions for data languages.

In [10], an extension of register automata is offered in which the automaton can guess the data values, called look-ahead register automata. An equivalent characterization in terms of regular expressions and grammar is presented for this class of automata. One interesting property of look-ahead register automata is that they are closed under reversal.

[20] seeks a notion of context-free data languages. They extend register automata with stack which can hold data values, called pushdown register automata. The automaton has the ability to rewrite the registers with arbitrary data values. The stack operations transfer symbols from the registers to the stack. The transition is based on the register which matches the top of the stack. The automaton is able to accept some analogous languages in this way, for example $w\#w^r$, where w is a data string. A definition of context-free grammars over infinite alphabets is proposed and is shown equivalent to pushdown register automata. A lemma similar

to that of register automata holds in the case of pushdown automata which says that given a pushdown register automaton there exists an n such that if the automaton accepts a word it accepts a word with at most n data values. While it is not apparent how to determine n from the definition of the automata, the observation shows that the emptiness problem for this class of automata is decidable.

In [21], a notion of monoid recognizability for data languages is introduced. A corresponding extension of register automata is shown to be equivalent to recognition by monoids. This model is an extension of register automata with an equivalence relation of finite index. During a transition the automaton, in addition to reading the register, checks to which equivalence class the register configuration belongs to, selects the next state and updates the register accordingly. Though this certainly enhances computational' power, the automaton can still remember only finitely many data values. Moreover, decidability of the emptiness problem is obtained only in the case where register updates respect the equivalence. In [22] an EMSO characterization of monoid recognizablity is shown.

Another simple computational model, based on transducers is the Data Automaton model introduced in [7]. A Data Automaton $A = (B, C)$ consists of:

- a non-deterministic letter-to-letter string transducer B, the base automaton, with input alphabet Σ and some output alphabet Γ, and
- a non-deterministic automaton C, the class automaton, with input alphabet Γ

A data word w is accepted by A if there is an accepting run of B on the string projection of w, giving an output string $\gamma_1\gamma_2 \ldots \gamma_n \in \Gamma^*$, such that for each class $\{x_1, x_2, \ldots, x_k\} \subseteq \{1, 2, \ldots, n\}$, $x_1 < x_2 < \ldots < x_k$, the class automaton accepts $\gamma_{x_1}\gamma_{x_2} \cdots \gamma_{x_k}$.

Example 1.6. The language $L_{fd(a)}$, can be accepted by a data automaton $A = (B, C)$ as follows.

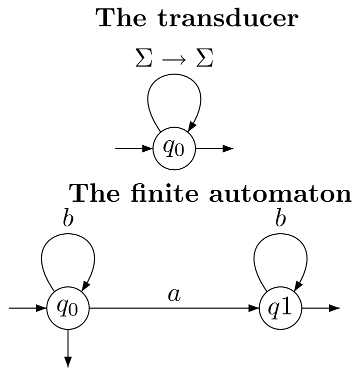

Fig. 1.4. Automaton in the Example 1.6

- The base automaton $B\colon \Sigma \to \Sigma$ is a copy automaton which copies the input (string part) to the output.
- The class automaton C accepts $b^* + b^*ab^*$, words containing at most one a.

Theorem 1.6 ([6]). *Class Memory automata and Data automata are expressively equivalent.*

1.10. Conclusion

We have taken the reader on a touristic journey of automata models over infinite alphabets, pointing out interesting sights very briefly, without studying any of the models as they really should be. We have looked at the models only from the perspective of decidability of the emptiness problem, and if there is any moral to the story, it is only this: systems with unbounded data give rise to infinite state systems, and identifying machine classes with manageable complexity is a challenge. Our bag of tools for addressing such problems needs considerable enrichment, and it is hoped that automata theory and logic will contribute substantially towards this. Developing an underlying algebraic theory may be an important step in this direction.

On a positive note, automata over infinite alphabets provide an abstract theoretical model in which we seem to be able to represent situations that arise in a wide variety of contexts: systems with unboundedly many processes such as those studied in infinite state verification, communicating systems such as Petri nets, the verification of software programs with recursion and pointers, navigation over trees representing semistructured data, web services handling unboundedly many clients, to list a few. Such diversity suggests that use of techniques across areas may lead to new techniques in the theory of automata over infinite alphabets.

References

[1] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation.* (Addison-Wesley, 1979). ISBN 0-201-02988-X.
[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Princiles, Techniques, and Tools.* (Addison-Wesley, 1986). ISBN 0-201-10088-6.
[3] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking.* (The MIT Press, 2000). ISBN 0-262-03270-8.
[4] M. Kaminski and N. Francez, Finite-memory automata, *Theor. Comput. Sci.* **134**(2), 329–363, (1994).
[5] F. Neven, T. Schwentick, and V. Vianu, Finite state machines for strings over infinite alphabets, *ACM Trans. Comput. Log.* **5**(3), 403–435, (2004).
[6] H. Björklund and T. Schwentick. On notions of regularity for data languages. In eds. E. Csuhaj-Varjú and Z. Ésik, *FCT*, vol. 4639, *Lecture Notes in Computer Science*, pp. 88–99. Springer, (2007). ISBN 978-3-540-74239-5.
[7] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable

logic on data trees and xml reasoning. In ed. S. Vansummeren, *PODS*, pp. 10–19. ACM, (2006). ISBN 1-59593-318-2.

[8] S. Demri and R. Lazic. Ltl with the freeze quantifier and register automata. In *LICS*, pp. 17–26. IEEE Computer Society, (2006).

[9] H. Sakamoto and D. Ikeda, Intractability of decision problems for finite-memory automat a., *Theor. Comput. Sci.* **231**(2), 297–308, (2000).

[10] D. Zeitlin. Look-ahead finite-memory automata. Master's thesis, Technion - Israel Institute of Technology, Tamuz, 5766, Haifa, Israel (July, 2006).

[11] R. Ramanujam and M. Amaldev, Class counting automata on datawords, *manuscript.* (2007). URL `http://www.imsc.res.in/~amal/cca.pdf`.

[12] J.R. Büchi, Weak second order arithmetic and finite automata, *Z. Math. Logik Grundlagen Math.* **6**, 66–92, (1960).

[13] E. Börger, E. Grädel, and Y. Gurevich, *The Classical Decision Problem.* Perspectives in Mathematical Logic, (Springer-Verlag, Berlin, 1997).

[14] E. G. Adel, P. G. Kolaitis, and Y. Vardi. On the decision problem for two-variable first-order logic (Aug. 28, 1997). URL `http://citeseer.ist.psu.edu/435533.html;http://www.math.ucla.edu/~asl/bsl/0301/0301-003.ps`.

[15] H. Andréka, I. Németi, and J. van Benthem, Modal logic and bounded fragments of predicate logic, *Journal of Philosophical Logic.* **27**(3), 217–274, (1998).

[16] E. Gradel and R. Aachen. On the restraining power of guards (Dec. 20, 1999). URL `http://citeseer.ist.psu.edu/386986.html;http://www-mgi.informatik.rwth-aachen.de/Publications/pub/graedel/Gr-jsl99.ps`.

[17] H. Kamp. *On tense logic and the theory of order*. PhD thesis, UCLA, (1968).

[18] N. Francez and M. Kaminski, An algebraic characterization of deterministic regular languages over infinite alphabets, *Theor. Comput. Sci.* **306**(1-3), 155–175, (2003).

[19] M. Kaminski and T. Tan. Regular expressions for languages over infinite alphabets. In eds. K.-Y. Chwa and J. I. Munro, *COCOON*, vol. 3106, *Lecture Notes in Computer Science*, pp. 171–178. Springer, (2004). ISBN 3-540-22856-X.

[20] E. Y. C. Cheng and M. Kaminski, Context-free languages over infinite alphabets, *Acta Inf.* **35**(3), 245–267, (1998).

[21] P. Bouyer, A. Petit, and D. Thérien, An algebraic approach to data languages and timed languages, *Information and Computation.* **182**(2), 137–162 (May, 2003). URL `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BPT-IetC.pdf`.

[22] P. Bouyer, A logical characterization of data languages, *Information Processing Letters.* **84**(2), 75–85 (Oct., 2002). URL `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/Bou-IPL2002.ps`.