3.1 Introduction

The theory of finite state automata over (finite) words is an area 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 [HU79, ASU86], model checking finite state systems [CGP99] 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 do not 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 $\mathcal{T}$. 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: $\Gamma = \{\#, 0, 1, 10, 11, \ldots\}$. Now, $L = \{w = b_0\#b_1\# \ldots b_n \mid w \in \Gamma^*, \exists i, j. b_i = b_j\}$. Note further that $\Gamma$ 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, \ldots), 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.

### 3.2 Languages of data words

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 happen 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 \( \Sigma \times \Gamma \), where \( \Sigma \) is a finite set, and \( \Gamma \) is a countably infinite set. Usually, \( \Sigma \) is called the **letter alphabet** and \( \Gamma \) is called the **data alphabet**. Elements of \( \Gamma \) are referred to as **data values**. We use letters \( a, b \) etc to denote elements of \( \Sigma \) and use \( d, d' \) to denote elements of \( \Gamma \).

The letter alphabet is a way to provide ‘contexts’ to the data values. In the case of XML, \( \Sigma \) consists of tags, and \( \Gamma \) consists of data values. Consider the XML description: 

\[
\text{<name> "Tagore"</name>}
\]

the tag \(<\text{name}>\) can occur along with different strings; so also, the string \("\text{Tagore}"\) 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 \((\Sigma \times \Gamma)^*\). A collection of data words \( L \subseteq (\Sigma \times \Gamma)^* \) is called a **data language**. In this thesis, 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 \( \text{str}(w) = a_1a_2\ldots a_n \), is the projection of \( w \) to its \( \Sigma \) components. Let
Chapter 3. Automata for data words

| \( L_{\geq n} \) | All data words in which at least \( n \) distinct data values occur. |
| \( L_{< n} \) | All data words in which every data value occurs at most \( n \) times. |
| \( L_{a*b} \) | All data words whose string projections are in the set \( a^*b^* \). |
| \( L_a \) | All data values under the label \( a \) are different. |
| \( L_{a\rightarrow b} \) | All data values occurring under \( a \) occurs under \( b \) as well. |
| \( L_{dd} \) | There is a data value which occurs in consecutive positions. |

Figure 3.1: Sample data languages

\( i \in [n] \). 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 in Figure 3.2 some example data languages which we will keep referring to in the course of our discussion; these are over the alphabet \( \Sigma = \{a, b\}, \Gamma = \mathbb{N} \).

Let \( \cdot \) denote concatenation on data words. For \( L \subseteq (\Sigma \times \Gamma)^* \), consider the Myhill-Nerode equivalence on \((\Sigma \times \Gamma)^* \) induced by \( L \): \( w_1 \sim_L w_2 \) iff \( \forall w \in (\Sigma \times \Gamma)^*, w_1 \cdot w \in L \iff w_2 \cdot w \in L \). The language \( L \) 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 \( \sim_{L_a} \) 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 \( L_a \); 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 and the following chapters.
3.3 Formulating an automaton mechanism

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, it is absolutely necessary that, we consider only data alphabets $\Gamma$ 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 tape-head is 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 $\Gamma$ 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 tape-head has to be a $\Sigma_0^1$ 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 finite state automaton uses only finitely many data values in its transition relation. 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 (VAS), 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 data words whose string projections belong to the letter language (subset of $\Sigma^*$) 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.

### 3.4 Register automata

The simplest form of memory is a finite random access read-write storage device, traditionally called register. In Register automata [KF94], 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. The definition we present below is a close variant of the definition in [KF94]. In terms of complexity of decision problems and language acceptance they are equivalent.
Definition 3.4.1. A $k$-Register automaton $A$ is given by $A = (Q, \Sigma, \Delta, \bot, q_0, F)$, where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\bot$ is the empty register symbol. The transition relation is $\Delta \subseteq (Q \times \Sigma \times [k] \times Q) \cup (Q \times \Sigma \times Q \times [k])$. For $p, q \in Q$, $a \in \Sigma$, $i \in [k]$, transitions of the form $(p, a, i, q)$ are called read transitions and transitions of the form $(p, a, q, i)$ are called write transitions.

The automaton $A$ has $k$ registers. $\bot$ is used above to denote an uninitialized register. A configuration of $A$ is of the form $(q, h)$ where $q \in Q$ denotes the current state and $h : [k] \to (\Gamma \cup \{\bot\})$ is a function from $[k]$ to $(\Gamma \cup \{\bot\})$, such that if for $i \neq j$, $h(i) \in \Gamma$ and $h(j) \in \Gamma$ then $h(i) \neq h(j)$, representing the current register configuration. For convenience, sometimes we identify the function $h$ with the set $\text{range}(h) = \cup_i \{h(i)\}$, for instance by $d \in h$ we mean that the data value $d$ is in the registers. The working of the automaton is as follows. Suppose that $A$ is in state $p$, with each of the registers $i$ holding data value $d_i$, and its input is of the form $(a, d)$. Now there are two cases:

- If $d \neq d_i$ for all $i$, then a register write is enabled and the automaton can make a write transition $(p, a, q, i)$ storing data value $d$ in register $i$ and the next state becomes $q$.

- Suppose that $d = d_i$, for some $i \in [k]$, and $(p, a, i, q) \in \Delta$. Then this read transition is enabled and when applying the transition, the registers are left untouched and the next state becomes $q$.

A run of $A$ on a data word $w = (a_1, d_1)(a_2, d_2)\ldots(a_n, d_n)$ is a sequence $\gamma = (q_0, h_0)(q_1, h_1)\ldots(q_n, h_n)$, where $(q_0, h_0)$ is the initial configuration of $A$, 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 $\Delta$. $\gamma$ is accepting if $q_n \in F$. The language accepted by $A$, denoted $L(A) = \{w \in (\Sigma \times \Gamma)^* \mid A$ has an accepting run on $w\}$.

Observe that at any configuration all the data values stored in the registers are different.

Example 3.4.2. Recall the language $L_a$ mentioned earlier: it is the set of all data words in which all the data values in context $a$ are distinct. The language $\overline{L_a}$ can
be accepted by a 2-register automaton $A = (Q = \{q_0, q_1, q_f\}, \Sigma, \Delta, \bot, q_0, F = \{q_f\})$, where $\Delta$ consists of:

$$\Delta = \left\{ (q_0, \Sigma, 1, q_0), (q_0, \Sigma, 0, 1), (q_0, a, 1, q_1), (q_0, \Sigma, q_1, 1), (q_1, \Sigma, 1, q_1), (q_1, \Sigma, 1, q_2), (q_1, a, 1, q_f), (q_f, \Sigma, 1, q_f), (q_f, \Sigma, 2, q_f), (q_f, \Sigma, q_f, 2) \right\}$$

$A$ 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 $q_1$ nondeterministically 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.

The automaton is shown in the Figure 3.2.

**Example 3.4.3.** The language $L_{dd}$ is accepted by a 1-register automaton $A = (Q = \{q_0, q_f\}, \Sigma, \Delta, q_0, F = \{q_f\})$ where

$$\Delta = \left\{ (q_0, \Sigma, 1, q_f), (q_f, \Sigma, 1, q_f), (q_0, \Sigma, q_0, 1), (q_f, \Sigma, q_f, 1) \right\}$$

The automaton (as shown in Figure 3.3) always stores the data values in the register 1 and stays in state $q_0$, if it sees a data value repeating it goes to state $q_f$ and stays there.

**Example 3.4.4.** A finite state automaton is a 0-register automaton. Since $a^*b^*$ is regular, the language $L_{a^*b^*}$ is accepted by a register automaton.
Example 3.4.5. The family of languages $L_{\exists n}$ is accepted by $n$-register automata with $n + 1$ states $q_0, \ldots, q_n$ (shown in Figure 3.4) in the following way. The automaton fills up the registers successively with new data values while keeping the number of registers filled in the states. Finally it accepts the word if the state $q_n$ is reached.

However, the languages $L_{<n}, L_a$ and $L_{a \rightarrow b}$ are not accepted by register automata. Below we see why it is so.

Note that a register 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. Again, our formulation is slightly different from the corresponding lemma in [KF94].
Chapter 3. Automata for data words

Lemma 3.4.6. If a k-register automaton $A$ accepts any word at all, then it accepts a word containing at most $k + 1$ distinct data values.

Proof. Let $w = (a_1, d_1)(a_2, d_2)\ldots(a_n, d_n)$ be a data word accepted by $A$ and $\rho = (q_0, h_0)(q_1, h_1)\ldots(q_n, h_n)$ be an accepting run of $A$ on $w$. If the size of the set $D = \{d_1, \ldots, d_n\}$ is $k + 1$ then the claim is proved else let $D' \subset D$ be a subset of size $k + 1$. For register configuration $h$ and data values $d_1, d_2$, we denote by $h[d_1/d_2]$ the register configuration obtained from $h$ by replacing $d_1$ by $d_2$. Let,

$$w' = (a_1, d'_1)(a_2, d'_2)\ldots(a_n, d'_n)$$

$$\rho' = (q_0, h_0)(q_1, h'_1)\ldots(q_n, h'_n), \forall i \geq 1, h'_i = h_i[d_i/d'_i]$$

where $d'_i \in D'$ and $d'_i \in h'_{i-1}$ if and only if $d_i \in h_{i-1}$. We show by induction on $n$ that $A$ has a run $\rho'$ on $w'$ as follows. For the base case, fix $d'_1 = d_1$ and $D' \subseteq D$ such that $d_1 \in D'$ and trivially $(q_0, h_0)(q_1, h_1)$ satisfies the conditions. For the inductive step assume that there is a partial sequence $(a_1, d'_1)(a_2, d'_2)\ldots(a_{i-1}, d'_{i-1})$ and $(q_0, h_0)(q_1, h'_1)\ldots(q_{i-1}, h'_{i-1})$ satisfying the above conditions. Assume $d_i$ is stored in register $j$ in $h_{i-1}$, that is $h_{i-1}(j) = d_i$. We define $d'_i$ to be $h'_{i-1}(j) = d$ and $h'_i = h_i[d_i/d]$. If $d_i$ is not in $h_{i-1}$ then we choose a data value $d \in D'$ not appearing in $h'_{i-1}$ and define $d'_i = d$. Observe that in both these cases the conditions are preserved. However, in order to show that $w'$ is accepted by $A$, it remains to be proved that $\rho' = (q_0, h'_0)(q_1, h'_1)\ldots(q_n, h'_n)$ is an accepting run for $w'$. We prove this using induction again. For the base case it is trivial. For the inductive step, assume $d_i$ is in $h_{i-1}$ in which case there is a read transition $(q_{i-1}, a_i, j, q_i)$ where $h_{i-1}(j) = d_i$. Since $d'_i$ is in $h'_{i-1}$ the same transition is enabled at $(q_{i-1}, h'_{i-1})$. Similarly, if $d_i$ is not in $h_{i-1}$ there is a write transition $(q_{i-1}, a_i, q_i, j)$. Since $d'_i$ is also not in $h'_{i-1}$ the same transition is enabled at $(q_{i-1}, h'_{i-1})$. This completes the proof. \[\square\]

Note that the language $L_a$ 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 $L_a$ can be 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 3.4.7.** 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 $D' \subseteq \Gamma$, $|D'| = k + 1$ be a subset of $\Gamma$ containing $k + 1$ different values. 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 $\text{NP}$, 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 $\text{NP}$ in [SI00]. 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 [NSV04, KF94, Zei06].

The word problem for register automata are $\text{NP}$-complete, while for deterministic register automata it is $\text{P}$-complete [SI00].

### 3.5 Data and Class Memory automata

The weakness of register automata arises from its finite memory. A way to overcome this is by allowing unbounded memory and hashtables provide an easy mechanism for providing that. Below we discuss an equivalent formulation of such an automaton.
Chapter 3. Automata for data words

A **Data automaton** \([BDM^{+11}]\) is a composite automaton consisting of a finite state transducer \(B\) and a finite state automaton \(C\). They use an internal alphabet \(\Sigma'\) for communication. Formally:

**Definition 3.5.1.** A data automaton is a tuple \(A = (B,C)\) where \(B\) is a finite state transducer, given by the tuple \(B = (Q_b, \Sigma, \Sigma', \Delta_b, O_b, I_b, F_b)\), with input alphabet \(\Sigma\) and output alphabet \(\Sigma'\). The automaton \(C = (Q_c, \Sigma', \Delta_c, I_c, F_c)\) is a finite state automaton with alphabet \(\Sigma'\).

A run \(\rho\) of a data automaton on data word \(w\) is defined in the following manner;

Let \(w' = a_1 ... a_n\) be the string projection of \(w\). Let \(\rho_B = \delta_1 ... \delta_n \in \Delta_b^*\) be a run of \(B\) on \(w'\). The run \(\rho_B\) uniquely defines an output word \(w'' = O_b(\delta_1)...O_b(\delta_n)\) (See section [2.1.2]). Let \(D(w)\) be the set of data values occurring in \(w\) and let \(w''_d\) be the subword of \(w''\) formed by the positions labelled by \(d \in D(w)\). For each \(d \in D(w)\), let \(\rho_d\) be a run of the automaton \(C\) on \(w''_d\). Define the run \(\rho\) as \((\rho_B, \{\rho_d \mid d \in D(w)\})\). We say \(\rho\) is successful if (1) \(\rho_B\) is a successful run of \(B\) on \(w'\) (2) For each \(d \in D(w)\), \(\rho_d\) is a successful run of \(C\) on \(w''_d\).

**Example 3.5.2.** The language \(L_a\) is easily accepted by the following way. The intermediate alphabet is \(\Sigma\) itself. The transducer \(B\) is a copy machine, copies every letter to the output. The automaton \(C\) accepts the language \(\Sigma^*a\Sigma^*a\Sigma^*\). It is clear that if in \(w\) there is a class with at least two \(a\)'s then \(C\) cannot have a successful run over that class.

**Example 3.5.3.** For accepting the language \(L_{dd}\), choose the intermediate alphabet to be \(\{0, 1\}\). While reading the string projection the transducer \(B\) chooses two consecutive positions and label them by ‘1’, all other positions are labelled ‘0’. The automaton \(C\) accepts the language \(0^*10^*0^* + 0^*\). Note that the automaton \(C\) specifies that in each class either all positions are labelled ‘0’ or there are exactly two positions with label ‘1’. Since the transducer \(B\) outputs ‘1’ on two positions, there is at least one class which contains a ‘1’ and because of \(C\) that class contains two ‘1’s. Finally, since \(B\) outputs exactly two ‘1’s on consecutive positions it can be inferred that there exist consecutive positions labelled with the same data value.

**Example 3.5.4.** The language \(L_{<n}\) is accepted in the following way. Again, the transducer \(B\) is a copy machine and the internal alphabet is \(\Sigma\). The finite state automaton \(C\) accepts the language \(\Sigma^0 \cup \Sigma^1 \ldots \cup \Sigma^n\).
Example 3.5.5. In the case of $L_{a^*b^*}$ the automaton $B$ accepts the language $a^*b^*$ and $C$ is a machine accepting all strings.

Example 3.5.6. For the language $L_{a\rightarrow b}$, the automaton $B$ is a copy machine. Automaton $C$ accepts the language $(a^*ba^*)^*$ which is the set of strings $w$ such that $w$ contains a ‘b’ if it contains an ‘a’.

Now, we give the definition of the finite state automaton equipped with a hash table called Class Memory Automaton (shortly CMA) [BS10].

Definition 3.5.7. A class memory automaton is a tuple $A = (Q, \Sigma, \Delta, q_0, F_\ell, F_g)$ 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 class memory automaton is equipped with a hashtable $h$ which maps from the set of data values $\Gamma$ to a finite set of hash values. 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 $q$ when the hash value for $d$ is $s$, 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 $F_\ell$. Thus acceptance depends on the memory of the data encountered.

Formally, a hash function is a map $h : \Gamma \rightarrow (Q \cup \{\bot\})$ 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 $d \in \Gamma$.

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

$$h'(d') = \begin{cases} q & \text{if } d = d', \\ h(d') & \text{if } d \neq d'. \end{cases}$$
A run of CMA $A$ 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 $(q_0, h_0)$ is the initial configuration of $A$, 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 $\Delta$. $\gamma$ is accepting if $q_n \in F_g$ and for all $d \in \Gamma$, $h_n(d) \in F_l \cup \{\perp\}$. The language accepted by $A$, denoted $L(A) = \{w \in (\Sigma \times \Gamma)^* \mid A$ has an accepting run on $w\}$.

**Example 3.5.8.** The language $L_a$ 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 $\Delta$ 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 $\{q_a, q_b\}$ and $F_g$ is the set $Q$. The idea is that for each class the automaton remembers if it has seen an ‘$a$’ by means of the hash function. A run terminates erroneously if in a class a second ‘$a$’ is seen. Finally the run is successful if all classes terminates in the local final state $q_b$.

![Figure 3.5: CMA accepting the language $L_a$.](image)

The automaton is shown in the Figure 3.5. The local accepting states are shown in red, while global accepting states are circled.

**Example 3.5.9.** The language $L_{a \rightarrow b}$ is accepted by CMA in the following fashion (shown in the Figure 3.6). The automaton has three states $q_0, q_a, q_b$ where $q_0$ is the initial state. For each class if the hash function carries the state $q_a$ then it indicates that so far in the class only ‘$a$’ has appeared. Similarly if the hash function indicates $q_b$ then it denotes that the class contains at least one ‘$b$’. The automaton updates the hash function by appropriately changing states on each pair. Finally the automaton accepts if all the classes end in the local final state $q_b$. 

26
Example 3.5.10. The language $L_{dd}$ is accepted by a three state CMA (shown in the Figure 3.7) in the following way. The automaton starts in the initial state $q_0$. At some point during the run nondeterministically the automaton changes state to $q_1$. The automaton checks if the following data value is the same by checking the hash function (if it is the case then the state associated with the data value should be $q_1$) and then moves to the final state $q_f$. The local final states are irrelevant in this case.

The following two important properties of CMA are proved in [BS10].

**Theorem 3.5.11** ([BS10]). CMA and Data automata are expressively equivalent. The translations from CMA to Data automata and vice versa are in $P$.

**Theorem 3.5.12** ([KP94][BS10]). Register automata are strictly less powerful than CMA in terms of expressiveness.

Next we discuss the emptiness problem for CMA, which follows from the decidability of Data automata. Here we give a proof of the same fact.
Chapter 3. Automata for data words

Theorem 3.5.13 ([BDM+11, BS10]). The emptiness problem for CMA is decidable.

Proof. Let $A = (Q, \Sigma, \Delta, q_0, F_l, F_g)$ be a given CMA. 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 $s \neq \bot$ we add a new transition $t_\delta$ such that $t_\delta = \{p, s^c\}$ and $t_\delta^* = \{q, q^c\}$. For each $\delta = (p, a, \bot, q)$ where we add a new transition $t_\delta$ such that $t_\delta = \{p\}$ and $t_\delta^* = \{q, q^c\}$. We add additional transitions $t_{(p, q)}$ for each $p \in F_g, q \in F_l$ such that $t_{(p, q)} = \{p, q^c\}$ and $t_{(p, q)}^* = \{p\}$. 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 automaton 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).

Thus emptiness for CMA is reduced to reachability in Petri nets which is known to be decidable. As it happens, it is also as hard as Petri net reachability [BDM+11]. Since the latter problem is not even known to be elementary, we need to look for subclasses with better complexity. CMA 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.

The word problem for CMA is NP-complete and the complexity remains the same for the deterministic subclass as well [BS10].

28
3.6 Discussion

In this chapter we saw two popular automaton models for data words, Register and Class Memory automata. While lacking in expressive power register automata have decision problems of relatively low complexity. Class memory automata, on the other hand, have better expressive power but their decision problems are of very high complexity. In the next chapter we discuss a model which falls between these classes both in terms of expressive power and complexity of decision problems.