University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Rafał Stefański
Student no. 345664

An automaton model for orbit-finite monoids

Master's thesis
in COMPUTER SCIENCE

Supervisor:
prof. dr hab. Mikołaj Bojańczyk

December 2018

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.
The thesis has never before been a subject of any procedure of obtaining an academic degree.
Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

We introduce a new automaton model and compare it with two existing models for recognizing data languages – deterministic register automata and orbit-finite monoids. We show that every language recognized by the new model can also be recognized by an orbit-finite monoid. It follows that some languages can be recognized using deterministic register automata, but cannot be recognized by this new model. We conjecture that the new model is equivalent to orbit-finite monoids.

Keywords

Register automata, Orbit-finite sets, Monoids, Data words

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatyka

Subject classification

Theory of computation
Formal language and automata theory

Tytuł pracy w języku polskim

Model automatowy dla monoidów orbitowo skończonych

Contents

Introduction 5

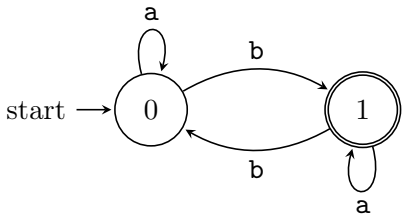
1. Register automata 7
1.1. Data words and data languages 7
1.2. Deterministic register automata 7
1.2.1. Examples 9

2. Orbit-finite sets and orbit-finite monoids 11
2.1. Orbit-finite sets 11
2.1.1. Sets with atoms 11
2.1.2. Orbit finiteness 12
2.2. Languages over orbit finite alphabets and monoids 13
2.2.1. Monoids with atoms 13
2.3. Examples 13
2.3.1. Comparison with deterministic register automata 15

3. An automaton model for orbit-finite monoids 17
3.1. Motivation 17
3.2. Definition 17
3.3. Examples 18
3.4. Connection with orbit-finite monoids 21
3.5. Comparison with other models 24
3.6. Further work 27

Introduction

One of the best-known parts of the formal language theory is the class of *regular languages*. It can be defined as the class of all the languages that can be recognized by a *deterministic finite automaton* (like the one below):



To check if a word (e. g. `abbab`) belongs to the language recognized by this automaton, we set a token in the 'start' state, and then, going through every letter in the word, we move the token as directed by the arrows. The input word belongs to the language if, in the end, the token is placed in a state marked with a double circle (an accepting state). In our example this would look like this:

$$0 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{b}} 0 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{b}} 1$$

We finish in state 1, which is marked with a double circle, so the word `abbab` belongs to the language.
One of interesting properties of this class is its robustness – we could add new features to the automaton without changing its expressive power. Some of the most notable extensions are:

- NFA – a *nondeterministic finite automaton* can have states with multiple outgoing arrows marked with the same letter. A word w will be accepted if there exists a path from the starting state to any of the accepting states, such that its labels form the word w. This means that whenever the automaton has a 'choice', we can assume that it will guess the correct option (provided that it exists).

- 2DFA – a *deterministic two-way finite automaton* has transitions that can also decide if the 'automaton head' will move one step backwards or one step forwards in the input word.

- 2NFA – a *nondeterministic two-way finite automaton* is a combination of an NFA with a 2DFA. It can move backwards and forwards in the input word and (when in doubt) it can guess the correct transition that will eventually lead to an accepting state (if only such transition exists).

Other well-known definitions of regular languages involve regular expressions such as

$$\texttt{a}^*\texttt{b}(\texttt{a}^*\texttt{ba}^*\texttt{ba}^*)^*$$

or finite monoids (discussed in the Chapter 3 of this thesis).
When we study languages over infinite alphabets this robustness usually fades away. For example, when considering data languages (described in Chapter 1 of this thesis), each of the presented variants (DFA, NFA, 2DFA, 2NFA, regular expressions and monoids) defines a different class of languages.
In this thesis, we propose a new model of register automata that works with infinite alphabets and we compare the class of languages it defines with some important classes of languages over infinite alphabets. The main difference between the new model and the already known variants of register automata (e.g. the one described in the Chapter 1 of this thesis) is that the registers of the new model are single-use only. To illustrate this idea (and to conclude this introduction) we present the following riddle.

Riddle. *Imagine a chemical laboratory that receives many samples of substances. At the end of each the day, the scientists that work in this lab have to report the approximate number of different substances seen that day – they have to decide if they have seen* 1 *substance,* 2 *substances,* 3 *substances, or* `many` *substances (which can be interpreted as any number greater than* 3*). For example, if the lab receives* 100 *samples of water, they should report that they have seen only one substance, and if they receive one sample of water, one sample of methane, one sample of ethane, and one sample of propane, they should report that they have seen* `many` *substances. All the substances received by the lab may only be stored in special test tubes and the laboratory is equipped with just* 10 *of them. The concentration of the samples is so low, that the only way of telling if two test tubes contain the same compound is to use a special machine. The machine accepts two test tubes, processes them, and produces two possible outputs:* `same` *or* `different`*. There is, however, a side effect of this procedure – during the examination the machine destroys the contents of the test tubes, leaving them empty. The laboratory receives the samples one by one. After receiving a sample, the scientists can fill up the test tubes and use the machine as many times as they want to (the volume of the sample they receive is much larger than the volume of a test tube). However, before receiving the next sample, they have to pass the current one forward, never to see it again. Is it possible for the lab team to perform the measurements in such a way, that will lead them to a valid report at the end of each day?*

Chapter 1

Register automata

In this section, we introduce data words and register automata. They were first defined in [3], but we are going to base this section on the first chapter of [2].

1.1. Data words and data languages

Define a *data word* to be a word over the alphabet $\Sigma \times \mathbb{A}$, where Σ is a finite set and $\mathbb{A}$ is a fixed infinite set. The Σ part of a letter is called its *label*, and the $\mathbb{A}$ part is called its *data value*. Intuitively, a *data language* is any set of data words that can be defined without explicitly using any value from $\mathbb{A}$. Moreover the only operation we can perform on elements from $\mathbb{A}$ is to check if two of them are equal. Valid data languages for $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ are for example:

1. The first letter's label is equal to `a`.
2. The first letter's data value is equal to the last letter's data value.
3. The first letter's data value appears somewhere else in the word.
4. Some data value appears at least twice in the word.
5. Some data value appears at least once with label `a` and at least once with label `b`.

Formally, we say that $L \subseteq (\Sigma \times \mathbb{A})^*$ is a data language, if for every permutation of atoms

$$\pi : \mathbb{A} \to \mathbb{A}$$

we have

$$\underset{w}{\forall}\, w \in L \quad \Leftrightarrow \quad \pi(w) \in L$$

Here, we naturally extend a function that works on data values to a function that works on data words.

1.2. Deterministic register automata

A deterministic register automaton consists of:

1. A finite set Σ of *labels*;
2. A finite set Q of *control states*;

3. A finite set R of *register names*;

4. An *initial control state* $q_0 \in Q$ and a set of *accepting control states* $F \subseteq Q$;

5. An equivariant (see below) *transition function*:

$$f \in Q \times (\mathbb{A} \cup \{\bot\})^R \times (\Sigma \times \mathbb{A}) \to Q \times (\mathbb{A} \cup \{\bot\})^R$$

A deterministic register automaton can be used to define a data language. Like most of the other automata, it keeps track of its internal configuration and updates it, processing the word letter by letter. The internal state of a register automaton consists of a control state and of a valuation of the registers ($Q \times (\mathbb{A} \cup \{\bot\})^R$). The initial state is the initial control state and the empty register valuation. The configuration is updated using the transition function f. A data word is accepted if the final control state belongs to the set of accepting states F.
Let us explain what it means for a transition function f to be equviariant. If we look at the function f as a special case of a relation,

$$f \subseteq Q \times (\mathbb{A} \cup \{\bot\})^R \times (\Sigma \times \mathbb{A}) \times Q \times (\mathbb{A} \cup \{\bot\})^R$$

we see that any atom permutation π can be extended to f:

$$(\pi(f))(x) = \pi^{-1}(f(\pi(x)))$$

We say that a set S is *equivariant* if every bijection $\pi : \mathbb{A} \to \mathbb{A}$ can be naturally extended to elements of S, and if for every such π we have $\pi(S) = S$. (We can now rewrite the definition of a data language from the previous section, saying that it is any equivariant subset of $(\Sigma \times \mathbb{A})^*$). Knowing this, we can directly apply the definition of equivariance to f: $\pi(f) = f$. In other words, this means that:

$$\underset{x,\pi}{\forall} f(x) = \pi^{-1}(f(\pi(x)))$$

To make the concept more concrete we also cite another equivalent definition from [2]. We still consider f as a special (functional) case of a relation. We say that a transition relation is *syntactically equivariant* if it can be defined by a finite boolean combination of constraints of the following types:

1. the control state in the source (respectively, target) configuration is $q \in Q$;

2. the label in the input letter is $a \in \Sigma$;

3. the data value is undefined in register $r \in R$ of the of the source configuration (respectively, target configuration);

4. the data value in the input letter equals the contents of register $r \in R$ in the source configuration;

5. the data value in register $r \in R$ of the source configuration (respectively, target configuration) equals the data value in register $s \in S$ of the source configuration (respectively, target configuration).

We say that a function f is syntactically equivariant if it defines a syntactically equivariant relation. By [2, Lemma 1.3], a relation is equivariant if and only if it is syntactically equivariant. To extend this proof to functions we only have to notice that the property of "being a function" is preserved under applying an atom permutation π to a relation.

1.2.1. Examples

Here are some languages that can be recognized using deterministic register automata:

1. The first and the last data values are equal
2. The first data value of the word appears somewhere else in the word
3. There are no more than 3 different data values in the word

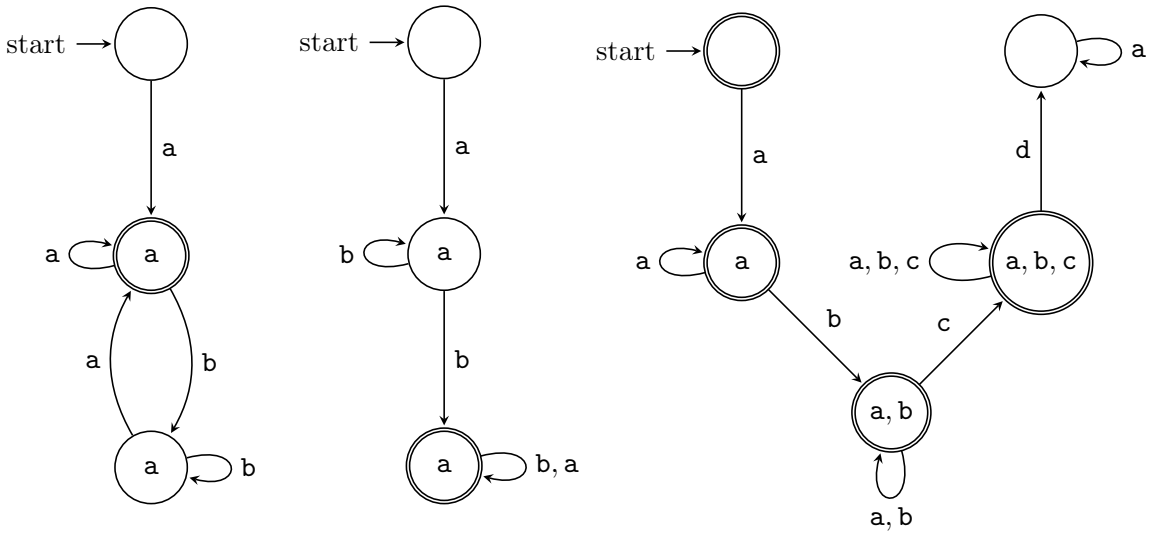


Figure 1.1: Register automata recognizing the example languages. In every state and in every transition values $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}$ represent different elements from $\mathbb{A}$. The Σ-labels have been skipped for clarity.

Here is an example of a language that cannot be recognized by any deterministic register automaton:

There is a data value that appears at least twice in the word.

Proof. Suppose that there is an automaton $\mathcal{A}$ that recognizes this language using k registers. After $\mathcal{A}$ reads a prefix of a word consisting of $k+1$ different data values, at least one data value from the prefix will not be present in the registers. Because of that, $\mathcal{A}$ has to go to the same control state when it sees a new data value and when it sees the "forgotten" data value. It means that $\mathcal{A}$ either has to either accept a word that does not belong to the language, or it has to reject a word from the language. So $\mathcal{A}$ cannot exist. □

Chapter 2

Orbit-finite sets and orbit-finite monoids

In this section, we define orbit-finite monoids – another model that can be used to define languages over infinite (but orbit-finite) alphabets.

2.1. Orbit-finite sets

We begin by defining *sets with atoms*.

2.1.1. Sets with atoms

In set theory "everything is a set", in sets with atoms everything is a set or an atom – an element from the fixed structure $\mathbb{A}$ (an infinite set equipped only with the equality relation). For the formal definition of a set with atoms we are going to cite Section 3.1 from [2]. The definition from [2] is actually more generic than the presented version, allowing for other choices of the structure of $\mathbb{A}$, but we only use an infinite structure equipped with the binary equality relation.

The cumulative hierarchy and its finitely supported elements

Consider first the *cumulative hierarchy* of sets with atoms, which is a hierarchy of sets indexed by ranks which are ordinal numbers. The empty set is the unique set of rank 0. For an ordinal number $\alpha > 0$, a set of rank α is any set whose elements are sets of rank smaller than α, or atoms. In other words, the cumulative hierarchy contains all possible objects built using the empty set, atoms and set brackets, including some objects that we do not want to consider. The definition of sets with atoms is obtained by restricting the cumulative hierarchy to sets that satisfy the "finite support condition", which models the idea of being definable using only the equality of the atoms and finitely many constants. The formal description is defined in terms of permutations

$$\pi : \mathbb{A} \to \mathbb{A}$$

If such permutation preserves a tuple of atoms: $\bar{a} = (a_1, ..., a_n)$, then it is called a $\bar{a}$-permutation. A permutation can be applied to a set in the cumulative hierarchy, by renaming its elements, (if the set happens to contain atoms), elements of its elements, and so on recursively. The result of applying a permutation π to a set X in the cumulative hierarchy is denoted by $\pi(X)$, and it is also a set in the cumulative hierarchy with the same rank.

Definition 2.1 (support)**.** *A tuple $\bar{a}$ of atoms is called a* support *of a set X in the cumulative hierarchy if $\pi(X) = X$ holds for every $\bar{a}$-permutation π. A set is called* finitely supported *if it has some finite support.*

An intuitive description of the support of a set is that the support consists of the atoms that are "hard-coded" into the definition of the set. Note that the order or repetition of atoms in the tuple is not relevant for the support, i.e. only the set of atoms that appear in the tuple matters. The support of a set with atoms is not unique e.g. supports are closed under adding atoms. A set with empty support is called *equivariant* (note that this definition of equivariance matches the definition from the previous chapter). Intuitively speaking, an equivariant set is one which can be defined without referring to any specific atoms.

Definition 2.2 (Set with atoms)**.** *A* set with atoms *is a set in the cumulative hierarchy which is hereditarily finitely supported, i.e. it is finitely supported, its elements are finitely supported, and so on.*

In many respects, sets with atoms behave like normal sets. For instance if X, Y are sets with atoms, then $X \times Y$, $X \cup Y$, X^* and the finite powerset of X are all sets with atoms. When talking about pairs (as in $X \times Y$ or X^*), we use the Kuratowski pair. Using pairs we can define sets with atoms which are binary relations, and using binary relations we can define sets with atoms which are functions. An arbitrary subset of a set with atoms might not be finitely supported, and therefore sets with atoms are not closed under taking arbitrary subsets, but only under taking finitely supported subsets.

2.1.2. Orbit finiteness

When talking about sets with atoms it makes little sense to talk about finite sets – they are not much different from the classical finite sets. Instead, we introduce an analog of finiteness for sets with atoms – orbit finiteness. In order to define it, we cite a section from [2].

Orbits and orbit finiteness

Intuitively speaking, a set is orbit finite if it has finitely many elements, up to atom permutations. The precise definition is given below. Let $\bar{a}$ be a tuple of atoms. Define an *$\bar{a}$-orbit* to be a set of form:

$$\{\pi(x) : \pi \text{ is an } \bar{a}\text{-permutation}\}$$

where x is an atom or set with atoms. It is easy to see that $\bar{a}$-orbits are either equal or disjoint, and therefore being in the same $\bar{a}$-orbit is an equivalence relation. A set with atoms is supported by $\bar{a}$ if and only if it is union, possibly infinite, of $\bar{a}$-orbits. The idea behind orbit finiteness is to consider sets which are finite unions of orbits. The number of $\bar{a}$-orbits depends on the choice of $\bar{a}$, but as the following theorem shows, whether or not the number of orbits is finite does not depend on the choice of support.

Theorem 2.1 ([2, Theorem 3.4])**.** *For every set with atoms X, the following conditions are equivalent:*

- *X is a finite union of some atom tuple $\bar{a}$ which supports X;*
- *X is a finite union of $\bar{a}$-orbits for every atom tuple $\bar{a}$ which supports X.*

A set with atoms which satisfies either of the above conditions is called *orbit-finite*.
To conclude this section we note that orbit-finite sets are closed under binary union, binary products, finitely supported subsets and images under finitely supported functions, but are not closed under taking the powerset [2, Lemma 3.8].

2.2. Languages over orbit finite alphabets and monoids

In this section, we consider languages over orbit-finite alphabets (we will focus on $\Sigma \times \mathbb{A}$ which is a special case of such alphabet). We demonstrate how to extend the concept of finite monoids to orbit finite-monoids, so that they can handle infinite (but orbit-finite) alphabets. Finally, we provide examples of languages that can and cannot be recognized this way and compare the expressive power of orbit-finite monoids and deterministic register automata. This section is based on [1].

2.2.1. Monoids with atoms

For this section we are going to cite [1], slightly adjusting the terminology:

Definition 2.3. *An orbit-finite monoid is a monoid whose carrier is orbit-finite, and whose binary concatenation operation is finitely supported. A morphism of orbit-finite monoids is a finitely supported function that is a monoid morphism.*

Free monoid

The following lemma shows that A^* (with regular word concatenation as the concatenation operation) deserves to be called free, for any orbit-finite set with atoms A:

Lemma 2.1 ([1, Lemma 3.2 on p. 8])**.** *Let M be an orbit-finite monoid and A an orbit-finite set. Every orbit-finite function with atoms $\alpha : A \to M$ can be uniquely extended to a monoid morphism $[a] : A^* \to M$ (the morphism is a valid set with atoms, but it may be orbit-infinite).*

Recognition

A monoid morphism $A^* \to M$ is said to *recognize* a language with atoms $L \subseteq A^*$ if there is a finitely-supported subset $F \subseteq M$ such that

$$\underset{w}{\forall} w \in L \iff \alpha(w) \in F$$

2.3. Examples

We present three languages over the alphabet $\mathbb{A}$. Two of them can be recognized using orbit-finite monoids and one of them cannot.

One of the letters of the word is equal to a fixed $a \in \mathbb{A}$

The monoid is a finite monoid $M = \{1, 0\}$ with the standard multiplication from $\mathbb{N}$ as the concatenation operation. The morphism is:

$$\alpha(x) = \begin{cases} 0 & x = a \\ 1 & x \neq a \end{cases}$$

And the accepting subset is $\{1\}$.

There are at most 3 different letters in the word

The monoid has 5 orbits:

- $\{\}$ – empty set (identity element), represents empty words,
- $\{x\}$ – one element set, represents words with only one letter,
- $\{x, y\}$ – two element set, represents words with only two letters,
- $\{x, y, z\}$ – three element set, represents words with only three letters,
- $\top$ – represents words with four or more letters.

The concatenation operation is defined as follows:

$$m_1 \cdot m_2 = \begin{cases} \top & m_1 = \top \text{ or } m_2 = \top \text{ or } |m1 \cup m2| > 3 \\ m1 \cup m2 & \text{otherwise} \end{cases}$$

The morphism is $\alpha(x) = \{x\}$ and the accepting subset is $M - \{\top\}$.

The first letter appears again

This time we are going to show that no monoid recognizes this language. First let us prove the following lemma:

Lemma 2.2. *For every orbit-finite set A there is a number k, such that for every $a \in A$ there is a support of a that contains no more than k atoms.*

Proof. Fix a support of A. First, let us prove this lemma for every S with only one orbit (with respect to this fixed support of A). Pick any $a \in S$ and any of its finite supports $\bar{s}$. Now from the definition of an orbit, for every $b \in S$, there is such π that $b = \pi(a)$. If so, then b is supported by $\pi(\bar{s})$ and this means that every element in S is supported by no more than $k = |\bar{s}|$ atoms. To extend this to A we note that there are only finitely many orbits in A, so there exists the maximum k among all the orbits of A. □

Now, suppose that there is an orbit-finite monoid M which recognizes the language 2.3. Take the value of k from Lemma 2.2 for this monoid, and consider

$$m = \alpha(a_1 a_2 a_3 ... a_{k+1})$$

The value m has a support $\bar{s}$ with at most k atoms, which means that at least one of the values a_i does not appear in s. Call this value a_j. There are infinitely many atoms in $\mathbb{A}$, so we can take another value e that is not equal to any of the a_i and is not present in $\bar{s}$. Let θ be a permutation that switches a_j with e and does not touch any of the other atoms. The function θ is a $\bar{s}$-permutation, so $m = \theta(m)$. We can assume that there is a support of α that contains none of the values a_i or e (this is because we could have fixed some finite support of α, and choose all of the values a_i and e outside of this support). This means that:

$$\alpha(a_1 a_2 a_3 ... a_j ... a_{k+1}) = \alpha(a_1 a_2 a_3 ... e ... a_{k+1})$$

If we multiply both sides by $\alpha(a_j)$ from the left, we obtain that:

$$\alpha(a_j a_1 a_2 a_3 ... a_j ... a_{k+1}) = \alpha(a_j a_1 a_2 a_3 ... e ... a_{k+1})$$

which means that $a_j a_1 a_2 a_3 ... a_j ... a_{k+1}$ and $a_j a_1 a_2 a_3 ... e ... a_{k+1}$ are either both accepted or both rejected. This is a contradiction because $a_j a_1 a_2 a_3 ... a_j ... a_{k+1}$ is in the language 2.3 and $a_j a_1 a_2 a_3 ... e ... a_{k+1}$ is not.

2.3.1. Comparison with deterministic register automata

On of the problems of comparing orbit-finite monoids with deterministic register automata are incompatible alphabets of the two formalisms – register automata can only work with alphabets of the form $\Sigma \times \mathbb{A}$, whereas monoids can work with any orbit-finite alphabet. To overcome this problem we might either limit ourselves to the alphabets of the form $\Sigma \times \mathbb{A}$, or consider *deterministic orbit-finite automata* – an extension of register automata ([2, Section 5.2]). It is not clear however how to use the second approach to generalize the automaton model described in the next section, so for the purpose of this thesis we choose the first solution.

The last example from the previous section (language 2.3) shows that

$$\text{orbit-finite monoids} \neq \text{deterministic register automata}$$

Actually, using the *representation theorem* ([2, Theorem 3.7]), one can construct a deterministic register automaton that recognizes the same language as every orbit-finite monoid M, obtaining that

$$\text{orbit-finite monoids} \subsetneq \text{deterministic register automata}$$

Chapter 3

An automaton model for orbit-finite monoids

In this section, we propose a version of register automaton and show that all the languages recognized using this model can also be recognized using orbit-finite monoids.

3.1. Motivation

One reason why deterministic register automata can recognize languages not recognizable by orbit-finite monoids is that they can compare a value stored in their register with infinitely many values that will appear later in the word. (This is clearly visible in the third example of from Section 2.3). As we will see this is the only reason. To prove this claim, we propose a new model of a register automaton which "loses" the value in a register every time it compares it with another value, and prove that as a tool of recognizing languages it is weaker than deterministic register automata.

3.2. Definition

Let us define a *deterministic automaton with disappearing registers*. This is another automaton model for recognizing equivariant languages over the alphabet $\Sigma \times \mathbb{A}$. Such an automaton is similar to the deterministic register automaton (as defined in Chapter 1). It consists of:

- a finite set Σ of *labels*;
- a finite set Q of *control states*;
- a finite set R of *register names*;
- an *initial state* $q_0 \in Q$;
- a *command function*:
$$f_c \in Q \times (\Sigma \cup \{\dashv\}) \to C$$
where C denotes the following set of commands:
  - `if` $r_a = r_b$ `then` q_1 `else` q_2 for every $r_a, r_b \in R$, $q_1, q_2 \in Q$,
  - `if current` $= r$ `then` q_1 `else` q_2 for every $r \in R$, $q_1, q_2 \in Q$,
  - `next letter and goto` q for every $q \in Q$,

– `fill` r `and goto` q for every $r \in R$, $q \in Q$,

– `accept` and `reject`;

When processing the input, the automaton keeps track its *configuration*, which consist of:

1. the position in the input word,
2. the current control state $q \in Q$,
3. the values kept in the registers (a function $R \to \mathbb{A} \cup \{\bot\}$).

The initial configuration consists of the first position in the word, the initial control state, and the empty value ($\bot$) in every register. When updating the configuration, the automaton executes the command chosen by f_c in the following way.

- For `if` $r_a = r_b$ `then` q_1 `else` q_2, the automaton checks if the value held in r_a is equal to the value held in r_b, updates the control state accordingly, and erases the values of r_a and r_b, inserting $\bot$ in their places;
- for `if current` $= r$ `then` q_1 `else` q_2, the automaton checks if the value held in the register r is equal to the currently seen data value, updates the control state accordingly, and erases the value of r, inserting $\bot$ in its place;
- for `next letter and goto` q the automaton tries to move its head to the next position in the word and sets the control state to q. If the automaton was already in the last position of the word, it proceeds to a special "end of word" position with label $\dashv$ and with the empty data value $\bot$;
- for `fill` r `and go to` q the automaton fills the register r with the currently seen data value, and updates the control state to q;
- for `accept` or `reject` the automaton accepts or rejects the input word accordingly and finishes the run.

Note that the only way for the automaton to move its head forward is to use the `move to next letter` command. Observe that all the registers whose values have been used are set to $\bot$ afterwards – this is why the registers are called "disappearing".

3.3. Examples

We now present two examples of languages that can be recognized using this kind of an automaton.

The same data value appears in some two consecutive positions

This language can be recognized using only one register. It will always store the value of the previously seen data value (or $\bot$ in the beginning). In every position, the automaton checks if the current data value is equal to the data value in the register. If it is, the automaton accepts the word. If it is not, the automaton saves the current letter in the register and proceeds to the next letter. The automaton rejects the input word, as soon as it reaches the end of the word.

There are at most 3 different letters in the word

(This example is also the solution to the riddle stated in the introduction).
This language is a little trickier. To recognize it we use 6 registers: $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{a}_3$, $\mathtt{b}_1$, $\mathtt{b}_2$, $\mathtt{c}_1$. The idea is that we only keep 3 values `a`, `b` and `c`, but we are keep `a` in 3 copies, `b` in 2 copies and `c` in 1 copy. Those values are going to be all the values the automaton has seen so far.

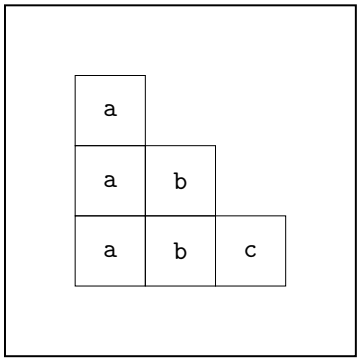
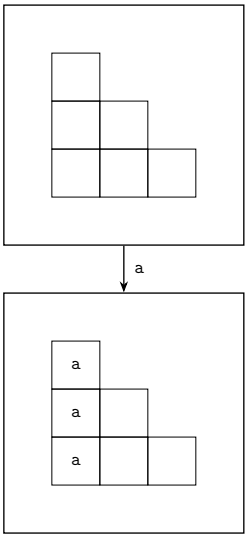


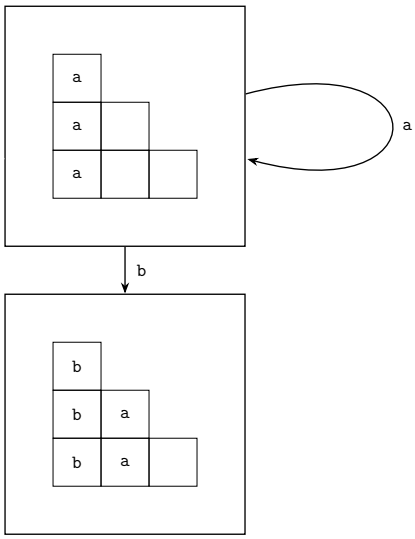
Figure 3.1: A sample configuration of the automaton

The automaton works as follows:

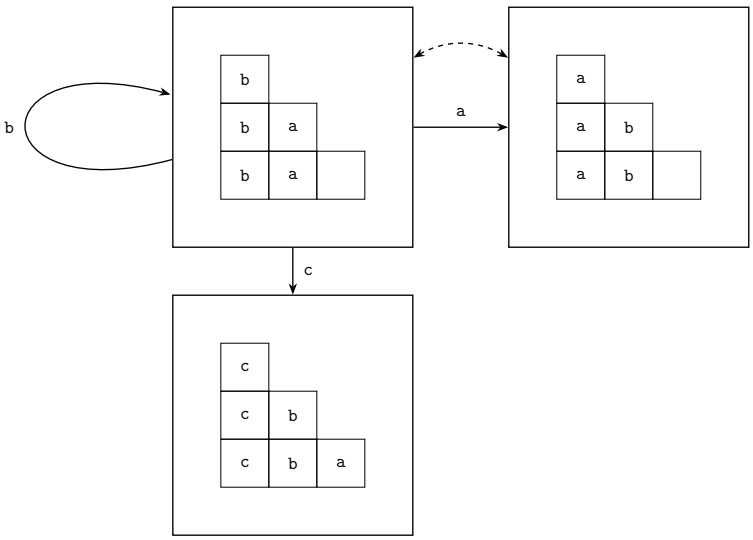
- when all of the registers are empty, the automaton fills up `a`-registers with the current data value and proceeds to the next letter;



- when only the value `a` is filled, the automaton checks if the current data value is equal to `a` (losing one of the copies of `a`):
  - if the current data value is equal to `a`, the automaton restores the lost copy of `a` using the current data value,
  - if not, the automaton moves two copies of `a` to `b`-registers, and fills up `a`-registers with the current data value;

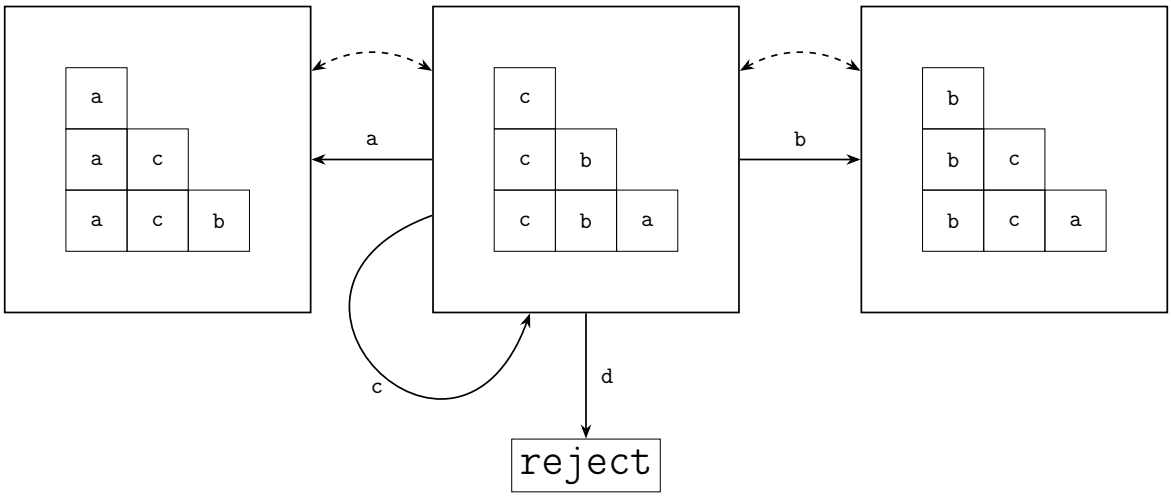


- when the values `a` and `b` are filled (but `c` is still empty), the automaton checks if the current data value is equal to `a`:
  - if it is, the automaton restores the used copy of `a`, using the current data value,
  - if it is not, the automaton checks if the current data value is equal to `b`:
    - if it is, the automaton moves the remaining two copies of `a` to `b`-registers and fills up `a`-registers with the current data value,
    - if it is not, the automaton moves the remaining one copy of `b` to the `c`-register, the remaining two copies `a` to `b`-registers and fills up `a`-registers with the current data value;



- when all of the values are filled, the automaton asks if the current data value is equal to `a`:
  - if it is, the automaton restores the copy of a `a`,
  - if it is not, the automaton asks if the current data value is equal to `b`:

* if it is, the automaton moves the remaining two copies of a to b-registers and fills up a-registers with the current value,
* if it is not, the automaton asks if the current data value is equal to c:
  - if it is, the automaton moves the remaining one copy of b to the c-register, the two remaining copies of a to b-registers, and fills up a-registers with the current data value,
  - if it is not, the automaton infers that it already has seen more than 3 different data values and rejects the input word.



The automaton accepts the word when it manages to proceed all of its letters without rejecting it.
One problem that still needs to be resolved are the "move" operations, that move a value from one register to the other. They are not explicitly allowed in the definition, but it turns out, that they do not change the expressive power of the automaton – we can get rid of them by keeping a permutation of register names in the control state (as there are only finitely many of such permutations). The complete schematic representation of the automaton can be found in Figure 3.2.

3.4. Connection with orbit-finite monoids

In this section, we prove the following result.

Theorem 3.1. *Every language recognized by a deterministic automaton with disappearing registers is also recognized by some orbit-finite monoid.*

Proof. We are given $\mathcal{A}$ – a deterministic automaton with k disappearing registers. Let us start the proof by defining the behaviour tree of $\mathcal{A}$ on any data (sub)word w. When we treat w as a subword, we assume that it does not end with the $\dashv$ mark, unless it has been explicitly included in w.

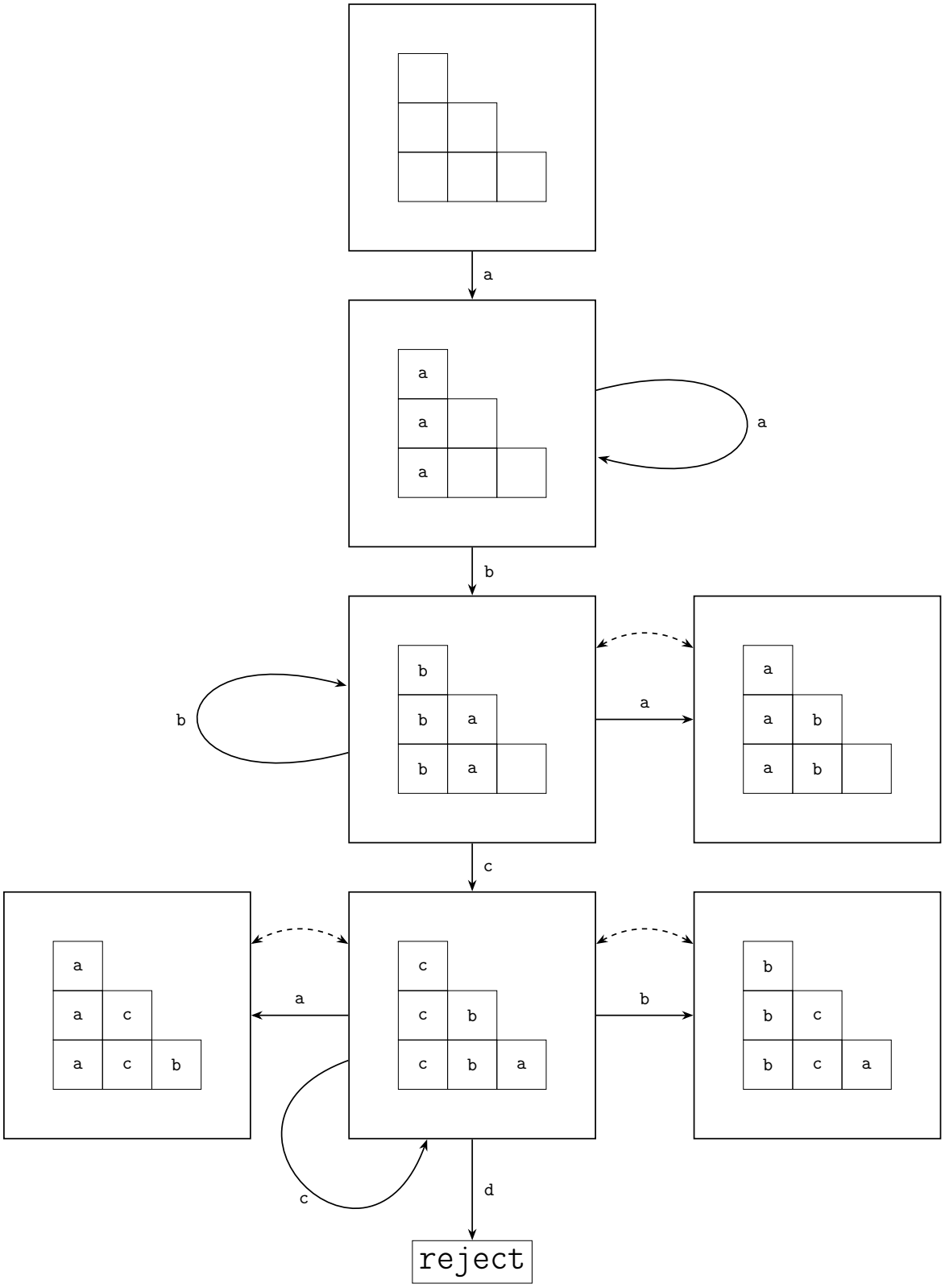


Figure 3.2: A complete schematic representation of the automaton described in 3.3. Solid edges represent (squashed) transitions of the automaton. Dashed edges represent equivalence of connected states.

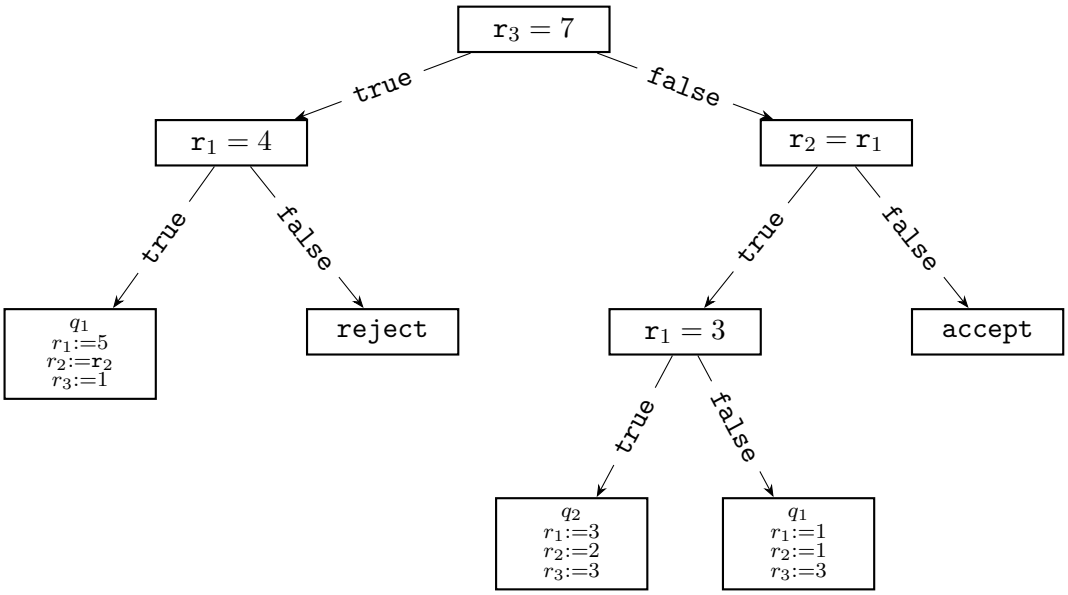


Figure 3.3: An example of a behaviour tree

If $\mathcal{A}$ enters w from the left side in the state $q \in Q$ with values r_1, r_2, ..., r_k in its registers, there are only orbit-finitely many possible outcomes:

1. $\mathcal{A}$ exits the (sub)word from the right side of w in the state q' with values r'_1, r'_2, ..., r'_k in its registers. Here *exits* means that the automaton was in the last position in w that was different then $\dashv$, and executed the `next letter and goto` q' command.

2. $\mathcal{A}$ accepts the word,

3. $\mathcal{A}$ rejects the word, possibly getting stuck in a loop.

If the word w and the state q are fixed, the outcome depends only on the the initial values of the registers. Let us define how to represent this kind of function. First, let us set the initial values of the registers to special "placeholder" values (outside of $\mathbb{A} \cup \{\bot\}$): $\mathtt{r}_1, \mathtt{r}_2, \cdots, \mathtt{r}_k$. The behaviour tree of $\mathcal{A}$ on the word w is defined as follows.

- As long as the automaton does not exit the word, reject it, accept it, loop, or executes `if` commands that require "placeholders" values, it runs normally. (It may use the `fill` command to override the real data values from w).

- If $\mathcal{A}$ executes an `if` command that requires the value of a placeholder, we consider both possible outcomes of the command. In this case, the behaviour tree has the placeholder-query in the root. Its left and right subtrees are the behaviour trees of the automaton after the two possible outcomes of the `if` command (q_a or q_b). Note that there are two possible types of a placeholder-query: $\mathtt{r}_i = \mathtt{r}_j$ and $\mathtt{r}_i = a$ for some $a \in A$.

- If the automaton accepts or rejects the word, the behaviour is an `accept`-leaf or a `reject`-leaf respectively. (Those leaves will be called terminal leaves).

- Finally, if the automaton leaves the word from the right in the state q', with register values equal to $r'_1, r'_2, ..., r'_k$ (some of which may be equal to regular elements of $\mathbb{A} \cup \{\bot\}$ and some of which may still be placeholders), the behaviour tree is a (non-terminal) leaf containing all this information.

An example of a behaviour tree can be found in the Figure 3.3. (In all of the pictures we use elements from $\mathbb{N}$ to represent elements from $\mathbb{A}$).

Observe that the depth of a behaviour tree is limited by the number of registers of $\mathcal{A}$ – the automaton overrides the value of each register it is using, so the path from the root to any of the leaves cannot pass twice through the same placeholder $\mathtt{r}_i$. Since the out-degree of every inner node is 2 and the depth of the tree is bounded (by k), the number of nodes in the tree is also bounded (by 2^{k+1}). This means that every tree stores at most ($z = k \cdot 2^{k+1}$) data values. This means that it can be stored as an element of $S \times (\{\bot\} \cup \mathbb{A})^z$ (where S is the finite set of all possible shapes of binary trees of depth not greater than k). This means that the set of all possible behaviour trees for $\mathcal{A}$ is orbit finite.

The entire behaviour of $\mathcal{A}$ can be stored as a function:

$$h : (\Sigma \times \mathbb{A})^* \to \mathtt{BehaviourTrees}^Q$$

where value $h(w)$ denotes the function from the starting control state to the behaviour tree on the word w.

Lemma 3.1. *The function h is compositional. This means that for every v, w, the value $h(vw)$ depends only on the values of $h(v)$ and $h(w)$.*

Proof. Say we want to compute $h(vw)(q)$, for some $q \in Q$. We first compute the tree $T_1 = h(v)(q)$. Then, we attach the tree $h(w)(q')$ to each of its non-terminal leaves – q' is the control state of $\mathcal{A}$ in which it exits v (this information is stored in the leaf). The resulting tree may contain some placeholder-query nodes that can already be resolved using the register valuation update found in the non-terminal leaves of T_1. After resolving all such queries the height of the resulting tree will be still at most k, because if a placeholder $\mathtt{r}_i$ is used in T_1 on the path from the root to the non-terminal leaf, then the non-terminal leaf has to store a $\mathbb{A} \cup \{\bot\}$ valuation for $\mathtt{r}_i$ since its placeholder value has already disappeared. (An example of this kind of composition can be found in Figures 3.4). □

This means that we can develop a monoid structure on

$$\mathtt{BehaviourTrees}^Q$$

in such a way that the h function is a monoid morphism.
Now, given the function h and a word w we want to decide whether $\mathcal{A}$ accepts w. For that we compute the tree

$$T = h(w \dashv)(q_{init})$$

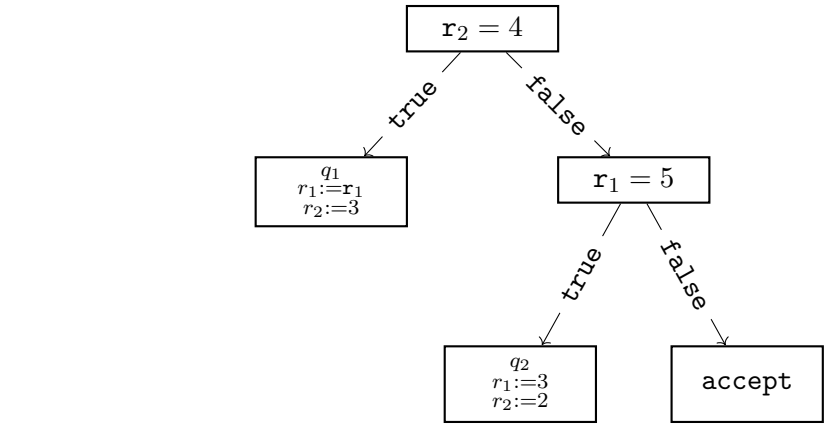
and substitute all the placeholder values with $\bot$. This narrows us down to only one leaf which has to be terminal (no $\mathcal{A}$ can exit a (sub)word that ends with a $\dashv$ mark). So we can simply read the leaf's label to say whether $\mathcal{A}$ accepts w. The function h is equivariant, so the language recognized by $\mathcal{A}$ can also be recognized by the orbit-finite monoid $\mathtt{BehaviourTrees}^Q$.

□

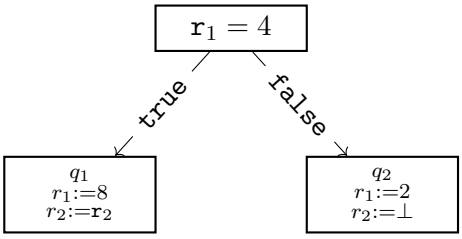
3.5. Comparison with other models

We have shown that

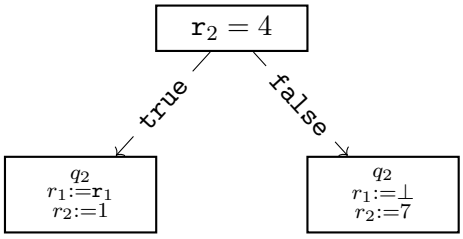
$$\text{deterministic automata with disappearing registers} \subseteq \text{orbit-finite monoids}$$



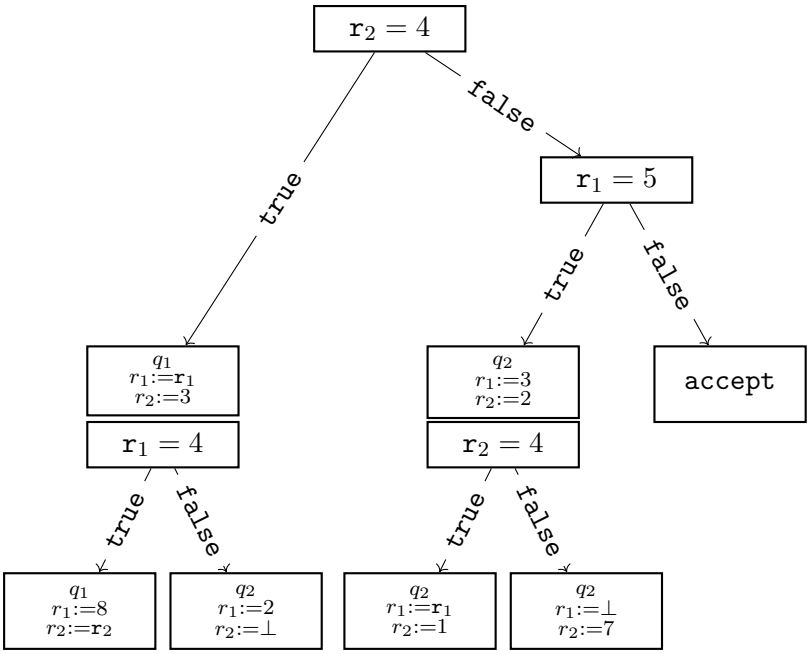
(a) Sample $B(v)(q_1)$ value



(b) Sample $B(w)(q_1)$ value

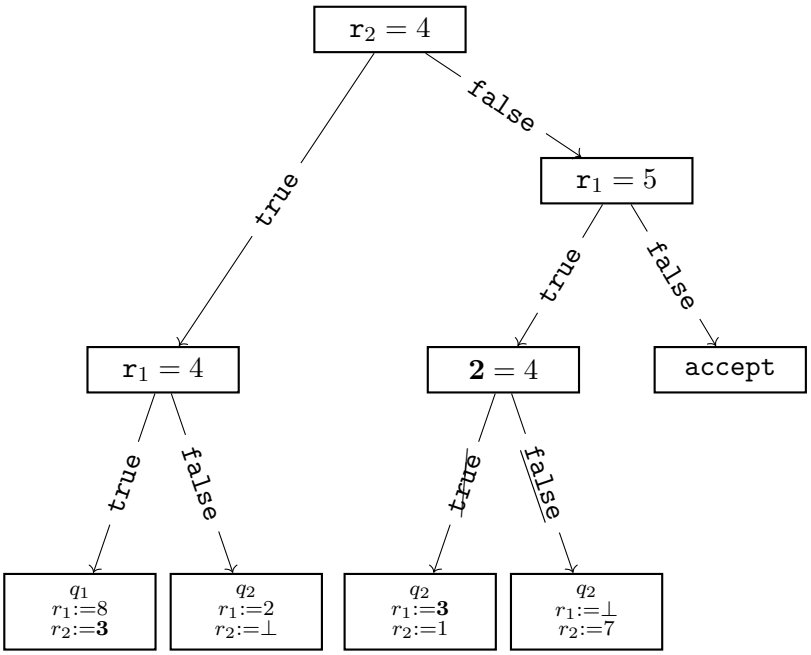


(c) Sample $B(w)(q_2)$ value

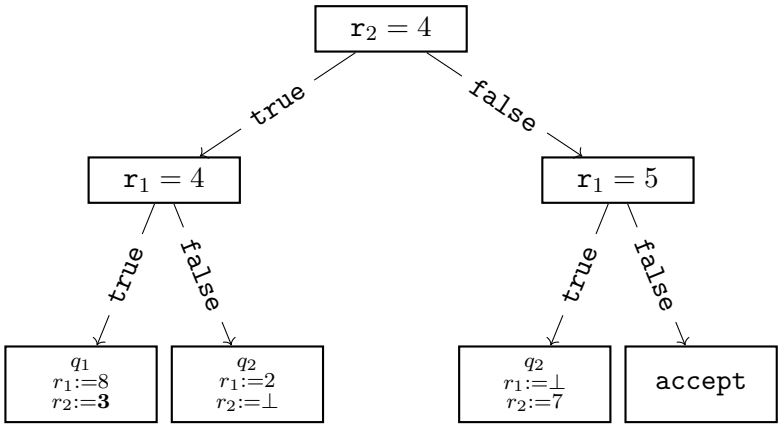


(d) Replacing non-terminal leaves in $B(v)(q_1)$ with matching trees from $B(w)$

Figure 3.4a: Example of behaviour trees composition



(e) Substituting placeholder values and resolving queries



(f) Final value of $B(vw)(q_1)$

Figure 3.4b: Example of behaviour trees composition

Since orbit-finite monoids are weaker than deterministic register automata, this proves that substituting regular registers with their "disappearing" version results in a weaker model.

3.6. Further work

The main question regarding deterministic automata with disappearing registers is

$$\text{deterministic automata with disappearing registers} \overset{?}{\supseteq} \text{orbit-finite monoids}$$

We would like to conjecture that this inclusion holds. The proof would, however, would require a way to store an abstract orbit-finite monoid M in a very concrete memory of a deterministic automaton with disappearing registers, which is a little problematic.
The two-way version of deterministic automata with disappearing registers seems to be very similar to the one-way version. We believe that all the languages recognized by two-way automata can also be recognized by orbit-finite monoids. The presented proof of Theorem 3.1 seems to work for this extension as well, but in order to complete it, one would additionally have to deal with cases in which the automaton starts to loop (in non-obvious ways).
If we combined the inclusion

$$\text{deterministic automata with disappearing registers} \overset{?}{\supseteq} \text{orbit-finite monoids}$$

together with the

$$\text{two-way deterministic automata with disappearing registers} \overset{?}{\subseteq} \text{orbit-finite monoids}$$

and with the rather obvious

$$\begin{array}{c}\text{two-way deterministic automata with disappearing registers}\\ \cup\!| \\ \text{one-way deterministic automata with disappearing registers}\end{array}$$

We would obtain that all the following formalisms recognize the same class of languages:

- Orbit-finite monoids
- One-way deterministic automata with disappearing registers
- Two-way deterministic automata with disappearing registers

Bibliography

[1] Mikołaj Bojańczyk. Nominal monoids. *Theory of Computing Systems*, 53(2):194–222, 2013.

[2] Mikołaj Bojanczyk. Slightly infinite sets. *A draft of a book available at https://www. mimuw. edu. pl/˜ bojan/paper/atom-book*, 2016.

[3] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.