Slightly Infinite Sets

Mikołaj Bojańczyk

April 15, 2017

Contents

I Data words and their automata 3

1 Register automata 4
- 1.1 Nondeterministic register automata. 4
- 1.2 Emptiness and universality for register automata 7
- 1.3 Alternating register automata 10
- 1.4 An emptiness algorithm using well quasi-orders 10
- 1.5 Most models of register automata are inequivalent 16

2 Two variable logic and data automata 18
- 2.1 Data automata 18
- 2.2 Recognising equality with successors 22
- 2.3 A logic recognised by data automata 26

II Sets with atoms 29

3 Sets with atoms and orbit-finiteness 30
- 3.1 The cumulative hierarchy and its finitely supported elements 30
- 3.2 Orbit-finiteness 33

4 Definable sets 39
- 4.1 Definable sets 39
- 4.2 Hereditarily orbit-finite sets 41
- 4.3 Definable equals hereditarily orbit-finite 42

5 Least supports 44
- 5.1 Least supports 44
- 5.2 A representation theorem. 46

6 **Homogeneous atoms** 48
6.1 Oligomorphism and quantifier elimination 50
6.2 The Fraïssé limit 52
6.3 Amalgamation 53

III Models of computation 59

7 **Computable functions on definable sets** 60
7.1 Effective structures 60
7.2 Computability of basic operations 62
7.3 Definable while programs 63
7.4 Example programs 66
7.5 An interpreter 70
7.6 Computational completeness of definable while programs 73

8 **Automata** 75
8.1 Orbit-finite automata 76
8.2 Pushdown automata and beyond 80

9 **Turing machines** 83
9.1 Computational completeness of alternating Turing machines 86
9.2 Determinism is weaker than nondeterminism 90

IV Solutions to the exercises 98

Part I
Data words and their automata

We begin with an investigation of concrete automata models for words over infinite alphabets. One goal of this part is to build up intuitions for the more abstract models that will be presented in the later parts.

1 Register automata

Define a *data word* over a finite alphabet Σ to be a word where every position has a label in $\Sigma \times \mathbb{A}$, where $\mathbb{A}$ is a fixed infinite set. The first coordinate is called the *label* and the second coordinate is called the *data value*. The idea is that we can test labels explicitly by asking questions like

> Does the second letter have $a \in \Sigma$ as its label?

but we can only test the data value for equality e.g. ask

> Do the third and fifth letters have the same data value?

In the later parts of this book, we will try to formalise what it means to only test data values for equality, but for now the intuitive understanding should be sufficient.

Example 1. By abuse of notation, we assume that a word over the alphabet $\mathbb{A}$ is also a data word, which uses no labels. Here are some examples of languages of data words, in all of these examples we use no labels:

1. the first data value is the same as the last data value
2. some data value appears twice
3. no data value appears twice
4. the first data value appears again
5. every three consecutive data values are pairwise distinct

□

We will introduce automata models for data words that capture the properties above. These models use registers to talk about data values.

1.1 Nondeterministic register automata.

The syntax of a *nondeterministic register automaton* consists of:

- a finite alphabet Σ of *labels*;
- a finite set Q of *control states*;
- a finite set R of *register names*;
- an *initial state* $q_0 \in Q$ and a set of *accepting states* $F \subseteq Q$;
- a *transition relation*

$$\delta \subseteq \underbrace{Q \times (\mathbb{A} \cup \{\perp\})^R}_{\text{configurations}} \times \underbrace{\Sigma \times \mathbb{A}}_{\text{input}} \times \underbrace{Q \times (\mathbb{A} \cup \{\perp\})^R}_{\text{configurations}} \tag{1}$$

subject to an equivariance condition described below.

The automaton is used to accept or reject data words where the alphabet is $\Sigma \times \mathbb{A}$. After processing part of the input, the automaton keeps track of a *configuration*, which is defined to be a control state plus a register valuation (i.e. a partial function from register names to data values). Initially, the configuration consists of the initial state and a completely undefined register valuation. The configuration is then updated according to the transition relation δ, and the automaton accepts if at the end of the word the control state belongs to the accepting set.

How to describe the transition relation? Since the space of configurations is infinite, the transition relation must satisfy some constraints, otherwise it cannot be represented in a finite way. We choose the following constraint, called *equivariance*: the transition relation can only compare data values with respect to equality. Equivariance can be formalised in two different ways below.

Semantic equivariance. A bijection $\pi : \mathbb{A} \to \mathbb{A}$ on the data values can be applied to configurations in the natural way, and therefore also to triples in the transition relation δ (the states and undefined values are not affected, only the data values). We say that δ is *semantically equivariant* if

$$\pi(t) \in \delta \qquad \text{for every } t \in \delta \text{ and every bijection } \pi : \mathbb{A} \to \mathbb{A}.$$

The advantage of semantic equivariance is that the definition is short, and it will be easy to generalise to other models, like alternating automata or pushdown automata. The disadvantage is that it is not clear how to represent a semantically equivariant transition relation, e.g. for the input of a nonemptiness algorithm. The converse situation holds for syntactic equivariance, as presented below.

Syntactic equivariance. We say that δ 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 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 (respectively, target configuration);
5. the data value in register $r \in R$ of the source configuration (respectively, target configuration) equals the data value in register $s \in R$ of the source configuration (respectively, target configuration).

Lemma 1.1 *Semantics and syntactic equivariance are the same.*

Proof
It is not difficult to see that semantically equivariant subsets of the set (1) are closed

under boolean combinations. Since the bijections of data values do not affect satisfaction of the constraints 1-5 used in the definition of syntactic equivariance, it follows that syntactic equivariance implies semantic equivariance.

We now show that semantic implies syntactic. Define an *orbit of transitions* to be a subset of the set (1) which is semantically equivariant and which is minimal for that property with respect to inclusion.

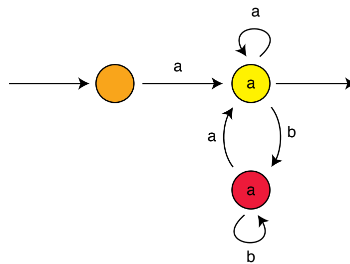
Claim 1.1.1 *Every orbit of transitions is syntactically equivariant.*

Proof (of Claim)
Because an orbit of transitions is uniquely defined by its states, which registers are undefined, and what is the equality type of the tuple of data values in the defined registers. All of this information can be expressed using the constraints 1-5 in the definition of syntactic equivariance. □

Once the number of registers and states is fixed, there are finitely many possible constraints as in the definition of syntactic equivariance. Boolean combinations make the number of possibilities grow, but it remains finite. Therefore, thanks to the above claim, there are finitely many possible orbits of transitions. Finally, every semantically equivariant relation is easily seen to be the union of the orbits contained in it. This union is finite, and each part of the union is syntactically equivariant, and thus the result follows. □

This completes the definition of nondeterministic register automata: the transition relation is required to be equivariant in either of the two equivalent senses defined above. The transition relation is called *deterministic* if the source configuration and the input letter determine uniquely the target configuration.

Example 2. Here is a deterministic register automaton which recognises language 1 from Example 1, i.e. the words in $\mathbb{A}^*$ where the first and last data values are equal. The automaton stores the first data value in its register, and then toggles between accepting or rejecting states depending on whether the input agrees with the register. Here is a picture:



The above picture should be interpreted as follows. There are three states, standing for the three coloured circles, with initial and final states depicted by the dangling arrow. Since there is one register, a configuration consists of a state and a possibly empty data value. Such configurations can be found in the picture above. For

every pair of distinct atoms $a \neq b$, we add a transition from the above picture to the automaton. Note how every arrow in the picture corresponds to an orbit of transitions.

The method of drawing above has its limitations. For example, if we wanted to add a transition that would involve the yellow state with an undefined register, we would need to draw a separate instance of the yellow state. □

Exercise 1. Show that deterministic register automata can recognise languages 4 and 5 from Example 1.

Exercise 2. For languages of data words one can also define the Myhill-Nerode relation, as used in minimisation of deterministic automata. Show a language of data words where every deterministic register automaton distinguishes (by its configuration) some two words which are Myhill-Nerode equivalent.

Exercise 3. Show there is a language of data words, for which there are at least two nonisomorphic deterministic register automata with a minimal number of registers and states (lexicographically).

Exercise 4. Show that a nondeterministic register automaton can recognise language 2 from Example 1, but a deterministic one cannot.

Exercise 5. Call a nondeterministic register automaton *guessing* if there exists a transition $t \in \delta$ such that some data value in the target register valuation appears neither in the source register valuation nor in the input. Give an example of language that needs guessing to be recognised.

A corollary of the above two exercises is that:

$$\text{deterministic} \subsetneq \text{nondeterministic without guessing} \subsetneq \text{nondeterministic}$$

Furthermore, the two nondeterministic variants are not closed under complementation, and the first two models are not closed under reverse.

Exercise 6. Call a nondeterministic register automaton *weakly guessing* if whenever the transition reading the i-th letter loads a data value d into some register r, then d appears in some position $j \geq i$ such that the transitions reading letters $\{i, \ldots, j\}$ do not remove data value d from register r. Show that for every nondeterministic register automaton there is a weakly guessing one which accepts the same words.

1.2 Emptiness and universality for register automata

In this section we discuss two standard decision problems: emptiness (does the automaton accept at least one input word) and universality (does the automaton accept all input words). When talking about decidability, we assume that the transition function is represented according to the syntactic equivariance condition.

Theorem 1.2 *Emptiness is decidable for nondeterministic register automata.*

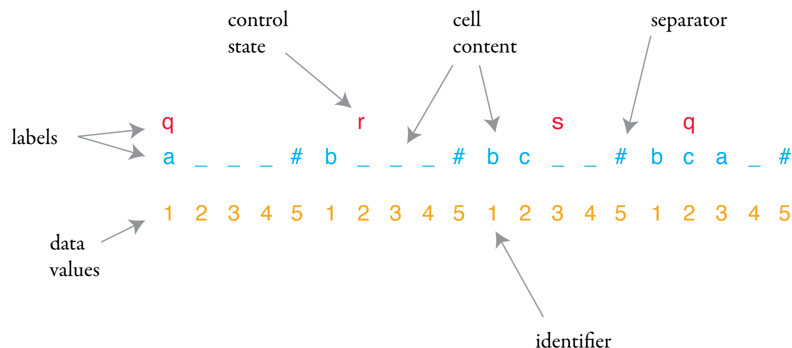
Proof
This proof just sketches the decidability argument, the complexity is discussed in Exercise 7. Define an *orbit of configurations* to be a set of configurations that is closed under bijections of data values. As in Lemma 1.1, an orbit of configurations can be defined by saying what is the states, which are the defined registers, and what is the equality type on the data values stored in the defined registers. Such a description takes finite space to store, and there are finitely many possible descriptions. The key observation that being in the same orbit of configurations is a congruence with respect to transitions, i.e. if two configurations are in the same orbit, then both are reachable or both are unreachable. The algorithm for nonemptiness computes the orbits of reachable configurations. Initially, we have the equality type of the unique initial configuration, which can be easily computed. If we have the equality type of some configuration, we can easily compute the equality types of all configurations reachable from it in one step; thus finishing the description of the algorithm. □

Exercise 7. The complexity of the emptiness problem depends on how the size $|\mathcal{A}|$ of the input automaton is measured. Show that that emptiness is:

- PSPACE-complete if $|\mathcal{A}|$ is the number of states and registers;
- NP-complete if $|\mathcal{A}|$ is the number of reachable orbits of configurations;
- polynomial time if $|\mathcal{A}|$ is the number of orbits of transitions.

Theorem 1.3 *Universality is undecidable for nondeterministic register automata.*

Proof
We reduce from the halting problem. Suppose that we have a Turing machine which is an instance of the halting problem. We encode a run of a Turing machine as a data word according to the following picture:



Each letter encodes a single cell in a single configuration. The word represents a sequence of configurations, padded with blanks so that they all have the same length, and separated by a letter #. The labels are used to store the contents of the cell (blue), plus the control state (red) of the head if the head happens to be over that cell. Finally, each cell gets a unique identifier, a data value (orange). The following claim shows that the halting problem reduces to universality of nondeterministic register automata, thus proving the theorem.

Claim 1.3.1 *There is a nondeterministic register automaton which accepts a data word if and only if it is* not *an encoding of an accepting run of the Turing machine.*

Proof
To prove the claim, we list the mistakes that can happen in a word that does not encode an accepting run of a Turing machine:

1. The data values identifying the cells are chosen wrong. This means that:

    (a) the separator # is used with more than one data value; or

    (b) there exist positions x, y with the same data value such that the successor positions $x + 1$ and $y + 1$ have distinct data values.

    The first condition can be tested using one register, the second condition using two registers.

2. There is a mistake between two consecutive configurations. Assuming the identifiers are chosen correctly, this can be tested using only one register, to tell which cells correspond to which ones in the following configuration.

3. The first configuration is not initial, or the last configuration is not accepting. For this, no registers are needed.

□ □

Exercise 8. The undecidability proof in Theorem 1.3 used automata with two registers but no guessing (as in Exercise 5). Show that, in the presence of guessing, universality remains undecidable even with one register.

Exercise 9. To express properties of data words, we can use first-order logic, where the quantifiers range over positions, and there are predicates for the order on positions, equality of data values, and the labels. For example, the following formula that every position with label a is followed by a position with label b and the same data value:

$$\underbrace{\forall x}_{\text{for every position } x} \ \Big(\underbrace{a(x)}_{x \text{ has label } a} \Rightarrow \underbrace{\exists y}_{\text{exists a position } y} \ (\underbrace{y > x}_{y \text{ is after } x} \wedge \underbrace{y \sim x}_{x \text{ and } y \text{ have the same data value}} \wedge \underbrace{b(y)}_{y \text{ has label } b})\Big)$$

Show that satisfiability is undecidable for this logic, i.e. one cannot decide if a given formula is true in some data word.

1.3 Alternating register automata

In a nondeterministic automaton, the transition is chosen nondeterministically in favour of acceptance, i.e. for acceptance it suffices that there is at least one choice of transitions that gives an accepting run. An alternating automaton is a generalisation of a nondeterministic automaton, where the syntax specifies which states chose transitions in favour of acceptance, and which states chose transitions against acceptance. The main result of this section is that emptiness is decidable for a restricted version of alternating register automata.

Alternating register automata The syntax of an *alternating register automaton* is defined the same way as for a nondeterministic register automaton, except that there is an additional partition of the states Q into two parts, called *existential* and *universal*.

We define the semantics of the automaton using *bags*, where a bag is defined to be a set (not necessarily finite) of configurations. For every input letter a (consisting of a label and a data value), we define a binary relation $\xrightarrow{a}$ on bags, such that $C \xrightarrow{a} D$ holds if:

- for every configuration $c \in C$ with an existential state, the bag D contains some configuration d such that (c, a, d) is a transition;
- for every configuration $c \in C$ with a universal state, the bag D contains all configurations d such that (c, a, d) is a transition.

A data word $a_1 \cdots a_n$ is accepted if there exists a run

$$\text{initial bag } = C_0 \xrightarrow{a_1} C_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} C_n \in \text{accepting bags}$$

where the initial bag is defined to be the singleton of the initial configuration, and an accepting bag is defined to be any bag that contains only configurations with accepting states. We define $\to$ to be the union of all relations $\xrightarrow{a}$, ranging over all letters a. In terms of this notation, an alternating automaton is nonempty if an accepting bag is reachable from the initial bag via a finite number of steps of $\to$.

Exercise 10. Show that languages recognised by alternating register automata are closed under complement.

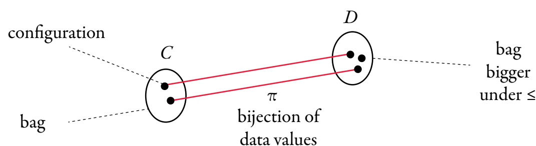
1.4 An emptiness algorithm using well quasi-orders

Nondeterministic register automata are the special case of alternating register automata where all states are existential. By Exercise 10, the emptiness and universality problems for alternating register automata are essentially the same problem, which is undecidable by Theorem 1.3. Furthermore, by the remarks in Exercise 8, this undecidability is true already for two registers and no guessing, or even one register with guessing. That is the limit of undecidability:

Theorem 1.4 *Emptiness is decidable for one register alternating automata without guessing.*

The rest of this section is devoted to proving the above theorem. The set of nonempty alternating automata is semi-decidable, i.e. there is an algorithm (guess a word and run the automaton on it) which terminates if and only if the input automaton is nonempty. Therefore, in order to prove decidability it suffices to show that the set of empty alternating automata is also semi-decidable. The rest of this section is devoted to designing an algorithm which inputs an automaton and terminates if and only if the input automaton is empty. In other words, we are searching for a finite and computable witness of emptiness.

As in the definition of semantic equivariance from Section 1, bijections of data values can be applied to configurations and to bags of configurations. The following order on bags is the key to our proof: we write $C \leq D$ if there is some bijection of the data values π such that $C \subseteq \pi(D)$. Here is a picture:



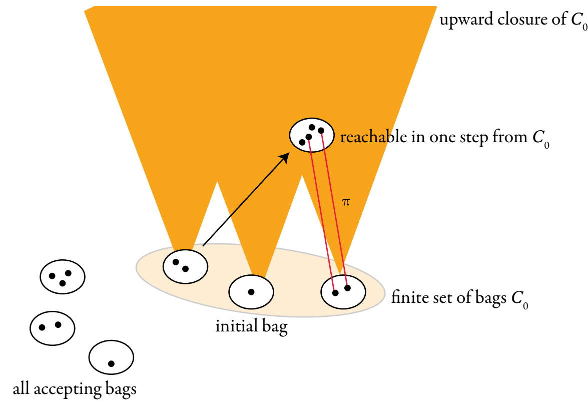
This is easily seen to be a quasi-ordering, i.e. a transitive and reflexive relation. Call a set of bags *upward closed* if it is upward closed with respect to this quasi-order. The *upward closure* of a set of bags is the least upward closed set of bags that contains it.

The following lemma gives the semi-decidability of emptiness.

Lemma 1.5 (Finite Emptiness Witness Lemma) *An alternating one register automaton without guessing accepts no words if and only if there exists some finite set of bags $\mathcal{C}_0$ which is a witness in the following sense:*

1. *$\mathcal{C}_0$ contains no accepting bags;*
2. *the initial bag is in $\mathcal{C}_0$;*
3. *if $C \to D$ and $C \in \mathcal{C}_0$, then D is in the upward closure of $\mathcal{C}_0$.*

Here is a picture of the witness from the above lemma:



The idea is that the orange area, i.e. the upward closure of C_0, is a trap in the sense that no transition can leave the orange area. It is straightforward to see that existence of a witness is semi-decidable: guess the set C_0 and check the three conditions. For the third condition, it is useful that there is no guessing, since the relation $\to$ has finite outdegree without guessing (with guessing, the condition would still be verifiable). Therefore, the Finite Emptiness Witness Lemma completes the proof of Theorem 1.4. It remains to prove the lemma, which we do in the rest of this section. Fix an alternating one register automaton.

There are two key properties of the relation $\leq$ which make the Finite Emptiness Witness Lemma true: it is a well quasi-order and it is compatible with transition relation $\to$ on bags. These are explained and proved below.

Well quasi-order. We say that a quasi-order is a *well quasi-order* if it is well-founded (no infinite strictly decreasing chains) and has no infinite antichains. The technique of well quasi-orders, as used in the following proof, is one of the most common methods of proving decidable properties for systems with infinitely many configurations.

Exercise 11. Show that a quasi-order is a well-quasi-order if and only if every infinite sequence contains a monotone subsequence, i.e. one where $i \leq j$ implies $x_i \leq x_j$

Exercise 12. Show that for every dimension $d \in \{1, 2, \ldots\}$, the set $\mathbb{N}^d$ is a well quasi-order with respect to the coordinatewise ordering.

Lemma 1.6 *The relation $\leq$ on finite bags is a well quasi-order.*

Proof
It is clear that the relation is well-founded, since a strict decrease on finite bags implies a strict decrease in the cardinality. It remains to show that there is no infinite antichain. Define the *profile* of a bag C to be the following information:

- for each state $q \in Q$, does the bag contain a configuration with state q and an undefined register;

- for each nonempty set of states $P \subseteq Q$, how many data values d satisfy the following property: for every state $p \in Q$, the bag contains a configuration with state p and data value d if and only if $p \in P$.

A profile can be seen as a binary vector indexed by Q plus a vector of natural numbers indexed by nonempty subsets of Q. The profile mapping takes incomparable pairs (of bags under $\leq$) to incomparable pairs (of profiles seen as vectors ordered coordinatewise). Therefore, the profile mapping takes infinite antichains to infinite antichains. Since there are no infinite anitichains in the latter space by Exercise 12, it follows that there are no infinite antichains in the former space. □

In our proof, we will be using the following corollary of being a well quasi-order.

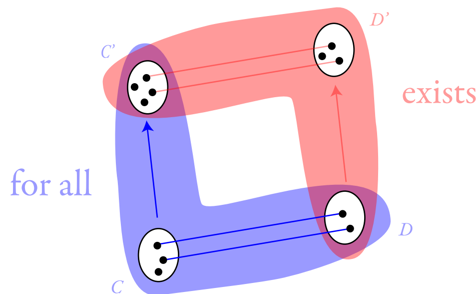
Lemma 1.7 *Every upward closed set of bags is the upward closure of a finite set of bags.*

Proof
By well-foundedness, every upward closed set is the upward closure of its minimal elements. The minimal elements form an antichain, and hence there can only be finitely many of them (up to renamings). □

Compatibility. The following lemma shows that the order $\leq$ on bags is compatible with the transition relation $\rightarrow$ on bags in the sense that making the source bag smaller makes doing transitions easier.

Lemma 1.8 *The relation $\leq$ on bags is compatible with $\rightarrow$ in following sense: for every transition $C \rightarrow C'$ and every $D \leq C$ there exists some $D' \leq C'$ with $D \rightarrow D'$.*

Proof
Here is the picture of compatibility:



Because $\rightarrow$ is closed under permutations of data values, and also closed under making the first argument a smaller bag. □

Using compatibility, we can prove the right-to-left implication in the Finite Emptiness Witness lemma. Suppose then that $\mathcal{C}_0$ is a finite set of bags which satisfies the three conditions in the lemma. Let $\mathcal{C}$ be the upward closure of $\mathcal{C}_0$. Since accepting bags are downward closed, condition 1 in the lemma implies that $\mathcal{C}$ contains no

accepting bags. Condition 2 implies that $\mathcal{C}$ contains the initial bag. Finally, compatibility ensures that if $C \in \mathcal{C}$ and $C \to C'$, then also $C' \in \mathcal{C}$. In other words, $\mathcal{C}$ is an invariant which witnesses that the initial bag cannot reach any accepting bag.

Finding the finite emptiness witness. To complete the proof of the Finite Emptiness Witness lemma, we need to prove the left-to-right implication, i.e. find the finite witness $\mathcal{C}_0$ in any alternating automaton that accepts no words.

Define $\mathcal{R}$ to be the set of bags which can reach some accepting bag in a finite number of steps in the relation $\to$.

Lemma 1.9 *$\mathcal{R}$ is downward closed.*

Proof
Take any finite path

$$C_n \to C_{n-1} \to \cdots \to C_1 \in \text{accepting bags.}$$

To prove the lemma, we prove by induction on n that if $D \leq C_n$, then $D \in \mathcal{R}$. The induction base is the fact that the set of accepting bags is downward closed. For the induction step, we use the compatibility established in Lemma 1.8. □

Stated differently, the above lemma says that the complement of $\mathcal{R}$ is upward closed. Apply Lemma 1.7 to this complement, yielding a finite set of bags $\mathcal{C}_0$. We will prove that $\mathcal{C}_0$ is a witness in the sense of the Finite Emptiness Witness Lemma. By definition of $\mathcal{C}_0$, every bag C satisfies

$$C \text{ cannot reach an accepting bag} \quad \text{iff} \quad C \text{ is in the upward closure of } \mathcal{C}_0.$$

To prove that $\mathcal{C}_0$ is a witness, let us check the three conditions from the Finite Emptiness Witness Lemma. Clearly there can be no accepting bags in $\mathcal{C}_0$, because an accepting bag can reach an accepting bag in zero steps. By assumption that the automaton is empty, the initial bag cannot reach an accepting bag, and hence the initial bag is in the upward closure of $\mathcal{C}_0$. Only the empty bag is smaller than the initial bag, and the empty bag is accepting, hence the initial bag must actually be in $\mathcal{C}_0$, and not only in its upward closure. Finally, let us prove the third condition. The upward closure of $\mathcal{C}_0$ is closed under taking a step of $\to$, since if C cannot reach an accepting bag, then the same is true for anything reachable from C. This implies the third condition. This completes the proof of the Finite Emptiness Witness Lemma and of Theorem 1.4.

The general technique. Using the same proof, we obtain the following generalisation of Theorem 1.4.

Theorem 1.10 *The following problem is decidable.*

- **Input**.
  - *A directed graph where every node has finite outdegree;*

  – *A well quasi-order $\leq$ on vertices which is compatible with the edge relation;*
  – *A source vertex plus a set of target vertices that is downward closed.*

  The graph is represented by algorithms for: enumerating the vertices, testing membership in the target set, testing the well quasi-order, and computing the neighbour list of a given vertex.

- **Question.** *Is there a path from the source to one of the targets?*

A temporal logic for data words. One register alternating automata can be dressed up in the syntax of a temporal logic. The idea is to add one register to linear temporal logic LTL. We do not give the detailed syntax and semantics, only some examples. We are extending LTL, so we can write a formula

$$a \text{ until } b,$$

which is true in a (data) word if a prefix of the label sequence is in a^*b. Instead of a, b we could have used simpler formulas, and Boolean combinations are allowed. There is also an operator to access the next position, so e.g. the formula

$$\underbrace{(a \vee \neg a)}_{\top} \text{ until } (a \wedge \text{ next } a)$$

says that there exist two consecutive positions with label a. We use finally φ as syntactic sugar for $\top$ until φ. If we only use the operators until and next, then we have exactly the logic LTL, which is insensitive to the data values. To access the data values, we can add an operator store which stores the current data value, and a formula same which is true whenever the current value is equal to the stored one. For example, the formula

$$\text{store}\big(\text{next } \neg\big(\text{finally same}\big)\big)$$

says that the first data value does not repeat, i.e. after storing it one cannot find the same one again. In principle we could have several different registers for storing data values, but if we want to translate the logic to one register alternating automata, then only one register is allowed (and hence there is no need to give it a name). The register can be reused, e.g. the following formula says that whenever the first data value of the word is used, then the next two positions have distinct data values:

$$\text{store}\big(\text{next } \neg\big(\text{finally}(\text{same} \wedge \text{next } (\text{load}(\text{next same})))\big)\big)$$

Exercise 13. For a possibly infinite alphabet Σ, define the Higman ordering on Σ^* to be the relation of not necessarily connected substrings. Show that this is a well quasi-ordering.

Exercise 14. Suppose that the data values are equipped with a total order. Show that emptiness remains decidable for one register alternating automata without guessing, even when the machine can use the order to compare the register with the current data value.

Exercise 15. For a Turing machine with one tape, define a *gain* to be the process of taking a configuration and inserting nondeterministically one new cell in some position not below the head, with any label from the work alphabet. Define a *gainy computation step* of a Turing machine to be a finite (possibly zero) number of gains followed by a normal step of computation. Show that the halting problem is decidable for Turing machines with semantics defined using gainy computation steps.

Exercise 16. Show that there is an infinite antichain for the following order on $\mathbb{A}^*$:

$$w \leq v \qquad \text{if} \qquad w \text{ is Higman smaller or equal to } \pi(v) \text{ for some permutation of } \mathbb{A}.$$

Exercise 17. Show that there is a language $L \subseteq \mathbb{A}^*$ that is upward closed under the Higman order, but is not recognised by a nondeterministic register automaton.

1.5 Most models of register automata are inequivalent

The goal of this section is to collect exercises which show that with one exception, the only inclusions between models of register automata are the ones that trivially follow from the definitions. To have a richer landscape, we also consider the two-way variant of register automata, where the head of the automaton can move both ways, with the input being extended by markers on both sides. For the purpose of this section, we assume that all models allow ϵ-transitions.

Exercise 18. Show that a deterministic two-way register automaton can recognise

$$\{a_1 \cdots a_n : a_1, \ldots, a_n \text{ are distinct and } n \text{ is a prime number}\}.$$

Exercise 19. Show that for every nondeterministic two-way register automaton $\mathcal{A}$ with one register, if the labels are Σ, then the following language is regular:

$$\{b_1 \cdots b_n \in \Sigma^* \quad : \quad \mathcal{A} \text{ accepts } (b_1, a_1) \cdots (b_n, a_n) \text{ for some distinct data values } a_1, \ldots, a_n \in \mathbb{A}\}.$$

Exercise 20. Find a language that is recognised by an alternating register automaton with guessing, but not by any alternating register automaton without guessing.

Exercise 21. Show that every two-way nondeterministic register automaton can be simulated by an alternating register automaton (with guessing and ϵ-transitions.)

We now present a series of exercises with a more systematic study of the following models of automata: one-way deterministic and nondeterministic, two-way deterministic and nondeterministic, as well as one-way alternating with or without guessing. We assume ϵ-transitions are allowed in all models. The picture with these six models is in Figure 1. The picture shows the obvious containments which follow from the syntax as well as the less obvious containment from Exercise 20. In the solutions to the following exercises, one is allowed to give answers conditional on open problems in complexity theory such as P = NP.

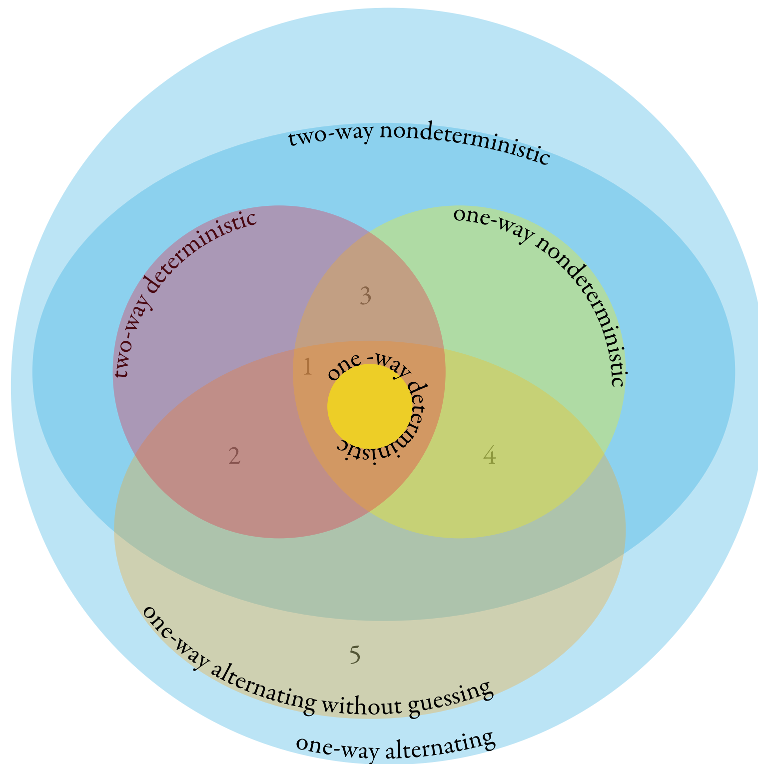


Figure 1: Six classes of register automata and their combinations. Point 1 is the language: "last letter appears only once", while point 2 is the language "all letters are distinct". The remaining points 3, 4, 5 are Exercises 22-24, while Exercise 25 sums up the results by saying that all combinations are possible.

Exercise 22. Show a language that witnesses point 3 in Figure 1.

Exercise 23. Show a language that witnesses point 4 in Figure 1.

Exercise 24. Show a language that witnesses point 5 in Figure 1.

Exercise 25. Show that all coloured areas in Figure 1 contain languages.

Bibliographic notes for Section 1. Register automata where introduced in [35], under the name of *finite memory automata*, together with a decidability proof for the emptiness problems in the deterministic and nondeterministic one-way cases (Theorem 1.2 in this text). The presentation using syntactic and semantics equivariance, in particular Lemma 1.1, is essentially due to [8, 12]. An in-depth study of various kinds of register automata can be found in [48], including undecidability of universality of nondeterministic one-way automata (Theorem 1.3). The non-equivalence results summarised in Figure 1 are originally found in [35, 48] and an unpublished Masters' thesis in Polish [54]. For a survey on automata and logic for infinite alphabets, see also [52].

Decidability of emptiness for alternating one-way register automata with one register, Theorem 1.4 in this text, was first shown in [25]. A tree extension of the result can be found in [34]. The well quasi-order technique was independently introduced in [2] and [28], and is currently known as the technique of *well-structured transition systems*.

2 Two variable logic and data automata

In this section we define an automaton model for data words that does not use registers, called *data automata*. There are three reasons to discuss data automata: emptiness of data automata are a pretext to discuss an important decidability result about vector addition systems; there is a nontrivial result that data automata generalise nondeterministic register automata; and data automata have a natural correspondence to two variable logic over data words.

2.1 Data automata

In the definition of a data automaton, we use a nondeterministic transducer over words without data, so we begin by describing this transducer.

Letter-to-letter transductions. Consider a nondeterministic finite automaton where every transition is labelled by a letter of an output alphabet. We view this automaton as a device which inputs a word, and outputs all possible words that label accepting runs (in other words, the semantics of such an automaton is a binary relation on words, which only contains pairs of words with equal lengths). A relation

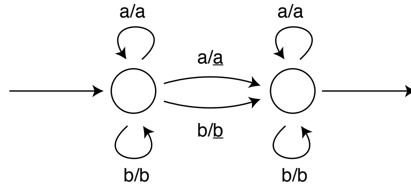
$$R \subseteq \Sigma^* \times \Gamma^*$$

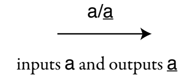
is called a *nondeterministic letter-to-letter transduction* if it can be described this way.

Example 3. Consider the set of pairs

$$(w, v) \in \{a, b\}^* \times \{a, b, \underline{a}, \underline{b}\}^*$$

such that v is obtained from w by underlining exactly one position. This relation is realised by the following automaton:





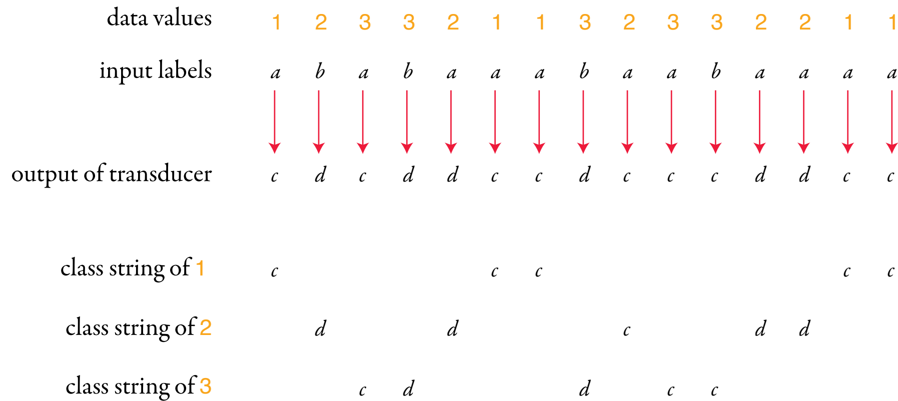
□

Data automata. We are now ready to define a data automaton.

Definition 2.1 *The syntax of a data automaton is given by:*

- *finite input and work alphabets* Σ *and* Γ*;*
- *a nondeterministic letter-to-letter transduction* $R \subseteq \Sigma^* \times \Gamma^*$*;*
- *a regular language* $L \subseteq \Gamma^*$ *called the* class condition*.*

A data automaton is used to accept or rejects data words in $(\Sigma \times \mathbb{A})^*$. For a data word, define a *class* to a maximal set of positions with the same data value, and define a *class string* to be a sequence in Σ^* obtained by taking some class and reading all of its labels from left to right. A data automaton accepts a data word if the sequence of labels can be transformed by the transducer so that in the resulting data word in $(\Gamma \times \mathbb{A})^*$, all class strings are in L. Here is a picture:



The language recognised by a data automaton is the set of accepted data words.

Example 4. A data automaton can check that every data value appears exactly twice. The transducer is the identity, while the class condition contains all words of length exactly two. □

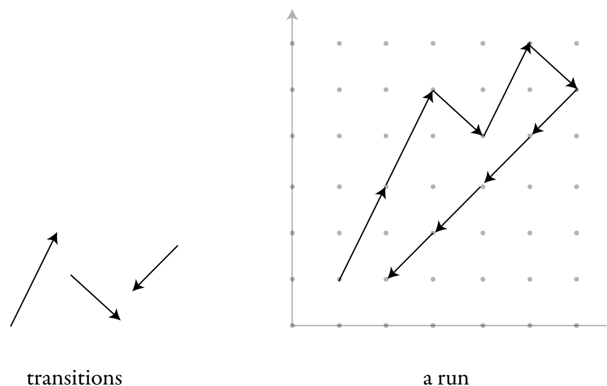
Example 5. A data automaton can check that some data value appears an even number of times. The transducer underlines exactly one position, as in Example 3. The class condition says that if a word contains an underlined position, then it has even length. □

Example 6. Suppose that the input alphabet is $\{a, b, c\}$. Consider a data automaton where the transducer is the identity and the class condition says that each letter occurs exactly once. The language recognised by this data automaton, after erasing data values, is the set of words where each letter appears the same number of times. □

Vector addition systems. We will prove that emptiness for data automata is decidable, because it reduces to reachability problem for vector addition systems, which is known to be decidable, although highly challenging. A *vector addition system* is any finite $\delta \subseteq \mathbb{Z}^d$ of integer vectors with a common dimension d, called the transitions. A run of a vector addition system is a sequence

$$v_0, v_1, \ldots, v_n \in \mathbb{N}^d \qquad \text{such that } v_i - v_{i-1} \in \delta \text{ for every } i \in \{1, \ldots, n\}.$$

Note that all vectors in the run must be nonnegative on all coordinates, even if δ can use negative numbers. Here is a picture in dimension two



The *reachability problem* for vector addition systems is to decide, given two vectors of natural numbers, if there exists a run that begins in the first vector and ends in the last one. The following famous result uses one of the most difficult decidability proofs; this proof is not included in these lecture notes.

Theorem 2.2 *The reachability problem for vector addition systems is decidable.*

A vector addition system can be used as a language recogniser in the following way. Define a *multicounter automaton* to be vector addition system $\delta \subseteq \mathbb{Z}^d$ together with designated initial and final vectors in $\mathbb{N}^d$, as well as a relation $\gamma \subseteq \delta \times \Sigma$ which associates to each transition the input letters that can be used for it. A word in Σ^* is accepted if there exists a run from the initial to the final vector, which is consistent with the input word according to γ. Emptiness for multicounter automata is the same problem as reachability for vector addition systems, and is therefore decidable.

Example 7. Here is a multicounter automaton which recognises the set of words over $\{a, b\}$ where the number of a's is equal to the number of b's. We use two counter names a, b. When reading an a letter, we can either increment the a counter, or decrement the b counter. When reading a b, we can either increment the b counter, or decrement the a counter. The initial vector is $(0, 0)$ and the final vector is also $(0, 0)$. □

Lemma 2.3 *Every regular language is recognised by a multicounter automaton.*

Proof
Consider a regular language recognised by a nondeterministic automaton with states which are numbers $\{1, \ldots, n\}$. To simulate this automaton, we use a multicounter automaton with n counters, where state q is encoded by a vector

$$(0, \ldots, 0, \underbrace{1}_{q\text{-th counter}}, 0, \ldots, 0)$$

A transition which goes from q to p is represented by the integer vector which decrements q and increments p. (To be completely precise, the lemma only works for regular languages $L \subseteq \Sigma^+$, i.e. regular languages of nonempty words, since otherwise we would need the initial and final vectors to be the same. If a regular language does not contain the empty word, then one can find a nondeterministic automaton with exactly one initial and exactly one accepting state.) □

For a language $L \subseteq \Sigma^*$ define $\mathsf{shuffle}L$ to be all words in Σ^* which can be labelled with data values so that all class strings are in L.

Lemma 2.4 *If L is regular, then* $\mathsf{shuffle}L$ *is recognised by a multicounter automaton.*

Proof
Suppose that L is recognised by a deterministic automaton with states Q. Without loss of generality assume that this automaton has no self-loops, i.e. transitions which have the same source and target state. We define a multicounter automaton with one counter per state from Q. The initial and final vectors are the same, namely the zero vector. For every transition $q \xrightarrow{a} p$ of the automaton recognising L, we create a transition in the multicounter automaton which reads a, decrements counter q and increments counter p. If q is the initial state, then we also create a transition which reads a and only increments counter p. If p is a final state, then we also create a transition which reads a and only decrements counter q. □

Emptiness for data automata. In the proof of the following theorem, we see that emptiness for data automata is the same thing as emptiness for multicounter automata, and therefore the same thing as reachability for vector addition systems.

Theorem 2.5 *Emptiness is decidable for data automata.*

Proof
A data automaton is nonempty if and only if there exists a word over the work alphabet which is a possible output of the transducer, and such that the word can be labelled by data values so that every class string is in the class condition of the data automaton. The set of possible outputs of the transducer is easily seen to be a regular language (a nondeterministic automaton can guess the input and the run of the transducer on it). Using the shuffle terminology, we have just shown that emptiness of data automata reduces to the following problem: given regular languages $L, K \subseteq \Gamma^*$ decide if

$$K \cap \mathsf{shuffle} L = \emptyset.$$

The language K is recognised by a multicounter automaton thanks to Lemma 2.3, while $\mathsf{shuffle} L$ is recognised by a multicounter automaton thanks to Lemma 2.4. Languages recognised by multicounter automata are easily seen to be closed under intersection, and therefore the problem above boils down to testing nonemptiness for an effectively obtained multicounter automaton, which is decidable thanks to Theorem 2.2.□

Exercise 26. Show that languages recognised by data automata are not closed under Kleene star.

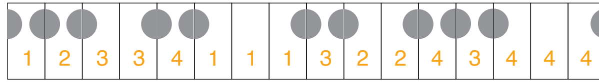
Exercise 27. Show that emptiness is decidable for vector addition systems if the definition of a run is modified so that the intermediate vectors are allowed to use negative coordinates, i.e. the intermediate coordinates are vectors in $\mathbb{Z}^d$.

2.2 Recognising equality with successors

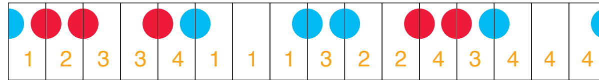
We now show that a data automaton can compute which positions in a data word have the same data value as their neighbours.

Lemma 2.6 *There is a data automaton which recognises the set of data words where the label of each position is the subset of {predecessor, successor} that indicates which neighbours of the position have the same data value.*

Proof
Instead of writing a set we draw a semicircle on the left side if the label does not contain "predecessor" and a semicircle on the right side if the label does not contain "successor", as in the following picture:



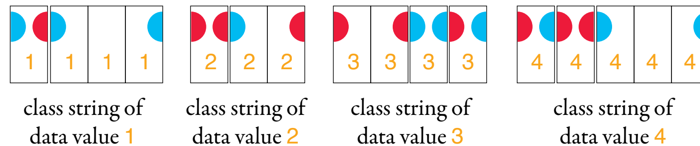
Given an input data word, the transducer in the data automaton guesses a colouring of the semicircles with two colours as in the following picture:



Such a colouring is called consistent (the above picture is consistent) if it satisfies the following conditions:

1. An edge connecting two consecutive positions is labelled by a monochromatic circle, or no circle at all (technically speaking, the label of such an edge is distributed across the two connected positions).
2. If x, y are consecutive positions in some class (but they might be separated by positions from other classes), then the right side of x and the left side of y either form no circle at all, or have different colours. To see that this condition is true in the data word from the picture above, consider the following picture which shows the class strings:

this little break indicates that there positions from
other classes in between, although this is not visible to the class condition



class string of data value 1 | class string of data value 2 | class string of data value 3 | class string of data value 4

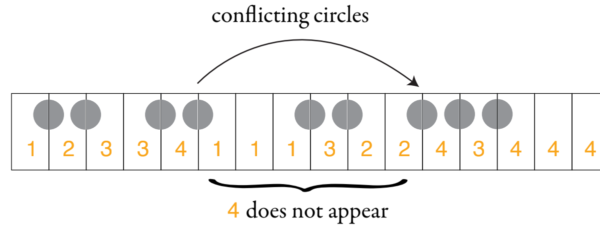
3. The first position in each class has a semicircle on its left, and the last position in each class has a semicircle on its right.

The notion of consistency is designed so that condition 1 can be checked by the transducer in a data automaton, and conditions 2 and 3 can be checked by the class condition. In the following two claims, we show that a data word belongs to the language if and only if it can be coloured in a consistent way.

Claim 2.6.1 *Every data word in the language can be coloured in a consistent way.*

Proof
Consider a data word in the language. We need to colour each grey circle (i.e. pairs

of consecutive positions with different data values) with a single colour so that condition 2 is satisfied. We say that two circles are in conflict if the left half of the left circle has the same data value as the right half of the second circle, and this data value does not appear in between, as in the following picture:



Condition 2 in the definition of consistency says that conflicting circles cannot have the same colour. If we view the conflict relation as a directed graph on circles, with arrows pointing from left to right, then this graph is a forest, i.e. every circle has indegree at most one. A forest can always be coloured with two colours so that edges have endpoints with different colours. □

Claim 2.6.2 *If a data word can be coloured consistently, then it is in the language.*

Proof
Suppose that a data word can be coloured consistently. We need to show that there is a circle connecting two consecutive positions if and only if these positions have different data values.

For the left-to-right implication, consider a circle connecting x and its successor. By condition 1 this circle is monochromatic, say it has colour c. By condition 2 of consistency, the next position in the class of x has its left side coloured with a different colour than c, and hence the next position in the class of x cannot be the successor of x.

We prove the right-to-left implication by doing an inductive left-to-right pass. Consider consecutive positions x and $x + 1$ with different data values. To prove that they are connected by a circle, by condition 1 it suffices to prove that the left side of $x + 1$ has a semicircle. If $x + 1$ is the first position in its class, then it has a semicircle on its left by condition 3, otherwise $x+1$ has a previous position in its class and then we use the induction assumption. □

By Claims 2.6.1 and 2.6.2, a data word belongs to the language in the statement of the lemma if and only if its circles can be coloured in a consistent way. Checking if such a colouring exists is done by the data automaton. □

Exercise 28. Let $k \in \{0, 1, \ldots\}$. Show that there is a data automaton which recognises the set of data words where a position x is labelled by the set of those $i \in \{0, 1, \ldots, k\}$ such that x and $x + i$ have the same data value.

Exercise 29. Recall the notion of class string in the definition of a data automaton, where the positions from outside the class are erased, as in this picture:

data values	1	2	3	3	2	1	1	3	2	3	3	2	2	1	1
labels	*a*	*b*	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*a*	*a*
class string of 1	*a*					*a*	*a*							*a*	*a*

Consider an alternative definition of class string, where the positions from outside are replaced by question marks, like this:

data values	1	2	3	3	2	1	1	3	2	3	3	2	2	1	1
labels	*a*	*b*	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*a*	*a*
class string of 1	*a*	?	?	?	?	*a*	*a*	?	?	?	?	?	?	*a*	*a*

Show that data automata defined with this alternative notion of class string have the same expressive power as original model of data automata.

Exercise 30. In the spirit of the previous exercise, consider yet another definition of class string, where the positions from outside the class are coloured red, like this:

data values	1	2	3	3	2	1	1	3	2	3	3	2	2	1	1
labels	*a*	*b*	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*a*	*a*
class string of 1	*a*	*b*	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*a*	*a*

Show that data automata defined with this alternative definition are strictly more expressive than the original model of data automata.

Exercise 31. Consider a sequence of data values where every position is labelled by a subset of {cut, chosen}. We say a position is chosen if its label contains "chosen" and we say that two positions $x < y$ are in *the same interval* if "cut" does not appear in the labels of positions $x + 1, \ldots, y$. Show that there is a data automaton recognising the data words which satisfy the following two conditions:

- all chosen positions in the same interval have the same data value; and
- there is no non-chosen position which has the same data value as some chosen position in the same interval.

Exercise 32. Show that every language recognised by a nondeterministic register automaton is also recognised by a data automaton. (Hint: use the previous exercise.)

2.3 A logic recognised by data automata

In this section we use the decidability of data automata and Lemma 2.6 to show that satisfiability is decidable for a certain modal logic expressing properties of data words.

Define a *modality* to be a formula describing a relationship between two positions x and y in a data word, which is of the form $\alpha \wedge \beta$ where α is either "x has the same data value as y" or "x has a different data value than y" and β is one of the following four formulas:

$$x < y - 1 \qquad x = y - 1 \qquad x = y + 1 \qquad x > y + 1.$$

There are eight modalities in total. Using these modalities, we define a logic on data words, call it *data word modal logic*. Each formula of this logic selects a set of positions in a data word. The formulas of the logic are:

- Every letter $a \in \Sigma$ is a formula, which selects all positions that have label a.
- The set of formulas is closed under Boolean combinations, including negation.
- If m is a modality and φ is a formula, then $\langle m \rangle \varphi$ is a formula, which selects a position x if there exists a position y selected by φ such that $m(x, y)$.

Define the *language* of a formula to be the data words whose first position is selected.

Theorem 2.8 *Every language defined by a formula of the modal logic is recognised by a data automaton.*

The main step in the proof is to show that every modality can be recognised by a data automaton, in the sense made explicit by the following lemma.

Lemma 2.9 *Let m be one of the modalities. Consider data words each position is coloured by a subset of $\{red, blue\}$. The following property is recognised by data automaton:*

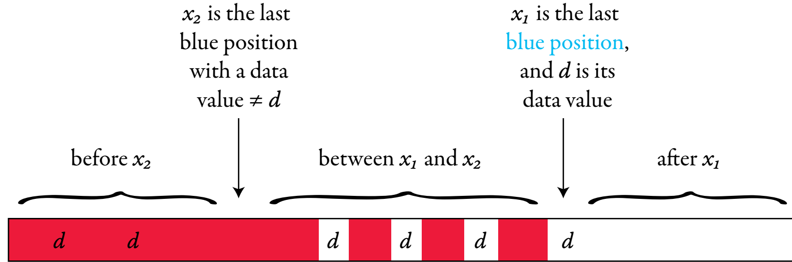
A position x is red if and only if there is a blue position y with $m(x, y)$.

Proof

Consider a modality $\alpha \wedge \beta$. If α says that "x and y have the same data value", then the property in the statement of the lemma can be checked using the class automaton, with the help of the labelling from Lemma 2.6 to test which neighbours have the same data value. If β says that y is the successor or predecessor of x, then we can also use Lemma 2.6. The only remaining case is when the modality m is

(*) $\qquad y$ has different data value than x and $y > x + 1$

or the symmetric one with $y < x - 1$. Because languages recognised by data automata are closed under reversing, we only consider the modality (*). Consider the following parts of a data word:



Some parts might be undefined, e.g. x_1 is undefined if there are no blue positions and x_2 is undefined if at most one data value is used for blue positions. For the modality (*), the property in the statement of the lemma can be reformulated as follows

1. if a position has data value $\neq d$, then it is red if and only if it is $< x_1 - 1$.
2. if a position has data value d, then it is red if and only if it is $< x_2 - 1$.

Both of these properties are regular properties of the labelling, assuming that the partition into the parts concerning x_1 and x_2 is known. This partition can be guessed and checked by a data automaton, assuming that special colours are used to mark the classes of x_1 and x_2. □

Proof (of Theorem 2.8)
Let φ be a formula of data word modal logic. Given an input word, the data automaton nondeterministically guesses for each position x in the data word a set Γ_x of subformulas of φ, intuitively speaking those formulas which select it. Then it checks that:

1. The set Γ_1 contains φ.
2. A position x has label a if and only if $a \in \Gamma_x$.
3. For every position x, the set Γ_x contains a subformula $\varphi_1 \wedge \varphi_2$ of φ if and only if Γ_x contains both φ_1, φ_2. Likewise for $\vee$ and $\neg$.
4. For every position x, the set Γ_x contains a subformula $\langle m \rangle \psi$ of φ if and only if there is some position y such that $m(x, y)$ and $\psi \in \Gamma_y$.

The first three conditions are regular properties of the labelling and can be checked by the transducer while guessing the sets Γ_x. For every fixed choice of subformula $\langle m \rangle \psi$, the last condition can be verified by a data automaton using Lemma 2.9, by choosing red for those positions that have $\langle m \rangle \psi$ in their label and choosing blue for positions that have ψ in their label. All of these checks can be done in parallel, hence one data automaton is sufficient. □

Exercise 33. Show that satisfiability for data word modal logic is at least as hard as emptiness for data automata.

Exercise 34. Consider the following alternating automaton based on the modal logic. The automaton has a set of states Q, an initial state q_0, a partition of states into universal and existential, and a transition relation

$$\delta \subseteq Q \times \Sigma \times \text{modalities} \times Q$$

The automaton accepts a data word $w \in (\Sigma \times \mathbb{A})^*$ if player $\exists$ has a winning strategy in the following game played by players $\exists$ and $\forall$. The game begins in the first position and the initial state. If the game is in position x and state q, then the player who owns q chooses a transition $(q, \sigma, m, p) \in \delta$ such that σ is the label of position x, and a position y that is related to x via the modality m. If there is no such transition or position, the player who owns q loses immediately. Otherwise, the game proceeds to state p and some position y. If the game lasts forever, player $\forall$ wins. Show that such an automaton can recognise the language

$$\{a_1 \cdots a_n a_1 \cdots a_n : a_1, \ldots, a_n \text{ are distinct data values}\}$$

Exercise 35. Show that the automaton model in Exercise 34 has undecidable emptiness. Show that if infinite plays are won by $\exists$ then emptiness becomes decidable.

Exercise 36. Consider first-order logic on data words, as described in Exercise 9. Show that adding a predicate $x = y + 1$ for testing the successor relation and limiting the logic to two variables, we get a logic that has the same expressive power as data word modal logic.

Bibliographic notes for Section 2. Data automata, the algorithm for their emptiness using vector addition systems, were introduced in [10]. The original application was for deciding two-variable logic, as in Excercise 36. The model from [10] had a built in test for "the next data value is different than the current one"; the fact that this is redundant (Lemma 2.6) was shown in [5, Proposition 3.6]. Exercise 32 is also due to [5]. Exercise 30 describes a model called *class automata*, which was studied in [14, 4]. Excercise 31 is [3, Theorem 2].

Theorem 2.2 on reachability for vector addition systems was originally shown by Mayr in [42], other proofs include [39] and [40]. Tree generalisations of two variable logic on data words and data automata were discussed in [16] and [33]; these are problems are connected to *branching vector addition systems*, see [30] and the references therein.

Part II
Sets with atoms

In the previous part, we discussed data words and their automata. In this part, we move to a more abstract and general setting, where data words turn out to be words over a finite alphabet, with an appropriate notion of finiteness, and register automata turn out to be finite automata.

3 Sets with atoms and orbit-finiteness

In this section, we introduce sets with atoms. Normal sets, such as $\emptyset$ or $\{\emptyset, \{\emptyset\}\}$, are built out of the empty set and brackets, although the structure of brackets might be complicated, for instance in the real numbers. In sets with atoms, one postulates the existence of an infinite set of atoms, and sets can contain those atoms as well. The atoms are modelled as a logical structure: a universe together with some relations and functions. Examples of atoms that will appear in this text are:

$(\mathbb{N}, =)$	natural numbers (or any countably infinite set) with equality
$(\mathbb{Q}, \leq)$	the rational numbers with their order
$(\mathbb{Z}, +1)$	the integers with the unary successor function

We will use the name *equality atoms* for the first structure, which corresponds to the data values in the first part about data words. Since we do not want to confuse the structureless equality atoms with natural numbers, we will use underlined names like $\underline{1}$ or $\underline{2}$ for examples of equality atoms.

We will construct sets with atoms by using the empty set, atoms, and set brackets. Not every object built this way is going to be considered a set with atoms. The intuitive idea behind a set with atoms is that it is an object built using atoms and set brackets in such a way which only uses the structure given by the atoms (i.e. relations and functions from the vocabulary of the atoms), and finitely many constants that refer to specific atoms. For example, in the equality atoms the set of even numbered atoms $\{\underline{0}, \underline{2}, \underline{4}, \ldots\}$ would *not* be a set with atoms, because a definition of this set would need to either explicitly mention infinitely many atoms, or refer to the notion of "even-numbered" which does not exist in the structure.

3.1 The cumulative hierarchy and its finitely supported elements

Fix a structure for the atoms. 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 will *not* want to consider, such as the set of "even-numbered" atoms mentioned above.

The definition of sets with atoms is obtained by restricting the cumulative hierarchy to sets which satisfy the "finite support condition", which models the idea of being definable using only the structure of the atoms and finitely many constants. The formal definition is defined in terms of automorphisms. We use the term *automorphism* in the same sense as in model theory, i.e. an automorphism of the atoms is any bijection of from the atoms to the atoms which preserves the relations and functions that are part of the atom structure. If an automorphism furthermore preserves a tuple of atoms $\bar{a} = (a_1, \ldots, a_n)$, then it is called an $\bar{a}$-automorphism. An automorphism 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 an automorphism 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 3.1 (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 $\bar{a}$-automorphism 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[1]. The support of a set with atoms is not unique, e.g. supports are closed under adding atoms, although for some atom structures a canonical least support can be found. A set with empty support is called *equivariant*. Intuitively speaking, an equivariant set is one which can be defined without referring to any specific atoms.

Example 8. [Supports] Consider the equality atoms and the set X of all atoms except $\underline{2}$. This set is supported by the atom $\underline{2}$, because any $\underline{2}$-automorphism will preserve X as a set, but it might rearrange its elements. The set X is not equivariant, so $\underline{2}$ is a minimal support, actually it is the least finite support. □

Since a set can have many definitions, there might not be a canonical support. For instance, when the atoms $(\mathbb{Z}, +1)$, then the set $\{2\}$ is supported by 2, but it is also supported by 1 because it can be defined by "the singleton of the successor of 1".

Definition 3.2 (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. As for definable sets, when talking about pairs, 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.

Example 9. [Finitely supported subsets of the equality atoms] Consider the equality atoms. Which subsets of the set of all atoms are finitely supported? We claim that, when the only structure is equality, then the finitely supported sets of atoms are exactly the finite and co-finite sets. It is not difficult to see that the finite and co-finite sets are finitely supported. For the converse implication, consider a set X of atoms

[1] For this reason, many authors use a set of atoms as a support, instead of a tuple of atoms. We use tuples so that we can distinguish between an $\{a_1, a_2\}$-automorphism and an (a_1, a_2)-automorphism. The former can swap a_1 and a_2, while the latter needs to fix both a_1 and a_2.

that is neither finite, nor co-finite. We will show that X cannot have finite support. Suppose then that a finite tuple of atoms $\bar{a}$ is a candidate for a finite support. Since both X and its complement are infinite, there must be atoms $a \in X$ and $b \notin X$ such that both a and b do not appear in the tuple $\bar{a}$. Let π be the permutation of atoms which swaps a and b, and is the identity on other atoms. This permutation is an $\bar{a}$-automorphism, so it should fix X, but it does not. □

Example 10. [Finitely supported subsets of ordered rational numbers] Consider the atoms $(\mathbb{Q}, <)$. In this case, the automorphisms are order preserving bijections. We claim that the finitely supported subsets of atoms are exactly finite unions of intervals. Consider a set X of atoms which is supported by a tuple of atoms $\bar{a}$. We claim that X is a union of intervals (open, closed, open-closed or closed-open) whose endpoints are either $-\infty, \infty$, or appear in $\bar{a}$. Indeed, consider atoms a, b that are not in $\bar{a}$ and are not separated by an atom in $\bar{a}$ in terms of the order. There is an $\bar{a}$-automorphism which maps a to b. Since the set X is supported by $\bar{a}$, it follows that $a \in X$ if and only if $b \in X$. □

In both examples above, the finitely supported sets of atoms coincide with subsets of atoms that can be defined by quantifier-free formulas which can use constants from the atoms. The reason is that both examples of atoms are *homogeneous* structures, which are discussed in Section 6, and when the atoms are homogeneous, then finitely supported relations on the atoms are exactly those that can be defined using quantifier-free formulas. In general, when the atoms are not homogeneous, finitely supported sets are not the same thing as quantifier-free definable sets, as shown in the following example.

Example 11. [Finitely supported subsets of the integer atoms] Consider the atoms $(\mathbb{Z}, <)$. For this structure, the automorphisms are exactly the translations, i.e. functions of the form $x \mapsto a + x$ for some constant $a \in \mathbb{Z}$. We claim that all sets in the cumulative hierarchy are sets with atoms. This is because under the integer atoms, the finite support condition is trivially true, because for every nonempty tuple of integers, only the identity automorphism preserves the tuple, and therefore every set in the cumulative hierarchy is supported by any such tuple, e.g. 1. On the face of it, this sounds like good news, e.g. sets with atoms under the integer atoms are closed under arbitrary subsets, unlike for most other atoms. However, there is a price to pay, as we shall see in later sections. □

The above examples show that in sets with atoms, the appropriate notion of powerset is the *finitely supported powerset*, which contains only the finitely supported subsets of a given set. As shown in Examples 9 and 10, the finitely supported powerset can be smaller than the standard powerset.

Exercise 37. For the equality atoms, find all equivariant binary relations on $\mathbb{A}$.

Exercise 38. For the atoms $(\mathbb{Q}, <)$, find all equivariant binary relations on $\mathbb{A}$.

Exercise 39. Show that a function $f : X \to Y$ is supported by a tuple of atoms $\bar{a}$ if and only if the following diagram commutes for every $\bar{a}$-automorphism π:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ \pi \downarrow & & \downarrow \pi \\ X & \xrightarrow[f]{} & Y \end{array}$$

Exercise 40. Consider the equality atoms. Let us denote the family of two-element subsets of $\mathbb{A}$ by $\mathsf{P}_2(\mathbb{A})$. Show that there is no function

$$f : \mathsf{P}_2(\mathbb{A}) \to \mathbb{A}$$

which is a set with atoms and maps every two-element set to one of its elements.

Exercise 41. Consider the equality atoms. Show a finitely supported graph, which admits a two-colouring that is not finitely supported, but does not admit any finitely supported two-colouring.

Exercise 42. Consider the equality atoms. Show that for every finitely supported partial order $<$ on $\mathbb{A}$, all atoms outside the support are incomparable.

Exercise 43. Consider the atoms $(\mathbb{Q}, <)$. Show that there is no finitely supported well-founded total order on $\mathbb{A}$.

Exercise 44. For a countable set A enumerated as $a_1, a_2, \ldots$ define the distance between two different bijections $A \to A$ to be $1/n$ where a_n is the first argument where the bijections disagree. Show that if X is an equivariant set with atoms over countable atoms $\mathbb{A}$, then all elements of X are finitely supported if and only if

$$\underbrace{\pi}_{\text{automorphisms of } \mathbb{A}} \qquad \mapsto \qquad \underbrace{(x \mapsto \pi(x))}_{\text{bijections of } X}$$

is a continuous mapping, and this continuity does not depend on the choice of enumerations of $\mathbb{A}$ or X.

3.2 Orbit-finiteness

In this section, we describe the main reason for considering sets with atoms: in sets with atoms there is a different, more relaxed, notion of finiteness. This notion is called orbit-finiteness and is the main notion in this book.

Oligomorphism. Before defining orbit-finiteness, we describe the key restriction on the atom structures that will be necessary for orbit-finiteness to make sense. A logical structure $\mathbb{A}$ is called *oligomorphic* if for every $n \in \{1, 2, \ldots\}$, the structure $\mathbb{A}^n$ has finitely many elements up to automorphisms. More precisely, for every n the following equivalence relation on n-tuples of atoms has finitely many equivalence classes:

$$\bar{a} \sim \bar{b} \qquad \text{if } \pi(\bar{a}) = \bar{b} \text{ for some automorphism } \pi \text{ of } \mathbb{A}$$

Example 12. The equality atoms $\mathbb{A} = (\mathbb{N}, =)$ are oligomorphic. Two n-tuples of atoms are equivalent if and only if they have the same equality type, and there are finitely many equality types for every fixed n. $\square$

Example 13. The ordered rational numbers $\mathbb{A} = (\mathbb{Q}, <)$ are oligomorphic. Two n-tuples of atoms are equivalent if and only if they have the same type with respect to $<$, for example $(3, 2.5, -1)$ is equivalent to $(8.2, 1, 0.5)$. $\square$

Example 14. The integers with successor $\mathbb{A} = (\mathbb{Z}, +1)$ are not oligomorphic. An automorphism is a translation, i.e. a function of the form $x \mapsto a + x$ for some fixed $a \in \mathbb{Z}$. For $n = 1$, there is only one equivalence class, but for $n = 2$ there are infinitely many equivalence classes, because the equivalence class of $(a, b) \in \mathbb{Z}^2$ is determined by the difference $a - b$. $\square$

Orbits and orbit-finiteness. We are now ready to introduce orbit-finiteness. Intuitively speaking, a set is orbit-finite if it has finitely many elements, up to atom automorphisms. The precise definition is given below, and it only makes sense when the atoms are oligomorphic.

Let $\bar{a}$ be a tuple of atoms. Define an *$\bar{a}$-orbit* to be a set of the form

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

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 a 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, at least as long as the atoms are oligomorphic:

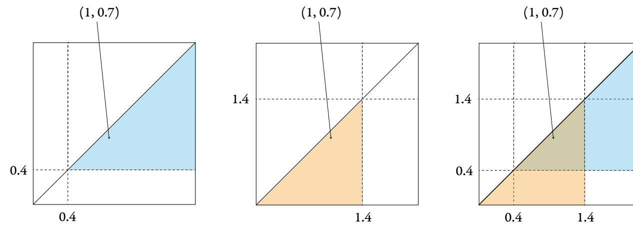
Theorem 3.3 *If the atoms $\mathbb{A}$ are oligomorphic, then for every set with atoms X, the following conditions are equivalent:*

- *X is a finite union of $\bar{a}$-orbits for 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*. We do not talk about orbit-finiteness when the atoms are not oligomorphic.

Example 15. [The set of all atoms is orbit-finite] If the atoms are oligomorphic, then the set $\mathbb{A}$ of all atoms is orbit-finite, by assumption on oligomorphism. If the atoms are the equality atoms or the rational numbers with order $(\mathbb{Q},<)$, then the set $\mathbb{A}$ is even a single $\emptyset$-orbit. There are, however, oligomorphic atom structures where there are two orbits of atoms. An example is two disjoint copies of the equality atoms, one coloured red and the other blue. □

Example 16. [Different decompositions of orbit-finite sets] Consider the atoms $(\mathbb{Q},<)$ and the 0.4-orbit of the ordered pair $(1, 0.7)$ plus the 1.4-orbit of the same ordered pair. This set union is illustrated below:



The set can be decomposed into non-intersecting orbits, e.g. under the action of $(0.4, 1.4)$-automorphisms, but then 9 orbits are needed, accounting for the coloured areas and some of the dotted lines. □

Example 17. [Number of orbits in X^2 can be arbitrarily big, even when X has one orbit] Even when the product has finitely many orbits, the number of orbits can grow a lot. Consider the equality atoms. Let $n \in \mathbb{N}$. We denote the set of non-repeating n-tuples of atoms by $\mathbb{A}^{(n)}$. This set is one equivariant orbit, because every non-repeating tuple can be mapped to every other non-repeating tuple by an automorphism of atoms, since the only structure is equality. The square of this set,

$$\mathbb{A}^{(n)} \times \mathbb{A}^{(n)}$$

has a number of equivariant orbits that is exponential in n. More precisely, the set has one orbit for every partial bijection between $\{1, \ldots, n\}$ and $\{1, \ldots, n\}$, because there are different possible ways in which the coordinates of the first tuple might be equal to the coordinates of the second tuple. In particular, the number of orbits of X^2 can be arbitrarily big, even when X has one orbit, and the number of orbits can depend on an additional parameter like the dimension n. □

We now begin the proof of Theorem 3.3. We use two lemmas. The second one, Lemma 3.5, is true for atoms which are not necessarily oligomorphic.

Lemma 3.4 *If the atoms are oligomorphic, then for every atom tuple $\bar{a}$ and every dimension $n \in \{0, 1, \ldots\}$ there are finitely many $\bar{a}$-orbits in $\mathbb{A}^n$.*

Proof
Let k be the dimension of $\bar{a}$. Two n-tuples of atoms $\bar{b}$ and $\bar{c}$ are in the same $\bar{a}$-orbit if and only if the $(k+n)$-tuples $\bar{a}\bar{b}$ and $\bar{a}\bar{c}$ are in the same $\emptyset$-orbit. By oligomorphism there are finitely many possibilities for the latter. □

Lemma 3.5 *If $\bar{a}$ is a tuple of atoms, then every set with atoms that is an $\bar{a}$-orbit is equal to the image of some $\bar{a}$-orbit in $\mathbb{A}^n$ under an equivariant function.*

Proof
Let X be a set with atoms which is a $\bar{a}$-orbit. By definition, X is of the form

$$\{\pi(x_0) : \pi \text{ is an } \bar{a}\text{-automorphism}\}$$

for some $x_0 \in X$ which is an atom or set with atoms. Choose some tuple of atoms $\bar{b}$ which supports x_0, which must exist because x_0 is an atom or a set with atoms. Consider the following set of pairs:

$$f = \{\pi(\bar{b}, x_0) : \pi \text{ is an atom automorphism}\}.$$

(An automorphism acts on pairs coordinate-wise.) Because the definition of f talks about arbitrary atom automorphisms, and not just $\bar{a}$-automorphisms, it follows that f is an equivariant set of pairs. We claim that f is actually a function

$$f : \emptyset\text{-orbit of } \bar{b} \to \emptyset\text{-orbit of } x_0.$$

To prove functionality, we need to show that for every atom automorphisms π and σ we have

$$\pi(\bar{b}) = \sigma(\bar{b}) \qquad \text{implies} \qquad \pi(x_0) = \sigma(x_0).$$

By applying π^{-1} to all terms in the above implication, we see that it is just another way of saying that $\bar{b}$ supports x_0. If we restrict the domain of f to the $\bar{a}$-orbit of $\bar{b}$, then the image is going to be the $\bar{a}$-orbit of x_0, which is the set X. □

Proof (Proof of Theorem 3.3)
Only the top-down implication needs to be shown. We will show that if $\bar{a}$ is an atom tuple which supports X, and $\bar{b}$ is another atom tuple, then if X splits into finitely many $\bar{a}$-orbits, then it also splits into finitely many $\bar{a}\bar{b}$-orbits. In other words, increasing the support can make the number of orbits grow, but will keep it finite. Since every two atom tuples can be extended to a common one, the result will follow.

Let $\bar{a}$ be a tuple of atoms and let X be a set with atoms which is a single $\bar{a}$-orbit. We need to show that for every tuple of atoms $\bar{b}$, the set X is a finite union of $\bar{a}\bar{b}$-orbits. Apply Lemma 3.5 to X, yielding a surjective equivariant function f with domain $\mathbb{A}^n$ such that $X = f(Y)$ for some $\bar{a}$-orbit $Y \subseteq \mathbb{A}^n$. By Lemma 3.4, Y is a finite union of $\bar{a}\bar{b}$-orbits. If f is an equivariant function, then images of $\bar{a}\bar{b}$-orbits are also $\bar{a}\bar{b}$-orbits. Therefore, X is also a finite union of $\bar{a}\bar{b}$-orbits. □

The following fact shows that orbit-finiteness has some of the same closure properties as finiteness. An important exception is powerset, which is discussed in Example 18.

Fact 3.6 *Assume that the atoms are oligomorphic. Show that orbit-finite sets are closed under binary union, binary product, finitely supported subsets and images under finitely supported functions.*

Proof

Binary union is immediate. Consider below the remaining cases.

- **Finitely supported subsets.** Let X be orbit-finite and let $Y \subseteq X$. Let $\bar{a}$ be a tuple of atoms which supports both Y and X. By Theorem 3.3, X is a finite union of $\bar{a}$-orbits. Since Y is supported by $\bar{a}$, it is a union of $\bar{a}$-orbits, and therefore this union must be finite.

- **Binary products.** By Theorem 3.3, it suffices to show that for every tuple of atoms $\bar{a}$, the product of every two $\bar{a}$-orbits X_1, X_2 is orbit-finite. By Lemma 3.5, for every $i \in \{1, 2\}$ we can find an equivariant function

  $$f_i : \mathbb{A}^{n_i} \to \text{sets with atoms}$$

  such that X_i is the image under f_i of some $\bar{a}$-orbit $Y_i \subseteq X_i$. The product $X_1 \times X_2$ is the image of $Y_1 \times Y_2$ under the equivariant function obtained by combining f_1 and f_2 in the natural way. The set $Y_1 \times Y_2$ is supported by $\bar{a}$, and therefore by Lemma 3.4 it is orbit-finite. Since orbit-finiteness is preserved under images of equivariant functions, the result follows.

- **Images under finitely supported functions.** Let $f : X \to Y$ be a finitely supported function and let $X_0 \subseteq X$ be orbit-finite. Let $\bar{a}$ be a tuple of atoms which supports both f and X_0. It is not difficult to see that whenever $x, y \in X_0$ are in the same $\bar{a}$-orbit, then also their images $f(x), f(y)$ are in the same $\bar{a}$-orbit. It follows that the number of $\bar{a}$-orbits in the image $f(X_0)$ is at most as big as the number of $\bar{a}$-orbits in X_0, and therefore finite.

□

Example 18. [Powerset does not preserve orbit-finiteness.] Recall that in sets with atoms, the appropriate notion of powerset is "the family of all finitely supported subsets". We show that even in the equality atoms, this powerset operation does not preserve orbit-finiteness. Indeed, take the set $\mathbb{A}$ of all atoms, which is orbit-finite. Every finite subsets is a finitely supported. Two finite sets of different cardinality cannot be in the same orbit, therefore the powerset is not orbit-finite. □

Exercise 45. Let $R \subseteq X \times X$ be a binary relation which is an orbit-finite set with atoms. Show that the transitive closure of R is also orbit-finite.

Exercise 46. Show the following converse of Theorem 3.3: if the atoms have finitely many $\emptyset$-orbits and the two conditions in the statement of the theorem are equivalent for every set with atoms X, then the atoms are oligomorphic.

Exercise 47. Assume that the atoms are oligomorphic. Show that in an orbit-finite set, for every atom tuple $\bar{a}$ there are finitely many elements supported by $\bar{a}$.

Exercise 48. Show a counterexample, in the equality atoms, to the converse implication from Exercise 47. In other words, show a set which is not orbit-finite, but where every tuple of atoms supports finitely many elements.

Exercise 49. Define a partial equivalence relation to be a relation which is symmetric and transitive, but not necessarily reflexive. If $\sim$ is a partial equivalence relation on a set X, then $X/_\sim$ denotes the family of equivalence classes, which is a family of pairwise disjoint sets which partitions elements that are equivalent to themselves. Assume that the atoms are oligomorphic and let $\bar{a}$ be an atom tuple. Show that if a set is orbit-finite and supported by $\bar{a}$, then it admits an $\bar{a}$-supported bijection with a set of the form $\mathbb{A}^n/_\sim$ where $\sim$ is a partial equivalence relation supported by $\bar{a}$.

Exercise 50. Assume that the atoms are oligomorphic. Show that if X is an orbit-finite set and $\bar{a}$ is an atom tuple, then

$$\{\pi(x) : x \in X \text{ and } \pi \text{ is an } \bar{a}\text{-automorphism}\}$$

is also orbit-finite.

Exercise 51. Assume that the atoms are oligomorphic. Show that orbit-finite sets are closed under orbit-finite union in the following sense. If X is an orbit-finite set and f is a function that maps each element of X to an orbit-finite set, then

$$\bigcup_{x \in X} f(x)$$

is an orbit-finite set.

Exercise 52. Show that in the equality atoms (actually, under any oligomophic atoms), every orbit-finite is Dedekind finite, i.e. does not admit a finitely supported bijection with a proper subset of itself.

Exercise 53. Show that in the equality atoms, there is a set that is not orbit-finite, but Dedekind finite in the sense from Exercise 52.

Exercise 54. Call a family of sets *directed* if every two sets from the family are included in some third set from the family. Consider the equality atoms. Show that a set with atoms X is finite (in the usual sense) if and only if it satisfies: for every set with atoms $\mathcal{X} \subseteq \mathsf{P}X$ which is directed, there is a maximal element in $\mathcal{X}$.

Exercise 55. Call a family $\mathcal{X}$ of sets *uniformly supported* if there is some tuple of atoms which supports all elements of $\mathcal{X}$. Assume that the atoms are oligomorphic.

Show that a set X is orbit-finite if and only if : (*) there is a maximal element in every set of atoms $\mathcal{X} \subseteq \mathsf{P}X$ which is directed and uniformly supported.

Exercise 56. Show that the following statement is true in the equality atoms but not in $(\mathbb{Q}, <)$. A set X is orbit-finite if and only if: (***) for every set with atoms $\mathcal{X} \subseteq \mathsf{P}X$ which is totally ordered by inclusion, there is a maximal element.

Exercise 57. Assume that the atoms are oligomorphic. Show the following variant of König's lemma. If a tree has orbit-finite branching and arbitrarily long branches, then it has an infinite branch.

Bibliographic notes for Section 3. The idea to have a model of set theory with atoms (also known as *ur-elements*) originates from the work of Frankel in 1922, further developed by Mostowski in the 1930s. At that time, these models were used to prove independence of the axiom of choice, and other axioms. In computer science, atoms were rediscovered by Gabbay and Pitts in [29], under the name *nominal sets*, as a formalism for modeling name binding. Since then, nominal sets have become a lively topic in semantics, see e.g. the book of Pitts [49]. Nominal sets were also independently rediscovered by the concurrency community, as a basis for syntax-free models of name-passing process calculi, see [44, 45].

The notion of oligomorphism made its appearance in model theory in 1959, thanks to a theorem proved independently by Engeler, Ryll-Nardzewski and Svenonius: for a countable structure, being oligomorphic is equivalent to having an ω-categorical theory. One implication of this theorem will be used later, in Lemma 4.3.

To the author's best knowledge, the notion of orbit finiteness was first introduced in [7], which is the conference version of [8]. The purpose of the papers [7, 8] was to find a model of monoids which would capture data values, and also admit minimisation (also known as syntactic monoids).

Exercise 55 is inspired by [49, Section 5.5.]. Exercise 52 is inspired by [6].

4 Definable sets

In this section we introduce definable sets, which are special cases of sets with atoms that can be represented in a finite way. We show that, assuming that the atoms are oligomorphic, the definable sets are exactly the sets with atoms which are hereditarily orbit-finite, i.e. are orbit-finite, have elements which are orbit-finite, and so on. Definable sets will then be used as the basis of computation: we will consider decision problems where the input is a definable set, or computing devices that transform definable sets into other definable sets.

4.1 Definable sets

Consider an arbitrary logical structure $\mathbb{A}$ for the atoms. The idea behind a definable set is that it can be defined using set builder notation. Before giving the formal definition, we give some examples. A typical definable set in the equality atoms is the family of sets which can be obtained from all atoms by subtracting at most one atom:

$$\{\mathbb{A}\} \cup \{\{b : b \in \mathbb{A}, b \neq a\} : a \in \mathbb{A}\}$$

Another example of a definable set, this time in the total order atoms, is the set of all closed intervals:

$$\{\emptyset, \mathbb{A}\} \cup \{\{c : c \in \mathbb{A}, a \leq c \leq b\} : a, b \in \mathbb{A}, a \leq b\}$$

We now present the formal definition of definable sets. Fix some infinite set of variables, which are meant to range over atoms. Let $\mathbb{A}$ be a logical structure. If $\bar{x}$ is a tuple of variables, then an *$\bar{x}$-valuation* is a function that maps each variable in the tuple to an element of the universe in $\mathbb{A}$. Define the *set builder expressions over* $\mathbb{A}$ as follows by structural induction:

- **Atom constant.** Every atom $a \in \mathbb{A}$ is a set builder expression. This expression has no free variables, and it represents the atom a.
- **Variable expression.** A variable x is a set builder expression, called a *variable expression*. The variable x is free in this expression.
- **Set expression.** Let $\bar{x}, \bar{y}$ be disjoint tuples of variables and let α be an already defined set builder expression with free variables contained in $\bar{x}\bar{y}$. Let φ be a first-order formula over the vocabulary of $\mathbb{A}$ with free variables $\bar{x}\bar{y}$, which is allowed to use constants from $\mathbb{A}$ (such constants are called *parameters*). Then

  $$\{\alpha(\bar{x}\bar{y}) : \text{ for } \bar{y} \text{ such that } \varphi(\bar{x}\bar{y})\}$$

  is a set builder expression, called a *set expression*. The free variables are $\bar{x}$, and the variables $\bar{y}$ are bound.
- **Union expression.** If $\alpha_1, \ldots, \alpha_n$ are set builder expressions, then so is $\alpha_1 \cup \cdots \cup \alpha_n$. Such an expression is called a *union expression*. The free variables are those which are free in at least on of $\alpha_1, \ldots, \alpha_n$.

For a set builder expression α with free variables $\bar{x}$, define $[\![\alpha]\!]$ to be the function which inputs a valuation of the free variables in α and outputs the corresponding set (or set of sets, etc.) defined in the natural way. When α has no free variables $\bar{x}$, then $[\![\alpha]\!]$ is simply a set (or atom).

Definition 4.1 (Definable sets) *A* definable set *over a logical structure* $\mathbb{A}$ *is any object of the form* $[\![\alpha]\!]$ *where* α *is a set builder expression without free variables.*

A definable set is either an atom, if α is an atom constant, or a set. Note that the same definable set can be represented using different set builder expressions.

Example 19. Let $\mathbb{A}$ be the ordered rational numbers. Here is an example of a set builder expression which has two free variables x, y and uses no parameters:

$$\beta(x, y) = \{z : \text{ for } z \text{ such that } x < z < y\}.$$

The variable z is bound in β. Given a valuation $(x \mapsto a, y \mapsto b)$ of the free variables, the expression returns the open interval from a to b. Using β, we can define the family of open intervals contained in $(0; 1)$:

$$\alpha = \{\beta(x, y) : \text{ for } x, y \text{ such that } 0 < x < y < 1\}.$$

Note that $0, 1$ are parameters in the expression. □

Exercise 58. A *definable nondeterministic automaton* over atoms $\mathbb{A}$ is the same definition as that of a nondeterministic finite automaton, except the instead of being finite, the components (i.e. states, input alphabet, etc.) are all required to be definable sets over $\mathbb{A}$. Show that every register automaton, as defined in Section 1, is a special case of a definable automaton over the equality atoms.

Exercise 59. Use the notion of definable automata from Exercise 58. Show that emptiness is undecidable for definable nondeterministic (or deterministic) automata over the atoms $(\mathbb{Z}, <)$.

4.2 Hereditarily orbit-finite sets

The notion of a definable set, as defined above, has a syntactic character: it is anything that can be represented using a set builder expression. In this section, we discuss a more semantic notion, namely hereditarily orbit-finite sets. As we will see in Theorem 4.2, these two notions coincide when the atoms are oligomorphic. For some applications the syntactic presentation is more convenient (e.g. definable sets are easy to use as the inputs of algorithms), while for some applications the semantic presentation is more convenient (e.g. Exercise 45 implies that hereditarily orbit-finite sets are closed under transitive closure, which is not clear for definable sets).

Define a *hereditarily orbit-finite set* to be a set with atoms which is orbit-finite, whose elements are orbit-finite, and so on until the empty set is or an atom is reached. This generalises the classical notion of a *hereditarily finite set*, where finiteness is used at every level. An example of a hereditary finite set is the standard set theory encoding of any natural number, e.g. here is the number 3:

$$3 = \{ \overbrace{\emptyset}^{0}, \overbrace{\{\emptyset\}}^{1}, \overbrace{\{\emptyset, \{\emptyset\}\}}^{2} \}$$

An example of hereditarily orbit-finite set is three copies of the atoms:

$$3 \times \mathbb{A}$$

(Recall that when talking about Cartesian product, we use the Kuratowski pairing function, which preserves hereditary orbit-finiteness.) Instead of 3, we could use other natural numbers or ASCII strings etc.

Example 20. Consider a register automaton, say nondeterministic. Every automaton is an example of a hereditarily orbit-finite set. Indeed, formally speaking an automaton is a tuple, so one needs to show that every component of the tuple is hereditarily orbit-finite (because the pairing function preserves hereditarily orbit-finite sets). The most interesting case is the transition relation, and here it suffices to show that the space of all configurations (and therefore also pairs of configurations) forms a hereditarily orbit-finite set. The number of orbits is at least exponential in the number of registers, due to the undefined registers. $\square$

4.3 Definable equals hereditarily orbit-finite

The goal of this section is to show the following theorem.

Theorem 4.2 *Assume that the atoms are countable and oligomorphic. Then a set is definable if and only if it is a hereditarily orbit-finite set with atoms.*

The key part of the above theorem is the following lemma.

Lemma 4.3 *In a countable oligomorphic structure, every equivariant set of n-tuples is first-order definable.*

Proof (rough sketch)
Fix a dimension of tuples n. For a natural number k, define $\equiv_k$ to be the equivalence relation on n-tuples of atoms which says that the tuples satisfy the same formulas of first-order logic with n free variables and given quantifier rank at most k. Define $\equiv$ to be the equivalence relation which says that the tuples are in the same equivariant orbit. Each of these equivalence relations is characterised by an appropriate Ehrenfeucht-Fraïssé game, which differ in the number of rounds that is played: in the game for $\equiv_k$ one plays k rounds, while in the game for $\equiv$ one plays infinitely many rounds. (The game characterisation of $\equiv$ uses a back-and-forth argument that works because of the assumption on countability.) We claim that when the structure is oligomorphic, then Spoiler wins the infinite round game if and only if for some k, he wins the k-round game. In other words, for every two tuples in different orbits, there is a formula of first-order logic that distinguishes them. This claim proves the lemma, since there are finitely many equivariant orbits of given dimension, and therefore each one is definable in first-order logic. Equivariant sets of n-tuples are finite unions of equivariant orbits.

To prove the claim, consider the situation in the infinite round game for $\equiv$ when Spoiler is about to extend a tuple $\bar{a}$ by a new element, and the other tuple is $\bar{b}$. If elements a and a' are in the same $\bar{a}\bar{b}$-orbit, then extending the tuple $\bar{a}$ by a or extending it by a' will give the same results as far as winning the game is concerned. Since there are finitely many $\bar{a}\bar{b}$-orbits, Spoiler has essentially finitely many different choices. The same holds for Duplicator. Therefore, one can use the König lemma to show that Spoiler wins the infinite round game if and only if for some k he wins the k-round game. □

Example 21. Consider the equality atoms. It is not difficult to see that every $\emptyset$-orbit in $\mathbb{A}^n$ is determined by its equality type, and therefore it is first-order definable, even without quantifiers. □

Example 22. Consider the atoms $(\mathbb{Q}, <)$. It is not difficult to see that every $\emptyset$-orbit in $\mathbb{A}^n$ is determined by the order type on the coordinates, and therefore it is first-order definable, even without quantifiers. □

In the above two examples, even quantifier-free formulas were sufficient to define equivariant sets of atom tuples. The reason is that the atom structures in these examples satisfy a property stronger than oligomorphism, which will be discussed in

Section 6, namely they are *homogeneous*. Not all oligomorphic structures have this property, as shown in the following example.

Example 23. Let $\mathbb{A}$ be the undirected graph which consists of a countably infinite disjoint union of cycles of length 4. It is not difficult to show that this is an oligomorphic structure. Consider the equivariant set

$$\{(a,b) : a,b \in \mathbb{A} \text{ are antipodal, i.e. } a \neq b \wedge \exists c\, E(a,c) \wedge E(c,b)\}.$$

The set is clearly definable in first-order logic, but not without quantifiers. □

Corollary 4.4 *Suppose that the atoms are a countable oligomorphic structure. If a set of n-tuples of atoms is supported by a tuple of atoms $\bar{a}$, then it is definable by a first-order formula with n free variables and constants from $\bar{a}$.*

Proof
Consider a set X of n-tuples that is supported by a tuple $\bar{a}$ of dimension k. Define

$$Y = \{\pi(\bar{a}\bar{b}) : \pi \text{ is an atom automorphism and } \bar{b} \in X\}.$$

This is an equivariant set, and therefore by Lemma 4.3 it is definable by a formula of first-order logic φ. A tuple $\bar{b}$ belongs to X if and only if it satisfies $\varphi(\bar{a}\bar{b})$. □

Proof (of Theorem 4.2)
Let us begin with the left-to-right implication. We show that for every set builder expression α, its semantics $[\![\alpha]\!]$ is an equivariant function that inputs tuples of atoms and outputs hereditarily orbit-finite sets. Equivariance is immediate, since truth of first-order formulas is invariant under automorphisms. To show that the outputs are hereditarily orbit-finite, we use induction on the size of α. Consider the interesting case in the induction step, i.e. a set builder expression of the form

$$\beta(\bar{x}) = \{\alpha(\bar{x}\bar{y}) : \text{ for } \bar{y} \text{ such that } \varphi(\bar{x}\bar{y})\}.$$

For a $\bar{x}$-tuple of atoms $\bar{a}$ we see that $[\![\beta]\!](\bar{a})$ is the image under $[\![\alpha]\!]$ of the set X of $\bar{x}\bar{y}$-tuples of atoms that begin with $\bar{a}$ and satisfy the formula φ. The set X is a set of atoms tuples of fixed dimension which is supported by $\bar{a}$, and therefore it is orbit-finite thanks to oligomorphism. By induction assumption $[\![\alpha]\!]$ is an equivariant function from atom tuples to hereditarily orbit-finite sets. From Fact 3.6 it follows that hereditarily orbit-finite sets are closed under images of finitely supported functions which output only hereditarily orbit-finite sets, and thus the result follows.

We now turn to the right-to-left implication in Theorem 4.2. By induction on the rank in the cumulative hierarchy, we show that every hereditarily orbit-finite set supported by an atom tuple $\bar{a}$ can be presented as $[\![\alpha]\!](\bar{a})$ for some set builder expression α, and is therefore definable. Let then X be a hereditarily orbit-finite set supported by $\bar{a}$. Since definable sets are closed under finite unions, it suffices to consider the case when X consists of a single $\bar{a}$-orbit. Choose some $x \in X$, with support $\bar{b}$. In particular, x is also supported by $\bar{a}\bar{b}$. By induction assumption, $x = [\![\alpha]\!](\bar{a}\bar{b})$ for some set builder expression α. Therefore,

$$X = \{\pi([\![\alpha]\!](\bar{a}\bar{b})) : \pi \text{ is } \bar{a}\text{-automorphism}\}.$$

Since $[\![\alpha]\!]$ is equivariant, it commutes with atom automorphisms, and thus

$$X = \{[\![\alpha]\!](\bar{a}\bar{c}) : \bar{c} \in Y\} \qquad \text{where } Y = \{\pi(\bar{a}\bar{b}) : \pi \text{ is an } \bar{a}\text{-automorphism}\}$$

To show that X is definable, it suffices to show that Y can be defined by a first-order formula over the atoms, with free variables $\bar{x}\bar{y}$ and constants from $\bar{a}$. This follows from the observation that Y is supported by $\bar{a}$ and Corollary 4.4. □

Exercise 60. Assume that the atoms are oligomorphic. Consider a language that is recognised by a definable nondeterministic automaton as in Exercise 58. Show that if the language is supported by a tuple of atoms $\bar{a}$, then it is also recognised by a definable nondeterministic automaton which is supported by $\bar{a}$.

Exercise 61. Consider the equality atoms. Consider an input alphabet as for data words, i.e. of the form $\Gamma = \mathbb{A} \times \Sigma$ where Σ is a finite (definable) set. Show that a language $L \subseteq \Gamma^*$ is recognised by a nondeterministic register automaton if and only if it is equivariant and recognised by a definable automaton.

Bibliographic notes for Section 4. Definable sets are based on set builder notation from set theory. The idea to use set builder notation as a way of representing hereditarily orbit finite sets (Theorem 4.2) originates from the programming language LOIS (*Looping over Infinite Sets*) [37, 38]. Earlier papers on sets with atoms, such as [12, 9], used different representations for orbit finite sets based on least supports (see Section 5). The name "definable" is inspired by terminology from model theory, where "definable set" means a set of tuples in a model that can be defined by a first-order formula with free variables, see [31, historical remarks on p. 82]. As mentioned in the bibliographical notes for Section 3, Lemma 4.3 is part of a theorem proved independently proved by Engeler, Ryll-Nardzewski and Svenonius, see [31, Theorem 7.3.1].

The idea to use orbit-finite sets (equivalently, definable sets) to model register automata, as is done in Excercises 58, 60 and 61 is from [7, 11, 12], and was one of the main motivations in developing the theory of sets with atoms. We will come back to automata in Section 8.

5 Least supports

In this section we show that in the equality atoms, one can always find a minimal (with respect to inclusion) support, i.e. there exist least supports. We then use least supports to get a classification of orbit-finite sets in the equality atoms. The classification says that every equivariant single orbit-finite set is isomorphic to a set obtained by taking non-repeating tuples of atoms of some dimension, and quotienting the tuples by some group acting on the coordinates.

5.1 Least supports

We say that a set with atoms x is supported by a finite set of atoms $\{a_1, \ldots, a_n\}$ if x is supported by the tuple $(a_1, \ldots, a_n)$. It is easy to see that this definition is well formed, i.e. it does not depend on the ordering of the set.

Theorem 5.1 (Least Support Theorem) *Consider the equality atoms. For every set with atoms x there is a least, with respect to inclusion, finite set of atoms supporting x.*

We say an atom automorphism fixes a set if it is the identity when restricted to that set.

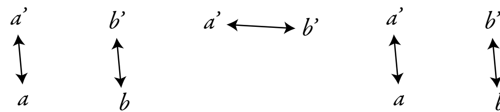
Lemma 5.2 *Consider the equality atoms. Let S, T be finite sets of atoms. Every atom automorphism π which fixes $S \cap T$ can be presented as a composition*

$$\pi = \pi_1 \circ \cdots \circ \pi_n$$

such that each π_i is an atom automorphism that fixes either S or T.

Before proving the lemma, let us remark how it proves the Least Support Teorem. To prove the theorem, it suffices to show that finite sets of atoms supporting x are closed under intersection. Suppose then that x is supported by S and also supported by T. To prove that it is also supported by $S \cap T$, we need to show that every atom automorphism that fixes $S \cap T$ also fixes x. By the lemma, it follows that every atom automorphism that fixes $S \cap T$ can be decomposed as a finite composition of atom automorphisms which fix x, and therefore it also fixes x.
Proof (of Lemma 5.2)
Let us prove the lemma in several steps.

1. *Transpositions.* Suppose first that π from the assumption of the lemma is a transposition, i.e. it swaps two atoms a, b. Choose some atoms a', b' which are not in $S \cup T$. Swapping a, b is the same as performing the following sequence of transpositions:

   By the assumption that a, b are not in $S \cap T$ and a', b' are not in $S \cup T$, each of the above transpositions fixes either S or T.

2. *Finite permutations.* An automorphism (i.e. permutation) of the atoms is called finite if it moves finitely many atoms. Every finite permutation is a finite composition of transpositions, and thus the previous item implies that the conclusion of the lemma is also true when π is a finite permutation.

3. *Infinite cycles.* Suppose that π is an infinite cycle, as in the following picture:

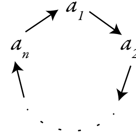
   $$\cdots \longrightarrow a_{-1} \longrightarrow a_0 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow \cdots$$

   We do not assume that the cycle contains all atoms. Since $S \cup T$ is finite, up to renumbering we can assume that there is some n such that elements from $S \cup T$ can appear only in $\{a_2, \ldots, a_n\}$. If we compose π with the transposition

$$a_1 \longleftrightarrow a_{n+1}$$

, then we get the permutation consisting of two cycles (one finite, one infinite) as in the following picture:

$$\cdots \longrightarrow a_{-1} \longrightarrow a_0 \longrightarrow a_{n+1} \longrightarrow a_2 \longrightarrow \cdots$$



The permutations drawn in blue fix $S \cup T$. Therefore, we have shown that the infinite cycle π is a composition of two permutations that fix $S \cup T$, and one finite cycle. To the finite cycle we can apply the previous item.

4. *General case.* Every permutation can be decomposed into independent cycles, some finite and some infinite. Both types of cycles were dealt with in the previous items. We only need to apply the construction to the finitely many cycles that contain atoms from $S \cup T$.

□

Exercise 62. Show that the atoms $(\mathbb{Q}, <)$ also have least supports.

Exercise 63. Show an example of oligomorphic atoms without least supports.

Exercise 64. Consider the equality atoms. Show that if a group is orbit-finite, then it is finite.

5.2 A representation theorem.

We use the Least Support Theorem to get a classification of orbit-finite equivariant sets in the equality atoms, up to equivariant bijections. Let us write $\mathbb{A}^{(n)}$ for the set of non-repeating n-tuples of atoms. This is a single-orbit set with atoms. To a tuple in $\mathbb{A}^{(n)}$ we can apply a permutation g of $\{1, \ldots, n\}$ as follows

$$g(a_1, \ldots, a_n) = (a_{g(1)}, \ldots, a_{g(n)})$$

Note that we have two groups acting on $\mathbb{A}^{(n)}$: atom automorphisms and permutations of the coordinates. These actions commute with each other in the following sense: if π is an atom automorphism and g is a permutation of the coordinates then

$$g(\pi(a_1, \ldots, a_n)) = \pi(g(a_1, \ldots, a_n)).$$

If G is a subgroup of all permutations of $\{1, \ldots, n\}$, then we define

$$\mathbb{A}^{(n)}/_G$$

to be $\mathbb{A}^{(n)}$ modulo the equivalence relation which identifies two tuples if they are in the same orbit with respect to the action of G. The following theorem shows that this is essentially the only possible construction for sets with atoms.

Theorem 5.3 (Classification of orbit-finite sets) *Consider the equality atoms. Every equivariant orbit-finite set with atoms is isomorphic (where isomorphisms are equivariant bijections) to a disjoint union of sets of the form*

$$\mathbb{A}^{(n)}/_G$$

where n is a natural number and G is a subgroup of theset $1, \ldots, n$.

Proof
It suffices to prove the theorem for equivariant sets which consist of one $\emptyset$-orbit, because every equivariant set with atoms is isomorphic to the disjoint union of its $\emptyset$-orbits. Let then X be a single $\emptyset$-orbit. Choose some $x \in X$. By the Least Support Theorem, there is some least support of x, let it be $\{a_1, \ldots, a_n\}$. Consider the relation

$$\{(\pi(a_1, \ldots, a_n), \pi(x)) : \pi \text{ is an atom automorphism}\} \subseteq \mathbb{A}^{(n)} \times X$$

which is equivariant by definition. Because $a_1, \ldots, a_n$ supports x, the above is actually an equivariant function, call it

$$f : \mathbb{A}^{(n)} \to X.$$

Consider the inverse image $f^{-1}(x)$, and define G to be the set of permutations g of $\{1, \ldots, n\}$ such that

$$f(g(a_1, \ldots, a_n)) = f(a_1, \ldots, a_n)$$

The set G is easily seen to be closed under composition and inverse, i.e. it is a subgroup of all permutations.

We show below that every tuples $\bar{b}, \bar{c} \in \mathbb{A}^{(n)}$ satisfy

$$f(\bar{b}) = f(\bar{c}) \qquad \text{iff} \qquad \bar{c} = g(\bar{b}) \text{ for some } g \in G \tag{4}$$

The above equivalence implies that X is isomorphic to $\mathbb{A}^{(n)}/_G$ and thus finishes the proof.

Let us first show the right-to-left implication in (4). Assume that $\bar{c} = g(\bar{b})$ holds for some $g \in G$. Let π be some permutation of the atoms mapping $\bar{a}$ to $\bar{b}$, which

exists because there is only one orbit of non-repeating tuples of given dimension.

$$\begin{array}{rcl} f(\bar{b}) & = & \text{(choice of } \pi) \\ f(\pi(\bar{a})) & = & \text{(equivariance of } f) \\ \pi(f(\bar{a})) & = & \text{(definition of } G) \\ \pi(f(g(\bar{a}))) & = & \text{(equivariance of } f) \\ f(\pi(g(\bar{a}))) & = & \text{(automorphisms and } G \text{ commute)} \\ f(g(\pi(\bar{a}))) & = & \text{(choice of } \pi) \\ f(g(\bar{b})) & = & \text{(choice of } g) \\ f(\bar{c}) & & \end{array}$$

We now show the left-to-right implication in (4). Assume that $f(\bar{b}) = f(\bar{c})$. Let π be some permutation of the atoms which maps $\bar{b}$ to $\bar{a}$. By equivariance of f,

$$f(\bar{a}) = f(\pi(\bar{b})) = \pi(f(\bar{b})) = \pi(f(\bar{c})) = f(\pi(\bar{c})).$$

Since f is equivariant, every input supports its output, and therefore the tuple $\pi(\bar{c})$ supports $f(\bar{a})$. By the Least Support Theorem, the atoms that appear in the tuple $\pi(\bar{c})$ must be $\{a_1, \ldots, a_n\}$, i.e. there must be some permutation g of coordinates such that

$$\pi(\bar{c}) = g(\bar{a})$$

Since $\pi(\bar{c})$ has the same image under f as $\bar{a}$, it follows that $g \in G$. Therefore,

$$\bar{c} = \pi^{-1}(g(\bar{a})) = g(\pi^{-1}(\bar{a})) = g(\bar{b})$$

which completes proof of (4). □

Bibliographic notes for Section 5. The Least Support Theorem was proved in [29, Proposition 3.4]. A generalisation of this theorem, for other kinds of atoms, kind be found in [12, Section 10]. The classification result from Theorem 5.3 is from [12, Theorem 10.17], although a similar construction can be found in [27, Definition 2]. Exercise 5.1 is based on [24, Lemma 2.14].

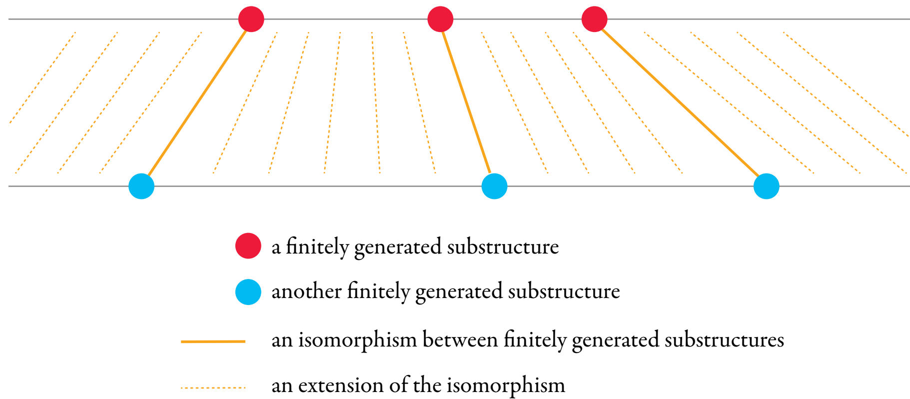
6 Homogeneous atoms

So far, the basic restriction on atom structures was oligomorphism. Oligomorphism implies that orbit-finiteness is a robust concept (i.e. it does not depend on the support of the orbits), and that orbits can be defined in first-order logic. A drawback is that it is not clear how to find oligomorphic structures. In this section, we study the notion of a homogeneous, which is a stronger property than being oligomorphic. In homogeneous structures, first-order logic collapses to its quantifier-free fragment. More importantly, every homogeneous structure is obtained by a taking a limit of a class of finite structures closed under amalgamation. This gives a way of producing homogeneous structures, taking the limit of classes such as: the class of all finite total orders, the class of all directed graphs, the class of all equivalence relations, etc.

Homogeneous structures. In this section we study atom structures over finite vocabularies, possibly including functions. An *(induced) substructure* of a structure $\mathfrak{A}$ is defined to be a structure obtained from $\mathfrak{A}$ by restricting the universe to some subset that is closed under applying functions from the vocabulary. Since we only talk about induced substructures, we simply omit the word induced. The substructure *generated* by a subset of the universe is defined to be the smallest substructure which contains the subset. A *finitely generated* substructure is defined to be one generated by a finite subset of the universe. When the vocabulary has no function symbols, then a finitely generated substructure is obtained by restricting the universe to an arbitrary finite subset.

Definition 6.1 *A structure is called* homogeneous *if every isomorphism between finitely generated substructures extends to a full automorphism.*

Example 24. The empty set of atoms, the equality atoms, and $(\mathbb{Q}, <)$ are all homogeneous structures over finite vocabularies. The proof for $(\mathbb{Q}, <)$ is in this picture



In Theorem 6.3, we will show that if a structure is homogeneous, and satisfies a further condition that is true whenever the vocabulary uses no functions, then it is oligomorphic. This explains why the structures mentioned above are oligomorphic. □

Example 25. Consider the powerset of the natural numbers

$$(\mathsf{P}(\mathbb{N}), \cup),$$

with the union function. This structure is not homogeneous, because $\emptyset \mapsto \{1\}$ is a finite partial automorphism which does not extend to a full automorphism. □

Example 26. [Integers are or are not homogeneous, depending on vocabulary] Whether or not a structure is homogeneous depends on the choice of predicates

in the vocabulary, and not just the automorphism group. Consider the structures $(\mathbb{Z}, \leq)$ and $(\mathbb{Z}, +1)$, with $+1$ being the successor function. The two structures have the same automorphism group, namely the translations. The first structure is not homogeneous, because the function that maps 0 to 0 and 1 to 2 is an isomorphism between finitely generated substructures. The second structure is homogeneous, because finitely generated substructures are half-intervals of the form $\{i, i+1, \ldots\}$. □

Example 27. [Bit vectors] We use the name *bit vector* for an infinite sequence of zeroes and ones which has finitely many ones. By ignoring trailing zeroes, a bit vector can be represented as a finite sequence such as 00101001. Define the *bit vector structure* to be the bit vectors equipped with a binary function that adds modulo two.

The bit vectors are actually a linear space over the two element field. The dimension of this linear space is countably infinite, an example basis consists of bit vectors which have a 1 on the n-th coordinate: $1, 01, 001, 0001, \ldots$. Another example of a basis is $1, 11, 111, 1111, \ldots$.

The substructure generated by a set of bit vectors is all possible linear combinations of those vectors. We show that the bit vector structure is homogeneous. What is a finitely generated substructure? This is a substructure generated by a finite set of linearly independent bit vectors. What is a finite partial automorphism? This is an arbitrary bijection between two finite sets of linearly independent bit vectors, extended homomorphically to their linear combinations. What is a full automorphism? This is an arbitrary bijection between two bases, extended homomorphically to their linear combinations. It follows that every finite partial automorphism extends to a full automorphism: use the Steinitz exchange lemma to extend the bijection of two finite linearly independent sets to a bijection of bases. □

6.1 Oligomorphism and quantifier elimination

Recall that in Example 26, we showed that the structure $(\mathbb{Z}, +1)$ is homogeneous. Recall from Example 14 that this structure is not oligomorphic. Therefore, homogeneous does not imply oligomorphic. In this section we show that the implication is true under a mild additional assumption on the functions in the structure (in particular, the mild assumption is true whenever there are no functions). Under this same assumption, we show that first-order logic collapses to its quantifier-free fragment.

Blowups. The *blowup* of a structure is the function which maps $n \in \mathbb{N}$ to the biggest size of a substructure that is generated by n elements. A structure is said to have *unbounded blowup* if the blowup maps some n to ∞.

Example 28. [Blowups] Every structure without functions has blowup $n \mapsto n$, because only functions can add elements. The blowup of $(\mathbb{Z}, +1)$ maps 1 to ∞, because every integer generates an infinite set. In the powerset of the natural numbers from Example 25, the blowup is $n \mapsto 2^n$ because n distinct singletons generated the powerset of their union. □

Example 29. Since a vector can be used 0 or 1 times in a linear combination, it follows that the blowup for the bit vector structure from Example 27 is $n \mapsto 2^n$. In particular, the bit vector structure has (exponentially) bounded blowup. □

Lemma 6.2 *Assume that the atoms are homogeneous. Two tuples of atoms satisfy the same quantifier-free formulas with constants from $\bar{a}$ if and only if they are in the same $\bar{a}$-orbit.*

Proof
For the right-to-left implication, the truth-value of quantifier-free formulas with constants from $\bar{a}$ is preserved under $\bar{a}$-automorphisms. Conversely, if two tuples of atoms satisfy the same quantifier-free formulas with constants from $\bar{a}$, then one can build an isomorphism between the substructures generated by them, which extends the identity on $\bar{a}$. By definition of homogeneous structures, this extends to a full automorphism, and therefore the tuples are in the same $\bar{a}$-orbit. □

Theorem 6.3 *Assume that the atoms are homogeneous, over a finite vocabulary, and have bounded blowup. Then the atoms are oligomorphic. Furthermore, a set of n-tuples of atoms is $\bar{a}$-supported if and only if it is definable by a quantifier-free formula with constants from $\bar{a}$.*

Proof
Let us first show oligomorphism. From the assumption on bounded blowup, there is some k such that substructures of the atoms with n generators have size at most k. It follows by a pumping argument that if an atom is generated from n atoms, then it is generated by a term of height at most k. Since the vocabulary is finite, there are finitely many terms of height at most k. It follows that, up to logical equivalence, there are only finitely many quantifier-free formulas with n variables. By Lemma 6.2, there are finitely many equivariant orbits of n-tuples, which proves oligomorphism.

Consider now the "Furthermore" part. The right-to-left implication is immediate. For the left-to-right implication, we use oligomorphism to show that an $\bar{a}$-supported set of n-tuples of atoms must necessarily contain finitely many $\bar{a}$-orbits, and each such orbit is definable by a quantifier free formula thanks to Lemma 6.2. □

A further corollary of Theorem 6.3 is that in structures which satisfy its assumptions, every formula of first-order logic is equivalent to a quantifier-free formula. This is because the truth value of formulas of first-order logic is preserved under full automorphisms, and therefore the set of tuples defined by a formula is equivariant, and therefore quantifier-free definable by Theorem 6.3. The same proof works also for richer logics, such as higher-order logics, and for formulas which contain certain atoms as constants.

Exercise 65. Show that if the atoms are oligomorphic, then they have bounded blowup.

6.2 The Fraïssé limit

We have seen some examples of homogeneous atoms, such as the equality atoms, $(\mathbb{Q}, <)$ or the bit vector atoms. In this section we present a construction, called the *Fraïssé limit*, which inputs a class of finitely generated structures, and produces a single (usually infinite) homogeneous structure, called its Fraïssé limit. Actually, the Fraïssé limit is the only possible way of producing a countable homogeneous structure, since we shall see that every homogeneous structure is the Fraïssé limit of its finitely generated substructures.

The age of a homogeneous structure. The *age* of a structure is defined to be its finitely generated structures (and isomorphic structures). The following theorem shows that a countable homogeneous structure is uniquely determined by its age.

Theorem 6.4 *Countable homogeneous structures with the same ages are isomorphic.*

We begin with a lemma which gives the only property of homogeneous structures that is used in the proof of the theorem. An embedding is an isomorphism of one structure with a substructure of another one.

Lemma 6.5 *Let $\mathfrak{A}$ be in the age of a homogeneous structure $\mathfrak{H}$. Every partial isomorphism between substructures of $\mathfrak{A}$ and $\mathfrak{H}$ extends to an embedding of $\mathfrak{A}$ in $\mathfrak{H}$.*

Proof
Let f be the partial isomorphism, and let $\mathfrak{B}$ be the substructure of $\mathfrak{A}$ on which the partial isomorphism is defined. Since $\mathfrak{A}$ is in the age, there is some embedding $g : \mathfrak{A} \to \mathfrak{H}$. These functions are illustrated in the following diagram:

$$\begin{array}{ccc} \mathfrak{B} & \xrightarrow{id} & \mathfrak{A} \\ f\downarrow & & g\downarrow \\ \mathfrak{H} & & \mathfrak{H} \end{array}$$

By following f^{-1} and then g, we get a partial automorphism of $\mathfrak{H}$. By homogeneity, this partial automorphism extends to a full automorphism, which makes the following diagram commute.

$$\begin{array}{ccc} \mathfrak{B} & \xrightarrow{id} & \mathfrak{A} \\ f\downarrow & & g\downarrow \\ \mathfrak{H} & \xrightarrow{\pi} & \mathfrak{H} \end{array}$$

The extension required by the lemma is then g followed by π^{-1}. □

Theorem 6.4 follows from the following lemma.

Lemma 6.6 *Let $\mathfrak{H}_1, \mathfrak{H}_2$ be countable structures with the same age, which both have the property from Lemma 6.5. Then every partial isomorphism between finitely generated substructures of $\mathfrak{H}_1$ and $\mathfrak{H}_2$ extends to a full isomorphism.*

Proof
We prove the following claim:

> Consider a partial isomorphism f between finitely generated substructures of $\mathfrak{H}_1$ and $\mathfrak{H}_2$. For every element a of either $\mathfrak{H}_1$ or $\mathfrak{H}_2$, the partial isomorphism can be extended to be defined also on a.

Let f and a be as in the claim. Without loss of generality, assume that a is in $\mathfrak{H}_1$. Let $\mathfrak{A}$ be the substructure of $\mathfrak{H}_1$ generated by a plus all the elements where f is defined. Since $\mathfrak{A}$ is in the age of $\mathfrak{H}_1$, it is also in the age of $\mathfrak{H}_2$. The function f is a partial isomorphism between $\mathfrak{A}$ and $\mathfrak{H}_2$, and therefore it extends by Lemma 6.5 to an embedding of $\mathfrak{A}$ in $\mathfrak{H}_2$, which is the extension required by the claim.

The lemma follows from the claim by a zig-zag construction. Define inductively a sequence of partial isomorphisms between finite substructures of $\mathfrak{H}_1$ and $\mathfrak{H}_2$, such that the next one extends the previous one, every element of both structures appears eventually in the source or target of the partial isomorphisms. The full isomorphism is then the limit of these partial isomorphisms. □

This completes the proof of Theorem 6.4.

Exercise 66. Consider a countably infinite directed graph, where every edge is chosen independently with equal probability one half. Show that with probability one, we get the same graph up to isomorphism, call this graph *the random directed graph*.

6.3 Amalgamation

Since a countable homogeneous structure is uniquely identified by its age, it is natural to ask: which classes of finite structures are ages of countable homogeneous structures? The special property which distinguishes the age of a homogeneous structure, among other classes of finitely generated structures, is amalgamation, which is described below. An *instance of amalgamation* consists of two embeddings with a common source:

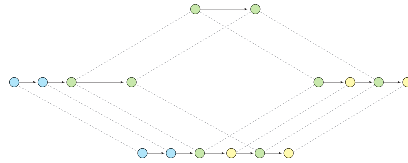
$$\begin{array}{ccccc} & & \mathfrak{A} & & \\ & \swarrow^{f_1} & & \searrow^{f_2} & \\ \mathfrak{B}_1 & & & & \mathfrak{B}_2 \end{array} \tag{5}$$

A *solution* of the instance consists of a structure $\mathfrak{C}$ and two embeddings g_1, g_2 such that the following diagram commutes

$$\begin{array}{ccccc} & & \mathfrak{A} & & \\ & \swarrow^{f_1} & & \searrow^{f_2} & \\ \mathfrak{B}_1 & & & & \mathfrak{B}_2 \\ & \searrow_{g_1} & & \swarrow_{g_2} & \\ & & \mathfrak{C} & & \end{array} \tag{6}$$

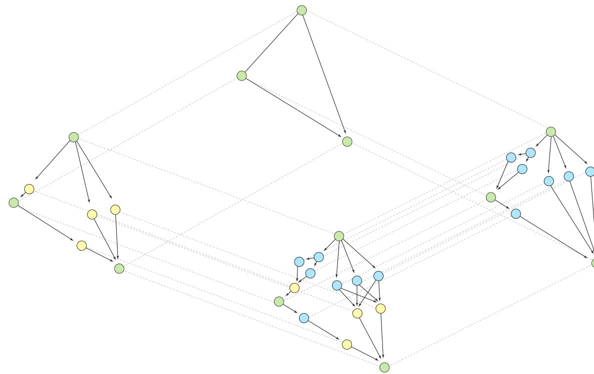
We say a class of structures is *closed under amalgamation* if every instance of amalgamation which uses structures from the class has a solution which also uses a structure from the class.

Example 30. Consider the class of finite total orders. This class is closed under amalgamation. Here is an example of an instance and solution of amalgamation:



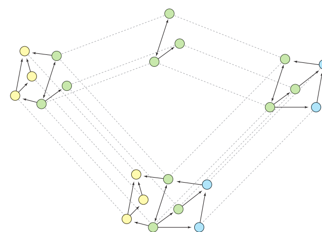
□

Example 31. Consider the class of all finite partial orders, not necessarily total orders. This class is closed under amalgamation. Here is an example of an instance and solution of amalgamation:



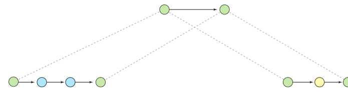
One way of amalgamating two partial orders, which is illustrated in the picture above, is to put the elements of the left (yellow) order after the elements of the right (blue) order, as long as they have the same relationship with the common (green) elements. Other strategies lead to other amalgamations. □

Example 32. Consider the class of all finite directed graphs. This class is closed under amalgamation. An instance of amalgamation is basically two directed graphs that have a common induced subgraph. One way of amalgamating them looks like this:



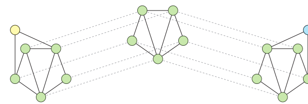
The amalgamation above is not unique; one could also add edges between the blue and yellow nodes. This shows that, for instance, the class of all cliques is closed under amalgamation. □

Example 33. Consider the class of finite directed graphs, where the edge relation is a partial successor, i.e. all vertices have out-degree and in-degree at most one, and no loops. The class is not closed under amalgamation, here is an instance without a solution:



□

Example 34. Consider the class of (undirected) finite planar graphs. Undirected edges are modelled by saying that the binary predicate is symmetric. The class is not closed under amalgamation, here is an instance without a solution:



□

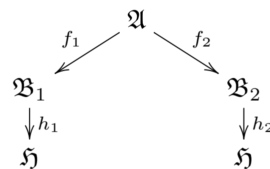
The following theorem shows that amalgamation characterises uniquely the ages of countable homogeneous structures.

Theorem 6.7 *Let $\mathscr{A}$ be a class of finitely generated structures over a common vocabulary, which is closed under isomorphism and substructures. Then $\mathscr{A}$ is the age of a countable homogeneous structure if and only if it is closed under amalgamation.*

The two implications of the theorem are proved in Lemmas 6.8 and 6.9.

Lemma 6.8 *The age of a countable homogeneous structure is closed under amalgamation.*

Proof
Let $\mathfrak{H}$ be a homogeneous structure. Consider an instance of amalgamation as in (5). Because the structures are in the age of $\mathfrak{H}$, there are embeddings h_1, h_2 as in the following diagram:



The diagram distinguishes the targets of h_1, h_2 because the embeddings $h_1 \circ f_1$ and $h_2 \circ f_2$ need not be the same embedding of $\mathfrak{A}$ in $\mathfrak{H}$. However, the images of both of these embeddings are isomorphic substructures of $\mathfrak{H}$; and by homogeneity there is a full automorphism π which extends this partial automorphism. In other words, the following diagram commutes:

$$
\begin{array}{ccccc}
 & & \mathfrak{A} & & \\
 & \swarrow^{f_1} & & \searrow^{f_2} & \\
\mathfrak{B}_1 & & & & \mathfrak{B}_2 \\
\downarrow h_1 & & & & \downarrow h_2 \\
\mathfrak{H} & & \xrightarrow{\pi} & & \mathfrak{H}
\end{array}
$$

The above diagram shows a solution of amalgamation. □

Lemma 6.9 *If a class of finitely generated structures is closed under isomorphism, substructures and amalgamation, then it is the age of a countable homogeneous structure.*

Proof

Let $\mathscr{A}$ be a class of structures satisfying the assumptions of the lemma. We define a sequence of finitely generated structures $\mathfrak{H}_1, \mathfrak{H}_2, \ldots \in \mathscr{A}$ with the following property:

- For every $n \in \mathbb{N}$, the structure $\mathfrak{H}_n$ is a substructure of $\mathfrak{H}_{n+1}$.
- Let $n \in \mathbb{N}$. Suppose that $\mathfrak{A}$ is a substructure of both $\mathfrak{H}_n$ and some $\mathfrak{B} \in \mathscr{A}$. If $\mathfrak{B}$ is generated by at most n elements, then $\mathfrak{B}$ embeds into $\mathfrak{H}_{n+1}$ in a way consistent with $\mathfrak{A}$, i.e. there is some f such that following diagram commutes:

$$
\begin{array}{ccccc}
 & & \mathfrak{A} & & \\
 & \swarrow^{id} & & \searrow^{id} & \\
\mathfrak{H}_n & & & & \mathfrak{B} \\
 & \searrow_{id} & & \swarrow_{f} & \\
 & & \mathfrak{H}_{n+1} & &
\end{array}
$$

The sequence is defined by induction. To obtain $\mathfrak{H}_{n+1}$ from $\mathfrak{H}_n$, we apply amalgamation to all possible embeddings of substructures of $\mathfrak{H}_n$ into structures from $\mathscr{A}$ which have generating sets of size at most n.

Define $\mathfrak{H}$ to be the limit (i.e. union) of the sequence $\mathfrak{H}_1, \mathfrak{H}_2, \ldots$. We claim that $\mathfrak{H}$ is homogeneous. The age of $\mathfrak{H}$ is clearly $\mathscr{A}$, because every structure with at most n generators embeds into $\mathfrak{H}_{n+1}$. By definition $\mathfrak{H}$ satisfies the property from Lemma 6.5. By Lemma 6.6 applied to the case of $\mathfrak{H}_1 = \mathfrak{H}_2 = \mathfrak{H}$, it follows that every finite partial automorphism of $\mathfrak{H}$ extends to a full isomorphism. □

This completes the proof of Theorem 6.7.

The countable homogeneous structure which is constructed in Lemma 6.9 is called the *Fraïssé limit*. The Fraïssé limit is the inverse operation of the age. Applying the Fraïssé limit to various classes from Examples 30, 31 and 32, we get several homogeneous structures.

- The Fraïssé limit of all directed total orders is $(\mathbb{Q}, <)$.
- There is a Fraïssé limit of all partial orders. This partial order is not easy draw. It contains as isomorphic copies the rational numbers, infinite antichains, and the infinite binary tree.
- There is a Fraïssé limit of all directed graphs.

Exercise 67. Show that the property from Exercise 56 holds also for Fraïssé limit of all undirected graphs. (The same is true for directed graphs.)

Exercise 68. Assume a finite relational vocabulary. Suppose that $\mathscr{A}$ is a class of structures that satisfies the assumptions of Theorem 6.7, and let $\mathbb{A}$ be its Fraïssé limit. Show that if membership in $\mathscr{A}$ is decidable, $\mathbb{A}$ is an effective structure.

Exercise 69. Define *monadic second-order logic* (MSO) to be the extension of first-order logic where one can also quantify over sets of vertices. A famous result on MSO is Rabin's Theorem, which says that the structure $\{0,1\}^*$ equipped with functions $x \mapsto x0$ and $x \mapsto x1$ has decidable MSO theory, i.e. one can decide if a sentence of MSO is true in it. Show that $(\mathbb{Q}, <)$ has decidable MSO theory.

Exercise 70. Define a *weak tree* to be a partially ordered set where for every x, the set $\{y : y < x\}$ is totally ordered. Show that the class of finite trees is not closed under amalgamation.

Exercise 71. Define a *tree* to be a weak tree in the sense of Exercise 70, together with a binary function which maps two nodes to their closest common ancestor. Show that the class of trees is closed under amalgamation. We use the name *random tree* for the Fraïssé limit of this class.

Exercise 72. Assume that the atoms is the random tree described in Exercise 71. Find a finitely supported equivalence relation on the atoms which has infinitely many infinite equivalence classes.

Exercise 73. Assume that the atoms is the random tree described in Exercise 71. Show that one cannot find an infinite equivariant set X and an equivariant relation on it which is a total dense order. Equivariance is important here, since if we only want a finitely supported one then this is easily accomplished by taking the path connecting some two atoms $a < b$, and using the order inherited from the atoms.

Exercise 74. Show that the random tree from Exercise 71 has decidable MSO theory.

Exercise 75. Consider the random directed graph, i.e. the Fraïssé limit of all finite directed graphs. Show that for every MSO formula with free variables $x_1, \ldots, x_n$ that represent elements (not sets of elements) there is an equivalent formula of first-order logic, but there is no algorithm which computes such equivalent formulas.

Exercise 76. If Σ is a finite alphabet. We model a word $w \in \Sigma^*$ as a structure, where the universe is positions in w, there is a binary predicate $<$ for the order relation, and there are unary predicates $a(x)$ for the labels. We denote the vocabulary used for this structure by $\Sigma_<$. Show that for every regular language $L \subseteq \Sigma^*$ there is a homogeneous structure $\mathfrak{A}$ over a vocabulary containing $\Sigma_<$ such that the age of $\mathfrak{A}$ after restricting to $\Sigma_<$ is exactly the structures corresponding to L.

Bibliographic notes for Section 6. The Fraïssé limit and homogeneous structures are a basic notion in model theory, see e.g. [31, Section 7]. The idea to use a homogeneous structure as the atoms, and the consider orbit-finite sets over those atoms, is from [12]. For an introduction to MSO and Rabin's Theorem as used in Exercise 69, see [53, Theorem 6.8]. Exercise 76 is essentially [17, Proposition 2].

Part III

Models of computation

In this part, we discuss models of computation: automata, including pushdown automata, Turing machines, and while programs. The general theme is as follows: we take a classical definition, such as the definition of a nondeterministic finite automaton, and then replace "finite set" by "definable set" or equivalently "hereditarily orbit-finite set". We then see which constructions work, and which ones fail.

7 Computable functions on definable sets

In this section we discuss how algorithms can manipulate definable sets. For example, in Section 8, we will introduce various models of automata where the states, transitions, etc. are definable sets. But what is the point of studying automata if one cannot run algorithms on them, such as testing for emptiness or minimisation? The goal of this section is to define what it means for an operation on definable sets to be computable. In these notes, we only care about computability in finite time – which is already non-trivial for infinite sets. Issues of computational complexity are ignored.

7.1 Effective structures

Our notion of computability for definable sets is straightforward: intuitively speaking, an operation on definable sets is computable if it can be computed assuming that definable sets are represented by set builder expressions. To talk about algorithms manipulating set builder expressions, we will use the following assumption on the atom structure.

Definition 7.1 (Effective structure) *Call a structure* $\mathbb{A}$ effective *if its universe is a decidable subset of* 2^**, its vocabulary is finite, and the following problem is decidable:*

- **Input** *A sentence of first-order logic, using the vocabulary of* $\mathbb{A}$ *extended with a constant for each element in the universe of* $\mathbb{A}$.
- **Question.** *Is the sentence true in* $\mathbb{A}$*?*

We say that a structure has *decidable first-order theory* if one can decide its theory, i.e. those sentences over the vocabulary of the structure which are true in it. The notion of effectivity for a structure, as defined in Definition 7.1, is a little stronger than having a decidable first-order theory, since we also allow constants in the first-order sentences. A countable structure $\mathbb{A}$ is isomorphic to an effective one if and only if one can find constants $c_1, c_2, \ldots$ such that $(\mathbb{A}, c_1, c_2, \ldots)$ has decidable first-order theory and every element of $\mathbb{A}$ is represented by some constant.

Example 35. The equality atoms $(\mathbb{N}, =)$ and the ordered rational numbers are isomorphic to effective structures. Other examples of an structures isomorphic to effective ones are: Presburger arithmetic $(\mathbb{N}, +)$ and Skolem arithmetic $(\mathbb{N}, \cdot)$. In the sequel, we will gloss over this isomorphism, and say things like: $(\mathbb{N}, =)$ is an effective structure. □

If $\mathbb{A}$ is an effective structure, then the atoms or set builder expressions which use atom parameters can be written down as bit strings and processed by algorithms (e.g. formalised as Turing machines). This motivates the following definitions, which are stated in terms of partial functions to to allow modelling programs that need not terminate.

Definition 7.2 (Computable function on definable sets) *Assume that the atoms* $\mathbb{A}$ *are effective. We write* $\mathsf{setbuil}\mathbb{A}$ *for the set builder expressions over* α *without free variables, and* $\mathsf{def}\,\mathbb{A}$ *for definable sets. A partial function*

$$\mathsf{setbuil}\mathbb{A} \xrightarrow{f} \mathsf{setbuil}\mathbb{A}$$

is called computable *if there is a Turing machine which inputs a set builder expression* α *(assuming that the atom parameters are represented as bit strings according to the assumption on* $\mathbb{A}$ *being effective) and outputs* $f(\alpha)$*, or does not terminate if* $f(\alpha)$ *is undefined. A partial function* g *from definable sets to definable sets is called* computable *if the following diagram commutes*

$$\begin{array}{ccc} \mathsf{setbuil}\mathbb{A} & \xrightarrow{f} & \mathsf{setbuil}\mathbb{A} \\ {\scriptstyle [\![_]\!]}\downarrow & & \downarrow{\scriptstyle [\![_]\!]} \\ \mathsf{def}\,\mathbb{A} & \xrightarrow{g} & \mathsf{def}\,\mathbb{A} \end{array}$$

for some computable partial function f *on set builder expressions.*

In the spirit of the above definition, we can also talk about computable yes/no properties of definable sets (such as emptiness), or about computable operations from pairs of definable sets to definable sets (such as intersection or set difference). Formally speaking, these are not new notions: a yes/not output can be represented by viewing the Booleans as definable sets via

$$\text{no} = \emptyset \qquad \text{yes} = \{\emptyset\},$$

while a pair of definable sets can be viewed as a single definable set via Kuratowski pairing. In the next section, we show that the most basic operations like testing for emptiness, Boolean operations, taking pairs, projecting pairs, etc. are all computable.

Exercise 77. Call a structure $\mathbb{A}$ *effectively oligomorphic*[2] if it is oligomorphic and for every n one can compute a first-order formula with $2n$ free variables which defines the "same orbit" binary relation on $\mathbb{A}^n$. Show that if a structure is effectively oligomorphic and has decidable first-order theory, then it is isomorphic to an effective structure.

Exercise 78. Assume that $\mathbb{A}$ is oligomorphic and has decidable first-order model checking. Show that the following conditions are equivalent:

1. given $n \in \mathbb{N}$, one can compuate first-order formulas $\varphi_1, \ldots, \varphi_m$ in the vocabulary of $\mathbb{A}$ with n free variables each, which define, respectively, all $\emptyset$-orbits in $\mathbb{A}^n$;
2. given n, one can compute the number of $\emptyset$-orbits in $\mathbb{A}^n$;
3. $\mathbb{A}$ is effectively oligomorphic.

[2] Using forcing, one can show that there are oligomorphic structures which have decidable model checking, but which are not effectively oligomorphic [50].

7.2 Computability of basic operations

In this section, we show that basic operations on definable sets are computable.

We begin with testing membership and inclusion. Consider two set builder expressions α and β, which are allowed to have free variables. Whether or not the inclusion $\alpha \subseteq \beta$ holds depends on the valuation of the free variables. For example, if the atoms are $(\mathbb{Q}, <)$ then the following containment

$$\{y : \text{ for } y \text{ such that } 0 < y < x\} \quad \subseteq \quad \{y : \text{ for } y \text{ such that } y \neq y\}$$

holds if and only if the free variable x satisfies $x \leq 0$. The following lemma, which is an archetypical symbol pushing result, shows that the constraint on the free variables which makes inclusion hold can be defined in first-order logic.

Lemma 7.3 (Symbol Pushing Lemma) *Assume that the atoms are effective. Let α, β be set builder expressions with free variables contained in $\bar{x}$. The set of $\bar{x}$-tuples of atoms which satisfy the inclusion $\alpha \subseteq \beta$ is definable by a formula of first-order logic which can be computed based on α and β. (The formula can use constants from the atoms.) Likewise for $\in$ instead of $\subseteq$.*

Proof

In this proof, from the assumption on effectivity we only use the part which says that atoms can be written down as bit strings (and hence it is clear what it means to represent α, β and the resulting first-order formula); we do not use the assumption that the first-order theory is decidable.

Induction on the size of the set builder expressions.

Consider first $\subseteq$. The interesting case is when the left side is a set expression, i.e. axiomatising those $\bar{x}$-tuples which satisfy the inclusion

$$\{\alpha(\bar{x}\bar{y}) : \text{ for } \bar{y} \text{ such that } \varphi(\bar{x}\bar{y})\} \quad \subseteq \quad \beta$$

The inclusion is true if and only if for every $\bar{y}$ which makes $\varphi(\bar{x}\bar{y})$ true, we have

$$\alpha(\bar{x}\bar{y}) \in \beta(\bar{x}).$$

This can be formalised in first-order logic, using the induction assumption to get a constraint on the variables $\bar{x}\bar{y}$ which makes the membership true.

For membership $\in$ the interesting case is when the right side is a set expression:

$$\alpha \quad \in \quad \{\beta(\bar{x}\bar{y}) : \text{ for } \bar{y} \text{ such that } \varphi(\bar{x}\bar{y})\}$$

This membership is true if and only if there is some $\bar{y}$ which satisfies $\varphi(\bar{x}\bar{y})$ and

$$\alpha(\bar{x}) = \beta(\bar{x}\bar{y}).$$

If α and β are atom expressions, then the constraint for $\alpha = \beta$ is that the corresponding variables are equal. If α and β are not atom expressions, then equality is the same as inclusion both ways, which can be described using the induction assumption. □

Corollary 7.4 *Assume that the atoms are effective. Then equality, membership and inclusion for definable sets over $\mathbb{A}$ are computable.*

Exercise 79. Suppose that $\mathfrak{A}$ is a relational structure over a finite vocabulary Σ where the universe A and the interpretations of all relations are definable sets. Show that if $\varphi(x_1, \ldots, x_n)$ is a formula of first-order logic over vocabulary Σ, then

$$\{(a_1, \ldots, a_n) \in A^n : \mathfrak{A} \models \varphi(a_1, \ldots, a_n)\}$$

is a definable set which can be computed given $\mathfrak{A}$ and φ.

Exercise 80. Show that if X is a definable set, then

$$\{(x, y) : x, y \in X \text{ and } x \in y\}.$$

is a definable set which can be computed given X.

Exercise 81. Define the *union-member closure* of a set X to be the least (with respect to inclusion) set which contains X and such that if y is in the union-member closure, then $\cup y$ is in the union-member closure and every element of y is in the union-member closure. Show that the union-member closure of a definable set is a definable set, which can be computed given X.

Exercise 82. Show that if $R \subseteq X \times Y$ is definable, then

$$x \in X \quad \mapsto \quad \{y \in Y : (x, y) \in R\}.$$

is a function with a definable graph, which can be computed given R.

7.3 Definable while programs

In Definition 7.2, we proposed a notion of computability for functions from definable sets to definable sets. A problem with that definition is that it deals with the representation using set builder expressions at each step; leading to cumbersome constructions. The goal of this section is to present a programming language, which deals directly with sets with atoms. The representation issues are tackled once, when designing the programming language. An added bonus of this approach is that sometimes, when the issue at hand is simple enough, the resulting code in sets with atoms is exactly the same as in normal sets. This is the case, for instance, with graph reachability, and it will also be the case for automata problems that will be discussed in Section 8, such as minimisation or converting a pushdown automaton to a grammar.

Our programming language is called *definable while programs*. We use two assumptions on the atoms:

1. they are an effective structure in the sense of Definition 7.1;

2. they are effectively oligomorphic in the sense of Exercise 77.

For some results we only need the first assumption, in those cases we can use e.g. Presburger arithmetic for the atoms. Before defining the programming language, consider the following example, which uses the atoms $(\mathbb{Q}, <)$. These atoms satisfy both assumptions, although formally speaking they are effective only up to isomorphism. Consider the definable set of all bounded open intervals:

$$I = \{\{z : \text{ for } z \text{ such that } x < z < y\} : \text{for } x, y \text{ such that } x < y\}.$$

The following program (comments are in blue) computes all bounded open intervals which contain 0:

```
load some definable sets or atoms into  variables
X := I
Y := ∅
z := 0

run some code in parallel for every element of X
for x in X do
  if z ∈ x then
    Y := Y ∪ {x}
```

After executing the program above, variable `Y` stores all bounded open intervals which contain 0. The above example illustrates the two key properties of the programming language: variables store definable sets, and one can run a for loop in parallel for all elements in a definable set.

Syntax. We assume that there is a countably infinite set of variable names. The language is untyped: every variable stores a definable set, which might be an atom. Although cumbersome, one can encode other data structures using definable sets, e.g. the natural numbers can be encoded by

$$0 \stackrel{\text{def}}{=} \emptyset \qquad 1 \stackrel{\text{def}}{=} \{\emptyset\} \qquad \cdots \qquad n \stackrel{\text{def}}{=} \{0, 1, \ldots, n-1\}.$$

Of course a reasonable implementation would have more features, such as booleans or integer arithmetic. We present the programming language using a minimal syntax, so as to concentrate on the aspects of the language that deal with atoms. Below we describe the possible instructions in the language.

- **Assignment of definable sets.** For every definable set x, which might be an atom, and every variable `x`, one can write an assignment `x :=` x. This part of the syntax depends on the atom structure at hand, since the notion of definable set depends on the atoms. In order for the syntax to be effective, one needs an assumption that definable sets can be represented, e.g. this is true for effectively oligomorphic structures, or for structures over a finite vocabulary where the universe is $\{0, 1, \ldots\}$.

- **Adding to sets.** If x, y are variables one can write an instruction x := x $\cup$ {y}, which adds the content of y to the set stored in x. If x does not store a set but an atom, then the instruction aborts with failure.

- **Sequential composition** If I and J are already defined programs, then also I;J is a program which first executes I and then J.

- **Control flow.** Suppose that x and y are variables, and I is an already defined program, and δ is one of $\in$ or $=$. Then

  if x δ y then I   while x δ y do I   for x in y do I

  are all programs. The nonstandard construct is the for loop. The general idea is that the instruction I is executed, in parallel, with one thread for every element x of the set y. The semantics are described in more detail below, in particular we explain how the results of the parallel threads are combined in a for loop.

Semantics. We now sketch an operational semantics for the language. Define a *program state* to be a function from program variables to definable sets (which might be an atom) which uses finitely many variables, i.e. it assigns the empty set to all but finitely many variables. If γ is a program state and I is a while program, then we write γI for the program state obtained by executing I when starting in γ. The mapping $\gamma \mapsto \gamma$I is a partial function, because the result is undefined when the program does not terminate. In the semantics, we also talk about the running time of a program, which intuitively stands for the maximal number of sequentially executed instructions.

We only explain the semantics for programs of the form

for x in X do I,

the other constructions being handled in the standard way. Suppose that we want to execute the for loop above on a program state γ. If the variable X stores an atom, then the for loop does nothing, i.e. it does not modify the program state γ. Assume otherwise. For every element x of the set stored in X, define γ_x to be the program state obtained from γ by setting variable x to x. Depending on the choice of x, the program I might not terminate on γ_x, or it might terminate in a finite number of steps n_x. If I does not terminate on some γ_x, or the numbers n_x are unbounded, then the for loop does not terminate. Otherwise it terminates, and its running time is one plus the maximal number n_x. Apart from terminating, what does the for loop do? The idea is to run the body of the loop on every program state γ_x and then aggregate the resulting program states into a single one. We use the following aggregation function. For a set Γ of program states, define its *aggregation* to be the program state which stores on variable x:

- a definable set x (possibly an atom) if all program states in Γ have x in x;
- the union $\bigcup_{\gamma\in\Gamma} \gamma(\mathtt{x})$ otherwise.

Note that in the second case, where union is used, some $\gamma \in \Gamma$ might store an atom in x. If that happens, the atom will be lost in the aggregation, because an atom has no elements and it therefore gets ignored when taking a union. Using this notion of aggregation, we define the result of evaluating the for loop to be the aggregation of the set

$$\{\gamma_x \mathtt{I} : x \text{ is an element of the set stored in } \mathtt{X}\}$$

This completes the definition of the semantics of the for loop.

Example 36. The following program implements the mapping $\{x\} \mapsto x$. More precisely, if the variable x stores a singleton $\{x\}$, then after running the program the variable result will store x, otherwise result will store the empty set.

```
result := ∅
for y in x do
  y:=y
if x = {y} then
 result := y
```

In the above, testing if x = {y} is syntactic sugar for first setting a fresh variable z to {y} and then testing if x and z have the same value. □

7.4 Example programs

In this section, we show some example programs. Like in Python, we use indentation to distinguish blocks in programs. We first extend the syntax with some features. We allow substitutions like

```
x := {y,{y,z}}
```

which are simulated by longer pieces of code like

```
x := ∅
x := x ∪ {y}
u := x ∪ {z}
x := x ∪ {u}
```

We extend the syntax with functions (with the usual semantics); the syntax of functions is illustrated on the Kuratowski pairing function

```
function pair(x,y)
  return {{x},{x,y}}
```

We write (a,b) instead of pair(a,b). Here is the function which projects a Kuratowski pair of sets into its first coordinate, and returns ∅ if its argument is not a Kuratowski pair of sets.

```
function first(p)
  ret := ∅
  for a in p do
    for x in a do
      for y in p do
        if p = {x,{x,y}} then ret:=x;
  return ret
```

The second coordinate of a pair is extracted the same way. Similarly, we could write functions for projections of pairs storing atoms, or pairs storing one atom and one set. Using the projections, we can extend the language with a pattern-matching construction

```
for (x,y) in X do I
```

which ranges over all elements of `X` that are pairs of elements of appropriate types. We use a similar convention for tuples of length greater than two.

Example 37. [Programs that use order on atoms] Consider the atoms $(\mathbb{Q}, <)$. The relation $\leq$ on atoms is a definable set of pairs, and is therefore a constant in the language. Therefore, we can write `x ≤ y` in our programs to compare atoms for order, which is technically speaking syntactic sugar for testing membership of `(x,y)` in $\leq$. For instance, the following program (which uses boolean operations in conditionals, which can be easily simulated in the language) generates the set of growing triples of atoms.

```
X := (ℚ,<)
T := ∅
for x in X do
 for y in X do
  for z in X do
    if (x ≤ y) and (y ≤ z) then T := {(x,y,z)}
```

Since the set of growing triples of atoms is a definable set, we could also just directly load it to `T` with one instruction. □

Example 38. Assume that the atoms are Presburger arithmetic, i.e. $(\mathbb{N}, +)$. One could easily write a multiplication function, by using a while loop to implement multiplication in terms of iterated addition. One could also write a primality test. The following program looks like it computes the set of all primes:

```
P := ∅
for x in ℕ
  if prime(x) then P := P ∪ {x}
```

Actually, the program does not terminate, because the body of the for loop has unbounded running time, and the semantics says that such loops do not terminate. □

Example 39. [Reachability] We write below a program which inputs a binary relation R and a set of source elements S, and returns all elements reachable (in zero or more steps) from elements in S via the relation R. The program is written using `until`, which is implemented by `while` in the obvious way.

```
function reach (R,S)
  New := S
  repeat
   Old := New
   for (x,y) in R do
     if x ∈ Old then New := Old ∪ {y}
  until Old = New
```

The program above is the standard one for reachability, without any modifications for the setting with atoms.

The program can be run for any atoms, including non-oligomorphic ones. Sometimes, it might even terminate (and thus give the correct result). For example, if atoms are Presburger arithmetic and the relation R is the definable set of pairs of natural numbers which disagree modulo 3, and S is $\{0\}$, then the program will terminate in two iterations of the until loop and return the set of all natural numbers. However, if R would represent the successor relation (or transition function of a nonterminating Minsky machine), then the program might not terminate.

If the atoms are an oligomorphic structure, then the program will always terminate in a finite number of steps. The argument was given already in Exercise 45, but we repeat it here. Suppose that $\bar{a}$ is a tuple of atoms that supports both the relation R and the source set S. Let X be the set that contains S and every element that appears on either the first or second coordinate of a pair from R. The set X is easily seen to be supported by $\bar{a}$ and to have finitely many $\bar{a}$-orbits. Let $X_1, X_2, \ldots, X_k$ denote the $\bar{a}$-orbits of X. Therefore,

$$X = X_1 \cup X_2 \cup \cdots \cup X_k. \tag{7}$$

It is easy to see that after every iteration of the `repeat` loop, the values of both variables `New` and `Old` are subsets of X that are supported by $\bar{a}$. Therefore the values of these variables are obtained by selecting some of the orbits listed in (7). In each iteration of the `repeat` we add some orbits, and therefore the loop can be iterated at most k times. □

Example 40. [Automaton emptiness] Using reachability from the previous example, it is straightforward to write an emptiness check for nondeterministic definable automata (recall them from Exercise 58):

```
function emptyautomaton(A,Q,I,F,delta)
  R := ∅
  for (p,a,q) in delta do
    R := R ∪ {(p,q)}
  return ∅ = (reach(R,I) ∩ F)
```

The program will always terminate if the atoms are oligomorphic. □

Example 41. [Automaton minimisation] The program for automaton emptiness answered a yes/no question. We now present a program that minimizes deterministic automata. The program is a function

```
function minimize(A,Q,q0,F,delta)
```

which inputs a definable deterministic automaton and returns the minimal automaton. (We assume that the atoms are oligomorphic.) We assume that all states are reachable, the non-reachable states can be discarded using the emptiness procedure described above. The code is a standard implementation of Moore's minimisation algorithm. The only point of writing it down here is that the reader can follow the code and see that it works with atoms.

In a first step, we compute in the variable `equiv` the equivalence relation which identifies states that recognise the same languages. To do this, we compute the complement of the equivalence relation. We first the non-equivalence relation to states that are distinguished by the empty word:

```
nonequiv := (F × (Q-F)) ∪ ((Q-F) × F)
```

(The code above uses $\times$, which is implemented using `for`.) Then iterate the following code using a `while` loop, until the set `R` does not grow any more:

```
for p in Q do
  for p' in Q do
    for a in A do
      for q in Q do
        for q' in Q
          if (p,a,q) ∈ delta and (p',a,q') ∈ delta and (q,q') ∈ R
             nonequiv := nonequiv ∪ {(p,p')}
```

Once the set `nonequiv` has stabilised, it contains exactly the pairs of states which recognise different languages. Therefore, we get the "same language" relation by doing complementation:

```
equiv := (Q × Q) - reach(R,base)
```

For the states of the minimal automaton, we use the equivalence classes of the relation `equiv`, which are produced by the following code.

```
function classes (equiv)
  for (a,b) in equiv do
    for (c,d) in equiv do
       if a=c then class := class ∪ {c}
    ret := ret ∪ {class}
  return ret
```

The remaining part of the minimisation program goes as expected: the states are the equivalence classes, and the remaining components of the automaton are defined as usual. □

Example 42. Call a monoid aperiodic if for every element m, the sequence $m^1, m^2, \ldots$ is ultimately constant. Assume that the atoms are oligomorphic. The following program inputs a monoid (its carrier and the graph of the monoid operation) and returns true if and only if the monoid is aperiodic. The program simply executes the definition of aperiodicity.

```
function aperiodic (Carrier,Monop)
  counterexamples := ∅
    for m in Carrier  do
      X := ∅
      power := m
      while power ∉ X
         X :=X ∪ {power}
         power := Monop(power,m)
      if power  ≠ Monop(power,m) then
        counterexamples := counterexamples ∪ {m}
    if counterexamples = ∅ then return true else return false
```

In the program above, `Monop(power,m)` is actually syntactic sugar for a subroutine which examines the graph of the multiplication operation `Monop`, and finds the unique element `x` which satisfies `(power,m,x) ∈ Monop`.

In the program, the set `X` is used to collect consecutive powers $\mathtt{m}, \mathtt{m}^2, \mathtt{m}^3, \ldots$. To prove termination, one needs to show that this set is always finite, even if the monoid in question is not aperiodic. Furthermore, there is a fixed upper bound on the size of such sets. Every power of `m` is supported by whatever supports `m` and the multiplication operation in the monoid. Since the carrier of the monoid is definable, it is also orbit-finite, by Theorem 4.2. In an orbit-finite set, there are finitely many elements with a given support, as shown in Exercise 47. It follows that for every `m`, the set of its powers is finite. Furthermore, there is a common upper bound on the size of these sets, because if two elements are in the same orbit, then the number of their powers is the same. □

7.5 An interpreter

In this section, we show how definable while programs can be simulated by normal programs (e.g. Turing machines). The idea is to describe the parallel execution in a symbolic way.

Definition 7.5 (Function computed by a definable while program) *A partial function*

$$f : \mathsf{def}\mathbb{A} \to \mathsf{def}\mathbb{A}$$

is said to be computed by a definable while program *if there is a while program* `I` *with a designated interface variable, such that for every definable set x, if the program* `I` *is executed in the program state where the interface variable stores x and all other variables are empty, then it* `I` *terminates if and only if $f(x)$ is defined, and if it terminates then the interface variable stores $f(x)$.*

Theorem 7.6 *Assume that the atoms are an effective structure. If a partial function*

$$f : \mathsf{def}\mathbb{A} \to \mathsf{def}\mathbb{A}$$

is computed by a definable while program (in the sense of Definition 7.5) then it is computable (in the sense of Defintion 7.2).

To prove the above theorem, we show how the semantics of definable while programs can be simulated by manipulating definable sets. A program state γ can be viewed as a set

$$\{(\mathtt{x}, \gamma(\mathtt{x})) : \mathtt{x} \text{ is a program variable used by } \gamma\ \}.$$

Assuming that the program variables are special cases of definable sets, e.g. they are natural numbers, a program state is a special case of a definable set. In particular, it makes sense to speak of definable sets of program states. The Interpreter Theorem follows as a special case of the following lemma, by using definable sets of program states that are singletons.

Lemma 7.7 (Interpreter Lemma) *Assume that the atoms are an effective structure. There is a computable function (call it an interpreter) which inputs a definable while program* `I` *and a definable set Γ of program states, and does the following:*

- *if for every $n \in \{0, 1, \ldots\}$ there is some program state $\gamma \in \Gamma$ such that* `I` *does not terminate on γ in at most n steps, then the interpreter does not terminate.*
- *Otherwise, the interpreter terminates and outputs $\{(\gamma, \gamma\mathtt{I}) : \gamma \in \Gamma\}$.*

Proof
Note that it follows implicitly from the statement of the lemma that the output is also a definable set, since computable functions on definable sets can only output definable sets.

In the proof of the lemma, it will be convenient to use the following data structure. For definable sets $X_1, \ldots, X_n$, define their *union-member structure* to be the relational structure whose universe is the union-member closure (as defined in Exercise 81) of $X_1 \cup \cdots \cup X_n$ and which is equipped with unary predicates for the sets $X_1, \ldots, X_n$ and a binary predicate $\in$ for set membership. Using Exercises 81 and 80, it follows that if $X_1, \ldots, X_n$ are definable sets, then their union-member structure is also definable set and can be computed.

The interpreter works by structural induction on the text of the program `I`. The proof essentially says that the operational semantics can be made effective. We only do the proof for `if`, `while` and `for`.

- **If.** Suppose that we want to compute the graph of the function

$$\gamma \in \Gamma \qquad \mapsto \qquad \gamma(\texttt{if x}\ \delta\ \texttt{y then I}) \tag{8}$$

where δ is one of $=, \in$. Consider the two sets

$$\underbrace{\{(\gamma, \gamma(\texttt{x})) : \gamma \in \Gamma\}}_{X} \qquad \underbrace{\{(\gamma, \gamma(\texttt{y})) : \gamma \in \Gamma\}}_{Y}$$

which are definable by the assumption that Γ is definable. Let $\mathfrak{A}$ be the union-member structure of these two sets. From Exercise 79 it follows that any relation that can be defined in first-order logic inside $\mathfrak{A}$ is also definable, which means that we can quantify over elements of $X \cup Y$, elements of elements of $X \cup Y$, etc. and also compare them for equality or membership. Note that the Kuratowski definition of pairs can be formalised in first-order logic inside the union-member structure. As an application of this principle, let $\Gamma_0 \subseteq \Gamma$ be the program states which satisfy the condition in the `if` statement:

$$\Gamma_0 = \{\gamma : \exists x \exists y \ \underbrace{(\gamma, x) \in X}_{\gamma(\texttt{x})=x} \wedge \underbrace{(\gamma, y) \in Y}_{\gamma(\texttt{y})=y} \wedge x \ \delta \ y\}.$$

By Exercise 79, the above set is definable and can be computed based on X and Y. We will use this type of reasoning several times in the proof: whenever something can be defined using first-order logic based on previously known definable things, then it is also definable and can be computed. Apply the induction assumption to compute the graph of

$$\{(\gamma, \gamma\texttt{I}) : \gamma \in \Gamma_0\}$$

Using first-order logic in the union-member structure of Γ, Γ_0 and the above relation, we define the graph of the function from (8), using the formula

$$\{(\gamma, \mu) : (\gamma \in \Gamma_0 \wedge \gamma\texttt{I} = \mu) \vee (\gamma \in \Gamma - \Gamma_0 \wedge \gamma = \mu)\},$$

and therefore the graph itself is definable.

- **While loop**. Consider a definable program of the form

```
while x = y do J
```

Let Γ be a definable set of program states on which we want to execute the above program. For $n \in \{0, 1, \ldots\}$ let $\Gamma_n \subseteq \Gamma$ be those program states which take at most n iterations to finish the while loop. Using the same approach as in the case of `if`, for each n we can compute Γ_n and also the semantics of the while loop with domain restricted to Γ_n. We try all n until $\Gamma_n = \Gamma$. If no such n exists, then the interpreter does not terminate.

- **For loop**. Assume that the input program `I` is

```
for x in X do I.
```

It is not difficult to show that the following ternary relation is definable

$$\{(\gamma, x, \gamma_x) : \gamma \in \Gamma, x \in \gamma(\mathtt{X})\}$$

Using the induction assumption, the following binary relation is definable

$$\{(\gamma, \gamma_x \mathtt{I}) : \gamma \in \Gamma, x \in \gamma(\mathtt{X})\}$$

By Exercise 82, the graph of the function

$$\gamma \in \Gamma \mapsto \{\gamma_x \mathtt{I} : x \in \gamma \mathtt{X}\}$$

is a definable set. The semantics of the entire for loop is the composition of the above function with the aggregation function used in the semantics of a `for` loop. Since functions with definable graphs are closed under composition, it suffices to show the following lemma.

Lemma 7.8 *If Δ is a definable family of definable sets of memory states, then the following partial function has a definable graph, which can be computed from Δ:*

$$\Gamma \in \Delta \quad \mapsto \quad \textit{the aggregation of } \Gamma.$$

Proof
We show that for every program variable `x` the function

$$\Gamma \in \Delta \quad \mapsto \quad \text{value of } \mathtt{x} \text{ in the aggregation of } \Gamma. \tag{9}$$

has a definable graph. Consider the union-member structure corresponding to Δ. Define Δ_0 to be those $\Gamma \in \Delta$ where all program states agree on variable `x`, this set is definable as it can be defined in first-order logic in the union-member structure of Δ. For the same reason, the function from (9) can be defined using first-order logic in the same structure. □

□

7.6 Computational completeness of definable while programs

In Theorem 7.7, we showed that definable while programs describe computable functions on definable sets. Do they describe all computable functions? Did we miss some feature in the programming language? In this section we show that definable while programs are computationally complete in a sense that is described below.

Theorem 7.9 (Computational completeness of definable while programs) *Assume that the atoms $\mathbb{A}$ are an effective structure which is furthermore effectively oligomorphic in the sense of Exercise 77. Then for every partial function*

$$f : \mathsf{def}\mathbb{A} \to \mathsf{def}\mathbb{A}$$

the following conditions are equivalent:

1. *f is computed by a definable while program;*
2. *f is finitely supported and computable.*

The implication from 1 to 2 is a corollary of Theorem 7.7 and the observation that the semantics of a definable while program are invariant under atom automorphisms which fix those atoms that appear in the code of the program. The implication from 1 to 2 does not use effective oligomorphism or even oligomorphism alone.

The rest of Section 7.6 is devoted to proving the implication from 2 to 1. Let f be as in item 2. To show item 1, we use the following lemma, which says that a while program can convert a definable set into a set builder expression defining it, at least up to automorphisms.

Lemma 7.10 *Assume that the atoms are effectively oligomorphic. Let $\bar{a}$ be a tuple of atoms. There is a definable while program which inputs a definable set x and outputs a set builder expression α and a tuple of atoms $\bar{b}$ evaluating its free variables such that*

$$\pi(x) = [\![\alpha]\!](\bar{b}) \qquad \text{for some } \bar{a}\text{-automorphism } \pi.$$

Proof

Suppose that the input set is x. The program enumerates through all possible set builder expressions α. For each set builder expression α, say with free variables $\bar{y}$, it does a for loop across all $\bar{y}$-tuples of atoms to compute the set

$$B_\alpha = \{\bar{b} : \bar{b} \text{ is a } \bar{y}\text{-tuple of atoms such that } x = [\![\alpha]\!](\bar{b})\}.$$

To compute B_α, we need to show that a while program can compute $[\![\alpha]\!](\bar{b})$ given α and $\bar{b}$; this is not difficult to do by structural induction on the set builder expression α. If the set B_α is empty, then the program proceeds to the next set builder expression α. By definition of definable sets, eventually a set builder expression α will be found so that B_α is nonempty. Suppose then that α is such that B_α is nonempty, and let n be the number of free variables in α, which means that $B_\alpha \subseteq \mathbb{A}^n$.

Let k be the dimension of the tuple $\bar{a}$. Using Exercise 78 and the assumption that the atoms are effectively oligomorphic, one can compute first-order formulas $\varphi_1, \ldots, \varphi_m$ in $k+n$ free variables which define all orbits of $\mathbb{A}^{k+n}$. Every $\bar{b} \in B_\alpha$ is in some $\bar{a}$-orbit, which means that there must be some i such that $\varphi_i(\bar{a}\bar{b})$ holds. Therefore, in particular there must be some i such that some tuple $\bar{b} \in B_\alpha$ satisfies $\varphi_i(\bar{a}\bar{b})$, and this i can be computed. (First-order formulas can be evaluated in the program, by using for loops to simulate quantifiers; this observation was already used in evaluating set-builder expressions.) A tuple of atoms $\bar{b}$ satisfies $\varphi_i(\bar{a}\bar{b})$ if and only if

$$\pi(x) = [\![\alpha]\!](\bar{b}) \qquad \text{for some } \bar{a}\text{-automorphism } \pi.$$

The program uses decidability of the first-order theory of $\mathbb{A}$ to enumerate all possible tuples $\bar{b}$ until it finds one which maps which makes $\varphi_i(\bar{a}\bar{b})$ true, and this is the output. □

We now complete the proof of the implication from 2 to 1 in Theorem 7.9. Suppose that f is a function from definable sets to definable sets which is supported by a tuple of atoms $\bar{a}$, and assume that f is computable. We present below a definable while program which computes f. Assume that on input we have a definable set x. By Lemma 7.10, a definable while program can compute α and $\bar{b}$ such that

$$\pi(x) = [\![\alpha]\!](\bar{b}) \qquad \text{for some } \bar{a}\text{-automorphism } \pi. \tag{10}$$

By the assumption that f is computable, a while program can compute a set builder expression β and an atom tuple $\bar{c}$ evaluating its free variables such that

$$f([\![\alpha]\!](\bar{a})) = [\![\beta]\!](\bar{c}) \tag{11}$$

Using the same ideas as in Lemma 7.10, a definable while program can compute the set $\bar{a}$-orbit of the tuple $\bar{b}\bar{c}$, i.e. the set

$$\{\pi(\bar{b}\bar{c}) : \pi \text{ is an } \bar{a}\text{-automorphism}\}$$

From the above, we can compute the set

$$f_0 \stackrel{\text{def}}{=} \{([\![\alpha]\!](\pi(\bar{b})), [\![\beta]\!](\pi(\bar{c}))) : \pi \text{ is an } \bar{a}\text{-automorphism}\} \stackrel{(11)}{=} \\ \{\pi([\![\alpha]\!](\bar{b}), f([\![\alpha]\!](\bar{b}))) : \pi \text{ is an } \bar{a}\text{-automorphism}\}.$$

Since f is a function supported by $\bar{a}$, the set f_0 is a subset of the graph of f. Since the partial function f_0 is supported by $\bar{a}$, and its domain contains $[\![\alpha]\!](\bar{b})$, it follows from (10) that the domain of f_0 also contains x. Therefore, we can simply evaluate f_0 in x to get $f(x)$. This completes the implication from 2 to 1 in Theorem 7.9.

Bibliographic notes for Section 7. The programming language of while programs with atoms is based on the language from [18]. This language was preceded by [9], which proposed a programming language in a functional paradigm (see also [43] for an implementation of the functional programming language together with an application to learning register automata). Both languages (definable while programs and the functional language) have the same expressive power. Definition 7.2 and Theorem 7.9 on computational completeness of while programs are inspired by [13, Theorems IV.1 and IV.2], although Theorem 7.9 is more general because it applies to computation on sets and not just objects that can be encoded as strings (see Section 9 for limitations of the second approach).

The programming languages from [9, 18] and their semantics were written under the assumption that the atoms are oligomorphic. The idea to extend the semantics to atoms that are not necessarily oligomorphic, but have decidable first-order theory, is inspired by the programming language LOIS (*Looping over Infinite Sets*) [37, 38]. The Symbol Pushing Lemma is inspired by LOIS.

8 Automata

In this section, we discuss automata in sets with atoms. Unless explicitly noted, we always consider the case when the atoms are effectively oligomorphic, with the ensuing good properties of orbit-finite sets, e.g. closure under unions, subsets and products. For example, these properties are enough to give equivalence of pushdown automata and context-free grammars. However, construction might fail if it uses the powerset operation, e.g. determinisation of finite automata.

8.1 Orbit-finite automata

In this section, we discuss the atom versions of nondeterministic and deterministic finite automata. The general idea is that these are the same as the register automata from Section 1. We prove that the two models are not equivalent, and that the deterministic version admits a Myhill-Nerode characterization.

Orbit-finite automata. The definition of an nondeterministic orbit-finite automaton is the same as the definition of a nondeterministic finite automaton, except the word "finite set" is replaced by "orbit-finite set with atoms". In other words, a *nondeterministic orbit-finite automaton* is a tuple

$$\mathcal{A} = (\underbrace{Q}_{\text{states}} \quad \underbrace{\Sigma}_{\text{input alphabet}} \quad \underbrace{I \subseteq Q}_{\text{initial states}} \quad \underbrace{F \subseteq Q}_{\text{states}} \quad \underbrace{\delta \subseteq Q \times \Sigma \times Q}_{\text{transitions}})$$

where all components are orbit-finite sets with atoms. An automaton is called *deterministic* if it has one initial state, and δ is a function from $Q \times \Sigma$ to Q. Acceptance is defined in the usual way. The language recognised by an automaton is the set of words it accepts.

Suppose that all components in an automaton are finitely supported, and Q and Σ are orbit-finite. By the assumption that the atoms are oligomorphic, the remaining components, i.e. the initial and accepting states, as well as the transitions, are all orbit-finite sets, as finitely supported subsets of products of orbit-finite sets.

An alternative definition would be a *definable nondeterministic automaton*, as mentioned in Exercise 58, where all components are definable sets. We begin by observing that definable automata are the same thing as orbit-finite automata up to isomorphism, at least as long as oligomorphic atoms are used. Define an *isomorphism* between automata

$$\mathcal{A}_i = (Q_i, \Sigma_i, I_i, F_i, \delta_i) \qquad \text{for } i \in \{1, 2\}$$

to be a pair of bijections $f : Q_1 \to Q_2$ and $g : \Sigma_1 \to \Sigma_2$ which are consistent with the transition relations and accepting/final states in the natural way. From Exercise 49 it follows that every orbit-finite automaton is isomorphic to a definable automaton, even via a finitely supported isomorphism. Since isomorphism does not affect the notions for automata that we study, like determinism, minimality, emptiness or universality, we will freely confuse orbit-finite and definable automata.

Atoms which support an automaton will also support the language recognised by the automaton. In particular, if an automaton is equivariant, then also its language is equivariant. This is because the definition of the language recognised by an automaton is defined in set theory, using only the membership predicate. The reasoning in the previous sentence is a special case of the following lemma:

Lemma 8.1 *Suppose that $\varphi(x, y)$ is a formula of first-order logic which only uses the set membership $\in$ predicate. Suppose that the interpretation of φ in the structure "sets with atoms equipped with membership" defines a function from x to y. If x and y satisfy $\varphi(x, y)$, then any support for x is also a support for y.*

Proof
Because the function defined by φ is equivariant, as a function from sets with atoms to sets with atoms. □

As shown in Exercise 58, definable (and therefore also orbit-finite) nondeterministic automata generalise register automata. Also, as shown in Exercise 61, when the languages are equivariant and the input alphabets are of the special form used in data words, then definable (and therefore also orbit-finite) nondeterministic automata have the same expressive power as register automata. However, when working with orbit-finite automata, we can use more fancy input alphabets, as in the following example.

Example 43. Consider the equality atoms. Let the input alphabet be

$$\Sigma = \{\{a, b\} : a \neq b \in \mathbb{A}\},$$

i.e. each letter is a set of two distinct atoms. Consider the language "the word is empty, or some atom appears in all letters", i.e.

$$L = \epsilon \cup \{a_1 \cdots a_n \in \Sigma^* : a_1 \cap \cdots \cap a_n \neq \emptyset\}.$$

A nondeterministic orbit-finite automaton which recognizes this language has states

$$Q = \mathbb{A}.$$

All states are both initial and accepting. (This does not mean that the automaton accepts all words, because sometimes no transition will be enabled.) The idea is that the automaton guesses which atom will appear in all letters, and then scans the word to see if its guess was correct. Therefore, the transition relation is

$$\delta = \{(a, \{a, b\}, a) : a \neq b \in \mathbb{A}\}.$$

□

As we will see when discussing Turing machines, using fancy input alphabets can have tremendous impact.

Example 44. The automaton from the previous example, unlike some automata, can be determinised. The deterministic automaton stores in its state the intersection of all letters it has read so far; with a special initial state indicating that it has read no letters. The initial state can be modelled as the set of all atoms, and the other states as sets of atoms of size at most two. The transition function is defined by

$$\delta(X, \{a, b\}) = X \cap \{a, b\}.$$

The accepting states are all states except $\emptyset$. □

Example 45. Consider the graph atoms, which are the Fraïssé limit of directed graphs. In these atoms, a finite set of atoms induces a finite directed graph; all finite

directed graphs can be obtained this way. Therefore, an automaton can read a sequence of atoms $a_1 \cdots a_n$ and recognize properties such as: "the sequence is a path". This leads to the following question: for which graph properties X, is the following language recognised by a nondeterministic automaton:

$$L_X = \{a_1 \cdots a_n : \text{ the graph induced by } a_1, \ldots, a_n \text{ satisfies } X\}$$

To recognise L_X, the automaton should be prepared for an arbitrary enumeration of the vertices of the graph, possibly with repetitions. Properties X for which L_X is recognizable by an automaton include "contains a clique of size three", "is not a clique", "contains a vertex connected to all other vertices" but do not include the complementary properties "does not contain a clique of size three" or "is a clique". A sufficient condition is definability by a formula of first-order logic with a quantifier prefix $\exists^*\forall$. Is this condition necessary? □

Exercise 83. Consider the following weakening of Minsky machines. The automaton has a finite set of states, as well as a finite set of counters, which store natural numbers. The automaton can test a counter for zero. Instead of the increment and decrement operations in Minsky machines, the automaton can execute operations of the form "make counter c strictly bigger" and "make counter c strictly smaller". The model is nondeterministic, since the automaton has no control over the increase or decrease of a counter. The automaton accepts by reaching an accepting state. Show that emptiness is decidable.

Exercise 84. Assume that the atoms are oligomorphic. Show that the expressive power of nondeterministic orbit-finite automata is not changed if ϵ-transitions are allowed.

Exercise 85. Assume that the atoms are oligomorphic. Show that the class of languages recognised by nondeterministic orbit-finite automata is closed under orbit-finite union, in the sense of Exercise 51.

Exercise 86. Consider the equality atoms. Show that languages recognised by nondeterministic orbit-finite automata are not closed under orbit-finite intersection, in the sense defined in Exercise 51.

What about closure under complementation? The standard proof relies on determinization, which relies on the subset construction. The subset construction fails in sets with atoms, because the powerset of an orbit-finite need not be orbit-finite. Complementation is actually impossible.

Lemma 8.2 *Languages recognised by nondeterministic orbit-finite automata are not closed under complementation.*

Proof
Languages recognised by nondeterministic register automata are not closed under

complements, and when the input alphabet is as in data words, then nondeterministic orbit-finite automata have the same expressive power as nondeterministic register automata (Exercise 61). A direct proof is also easy, by showing that the language

$$\{a_1 \cdots a_n \in \mathbb{A}^* : a_1, \dots, a_n \text{ are pairwise distinct}\}$$

is not recognised by any nondeterministic orbit-finite automaton, under the equality atoms. Indeed suppose that there would be such an automaton. Choose n so that for every state there is a tuple of less than n atoms which supports the state and the entire automaton. Consider an accepting run over an input word with $2n$ distinct atoms:

$$q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_{2n}} q_{2n}.$$

By choice of n, we can find a tuple $\bar{a}$ of less than n atoms which supports both q_n and the entire automaton. In particular, one of the atoms some $a_i \in \{a_1, \dots, a_n\}$ does not appear in $\bar{a}$, and also some $a_j \in \{a_{n+1}, \dots, a_{2n}\}$ does not appear in $\bar{a}$. Therefore we can find a $\bar{a}$-automorphism which fixes $\bar{a}$ but maps a_i to a_j. We have thus found a run which begins in the initial state, ends in q_n, and reads an input word which contains a_j. Combining this with the run from q_n to q_{2n} which also uses a_j, we get an accepting run over a word which contains some atom twice. □

Corollary 8.3 *Deterministic and nondeterministic orbit-finite automata have different expressive powers.*

Minimization of deterministic automata As observed above, determinization fails. Something that does work, and which justifies the usefulness of deterministic orbit-finite automata, is the Myhill-Nerode theorem. The following example is very close to Exercise 2.

Example 46. [Automata with registers do not minimise] Consider the equality atoms. Let the input alphabet be the atoms, and consider the language of words where at most two atoms appear, possibly with repetitions. To recognize this language, we can use a deterministic automaton with two registers. More formally, the state space is

$$\underbrace{(\mathbb{A} \cup \{\bot\})}_{\text{register 1}} \times \underbrace{(\mathbb{A} \cup \{\bot\})}_{\text{register 2}} \quad \cup \quad \{\text{reject}\}$$

The automaton begins in the state $(\bot, \bot)$. When it sees an atom which is not in the registers, it loads it into the first undefined register, if both registers are full it rejects. (The automaton does not even use states of the form $(\bot, a)$.)

We have just described a deterministic automaton, which recognizes the language, and which uses registers. The problem with this automaton is that it does not store the minimal amount of information. Because the registers are ordered, the states (a, b) and (b, a) are different. With respect to our language, the two states

should be equivalent. In other words, the automaton should have states which are unordered sets of atoms of size at most two. This example shows that in order to store the minimal amount of information, registers are not always the right choice. □

We now show that the problems mentioned above go away when we move from register automata to orbit-finite automata. We begin by recalling the standard definition of Myhill-Nerode equivalence. Suppose that $L \subseteq A^*$ is a language. We define two words $w, w' \in A^*$ to be L-equivalent if for every word $v \in A^*$, the language L contains either both or none of the words wv and $w'v$. As usual, this equivalence relation is a congruence with respect to appending letters, and therefore it makes sense to consider an automaton where the states are the equivalence classes, and where the transition function is defined so that after reading a word w, the state is the equivalence class of w. This automaton is called the *syntactic automaton of L*. Because the syntactic automaton is defined using the language of set theory, Lemma 8.1 says that any support of the language is a support of the syntactic automaton.

Theorem 8.4 *A language is recognised by a deterministic orbit-finite automaton if and only if its syntactic automaton has an orbit-finite state space.*

Proof
The right-to-left implication is immediate. For the left-to-right implication, we observe that states of the syntactic automaton can be obtained from the states of an arbitrary deterministic automaton recognising the language, by applying a finitely supported function. Since finitely supported functions preserve orbit-finite sets, it follows that the syntactic automaton must have an orbit-finite state space, if the original automaton did. □

Exercise 87. Consider the following extension of register automata to arbitrary orbit-finite alphabets: this is the special case of nondeterministic orbit-finite automata where the set of states is of the form

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

for some finite (not just orbit-finite) sets Q and R. Show that such an automaton cannot recognise the language from Example 43.

8.2 Pushdown automata and beyond

In this section, we discuss computation models beyond finite automata, mainly pushdown automata. Turing machines will be discussed in more detail in Section 9. For Turing machines, we will be most interested in comparing the expressive power of deterministic, nondeterministic and alternating Turing machines.

In all examples from this section, we assume that the atoms are oligomorphic. An orbit-finite pushdown automaton is defined the same way as normal pushdown automaton, except the word "finite" is replaced by "orbit-finite", and all components are required to be sets with atoms. A *orbit-finite pushdown automaton $\mathcal{A}$* consists

of orbit-finite input and stack alphabets A, Γ, distinguished initial states and stack symbols, an orbit-finite state space Q, and a finitely supported set of transitions

$$\delta \subseteq Q \times \Gamma \times (A \cup \epsilon) \times Q \times \Gamma^*.$$

The semantics of the pushdown automaton are defined in the standard way (we use acceptance through the empty stack). Because the semantics of an automaton are defined in the language of set theory, Lemma 8.1 implies that the language is supported by any atoms that support the automaton.

When the atoms are effectively oligomorphic, then emptiness is decidable for pushdown automata. The idea is that the emptiness algorithm for pushdown automata is a fix-point algorithm, and such algorithms generalise to orbit-finite sets, in the spirit of Exercise 45. We will formalise this in the next section, on algorithms transforming orbit-finite sets.

Example 47. [Pushdown automaton for palindromes.] For an orbit-finite alphabet Σ, consider the language of palindromes, i.e. those words in Σ^* which are equal to their reverse. This language is recognised by a orbit-finite pushdown automaton which works exactly the same way as the usual automaton for palindromes, with the only difference that the stack alphabet Γ is now an orbit-finite set, namely Σ. For instance, in the case when $\Sigma = \mathbb{A}$, the automaton keeps a stack of atoms during its computation. The automaton has two control states: one for the first half of the input word, and one for the second half of the input word. As in the standard automaton for palindromes, this automaton uses nondeterminism to guess the middle of the word. □

Example 48. [Pushdown automaton for modified palindromes.] The automaton in Example 47 had two control states. In some cases, it might be useful to have a set Q of control states that is orbit-finite. Consider for example the set of odd-length palindromes where the middle letter is equal to the first letter. A natural automaton recognising this language would be similar to the automaton for palindromes, except that it would store the first letter a_1 in its control state.

Another solution would be an automaton which would keep the first letter in every token on the stack. This automaton would have a stack alphabet of $\Gamma = \Sigma \times \Sigma$, and after reading letters $a_1 \cdots a_n$ its stack would be

$$(a_1, a_1), (a_1, a_2), \ldots, (a_1, a_n).$$

This automaton would only need two control states. Actually, using the standard construction, one can show that every orbit-finite pushdown automaton can be converted into one that has one control state, but a larger stack alphabet. □

The following exercise gives some motivation for studying orbit-finite pushdown automata.

Example 49. [Modelling boolean recursive programs with atoms] Pushdown automata without atoms are sometimes used to model the behaviour of recursive boolean programs. This modelling can be extended with atoms, which means that we can also model programs that have variables for atoms. Consider the total order atoms. Consider a recursive function such as the following one. (This program does not do anything smart.)

```
function f(a: atom)
begin
  b:=read() // read an atom from the input
  if b = a then
    return b
  else if b > a then  // the program can use the order on atoms
    return f(b) // do a recursive call
  else
    fail() // terminate the computation
end
```

The behaviour of this program can be modelled by an orbit-finite pushdown automaton. The input tape corresponds to the `read()` functions. The stack corresponds to the call stack of the recursive functions; the stack stores atoms since the functions take atoms as parameters. Since the only variables are atoms, the set of possible call frames is orbit-finite, and therefore the stack alphabet is orbit-finite.

An orbit-finite pushdown automaton could be used to model more sophisticated examples: many mutually recursive functions, boolean variables, other homogeneous data types for the atoms. □

Exercise 88. Consider the equality atoms. We say that two sets of atoms x, y are *fresh* with respect to each other if they can be supported by disjoint tuples of atoms. Consider a model of orbit-finite pushdown automata extended by one new kind of transition:

$$q \overset{\text{fresh(a)}}{\rightarrow} p,$$

where q, p are states and a is an input letter. When executing this transition, the automaton reads letter a and changes state from q to p, but only under the condition that a is fresh with respect to every letter on the stack and the current state q. Show that emptiness is decidable.

Exercise 89. Consider the equality atoms and the following higher order variant orbit-finite pushdown automaton. The automaton has a stack of stacks. There are operations as in a usual pushdown automaton, which apply to the topmost stack. There is also an operation "duplicate the topmost stack" and an operation "delete the topmost stack". Show that emptiness is undecidable.

Exercise 90. Define an orbit-finite context-free grammar like a normal context-free grammar, except that the terminals, nonterminals and rules can all be orbit-finite sets. Show that orbit-finite pushdown automata and orbit-finite context-free grammars define the same language classes.

Exercise 91. Show that a language that is orbit-finite context-free, but is not generated by any orbit-finite context-free grammar with a finite (not just orbit-finite) set of nonterminals.

Bibliographic notes for Section 8. Orbit-finite sets were originally introduced to model language recognisers such as automata or monoids in a machine independent way, i.e. via a theorem in the style of Myhill-Nerode (Theorem 8.4). To do this, one needed a representation of the state space which would: a) be simple enough to allow finite representations; b) generalise the configuration space of a register automaton; and c) allow quotienting as required in the Myhill-Nerode theorem. Orbit-finite sets are a solution to these requirements. The Myhill-Nerode theorem for orbit-finite sets was originally proved in [8, Lemma 3.3] for monoids and then in [12, Theorem 3.8] for automata; these respective papers are the ones which introduced orbit-finite monoids and orbit-finite automata. The computational complexity of automata minimisation are studied in [47]. A variant of the Myhill-Nerode theorem for timed automata is presented in [15]; the difficulty there is to deal with atoms which are not oligomorphic.

Context-free languages for infinite alphabets were originally introduced in [20], which contains a proof of equivalence for register extensions of context-free grammars and pushdown automata. The generalisation to orbit-finite pushdown automata and context-free grammars is from [12]. See also [46, 22, 23]. Higher-order pushdown automata for infinite alphabets, as in Exercise 89, come from [46, Section 6].

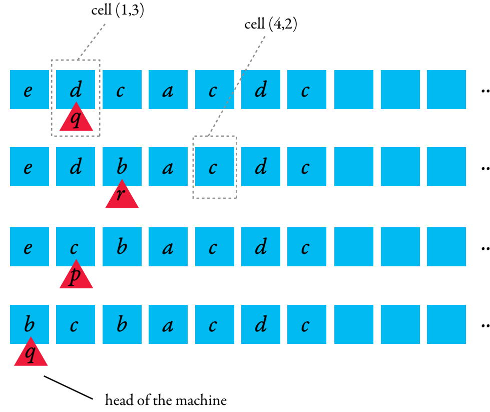
9 Turing machines

In this section, we talk about definable Turing machines. A definable Turing machine is defined the same way as a Turing machine, except that all components (states, input and work alphabets, transitions) are required to be orbit finite sets with atoms. For the sake of concreteness, we use a model which has one tape, and where the transition relation is a subset of:

$$\underbrace{(\overbrace{\Gamma}^{\text{work alphabet}} \cup \{\text{blank}\}) \times \overbrace{Q}^{\text{states}}}_{\text{what the machine sees}} \times \underbrace{\big(\{\text{accept, reject}\} \cup (\overbrace{\Gamma \times Q \times \{\text{left, stay, right}\}}^{\text{write a symbol and move the head}})\big)}_{\text{what the machine does}}.$$

We assume for the rest of this section that the atoms are oligomorphic and have decidable first-order model checking. By the assumption on oligomorphism, we definable is the same thing as hereditarily orbit-finite.

The tape in a Turing machine is a sequence of cells indexed by natural numbers (we use tapes that are infinite to the right only), each cell storing a symbol of the work alphabet or a blank symbol (and possibly also the state of the machine, if head happens to be in that cel). Here is a picture of a computation:



A finite computation of a nondeterministic Turing machine is defined to be a function (for some n, namely the computation time)

$$\rho : \underbrace{\mathbb{N}}_{\text{space}} \times \underbrace{\{1, \ldots, n\}}_{\text{time}} \to \underbrace{\Gamma \cup \{\text{blank}\}}_{\text{cells without the head}} \cup \underbrace{(\Gamma \cup \{\text{blank}\}) \times Q}_{\text{cells with the head}} \qquad (12)$$

which is consistent with the transition function of the Turing machine in the usual way. We assume that the first row (i.e. corresponding to the initial step of the computation) consists of some input word padded by an infinite sequence of blanks, with the head having the initial state and positioned over the first input letter.

Many of the standard results for standard Turing machines remain true for definable Turing machines, such as equivalence of single-tape and multi-tape machines, using the standard proofs. What does not work, however, is determinisation, and this is the focus of this section.

Example 50. [A Turing machine checking that all letters are different] Consider the equality atoms. Assume that the input alphabet is $\mathbb{A}$. We show a deterministic Turing machine which accepts words where all letters are distinct, and the atom $\underline{5}$ does not appear. The idea is that the machine iterates the following procedure until the tape is contains only blank symbols: if the first non-blank letter on the tape is $a \neq \underline{5}$, replace it by a blank and load a into the state, scan the word to check that a does not appear again, and go back to the beginning of the tape. If the first non-bnak letter on the tape is $\underline{5}$, then reject immediately. The set of states is

$$\underbrace{\{q_0\}}_{\text{initial state}} \cup \underbrace{\mathbb{A} - \{\underline{5}\}}_{\text{scan to the right if this atom reappears}} \cup \underbrace{\{q_1\}}_{\text{return to the beginning}}$$

□

Example 51. [Deatomisation] Consider the equality atoms. This example shows that when the input alphabet is the atoms, then a Turing machine can begin by removing the atoms from the input, and then carry on its computation without using atoms.

Define the *deatomisation* of a sequence of atoms $a_1 \cdots a_n$ to be the word

$$i_1 \# i_2 \# \cdots \# i_n \in \{0, 1, \#\}^*$$

such that i_k is the binary encoding of the number which represents the first position where atom a_k appears. Here is a picture:

a sequence of atoms	2		1		1		9		1
position numbers in binary	1		10		11		100		101
deatomisation	1	#	10	#	10	#	100	#	10

The point of deatomisation is that it stores all information about the input word up to atom automorphisms. It is not difficult to write a deterministic Turing machine, which transforms a sequence of atoms into its deatomisation. The machine only needs to test letters for equality. Therefore, any property that is decidable in terms of the deatomisation, such as "a prime number of distinct atoms, each one appearing a non-prime number of times", can be computed by a Turing machine with atoms.

The idea of deatomisation is quite specific to sequences of atoms. For sequences over other definable alphabets, the notion is less clear. As we will later see in Theorem 9.5, a deterministic deatomisation procedure is not going to be possible for some definable alhabets. □

Example 52. [A more fancy input alphabet] Consider the equality atoms. Let the input alphabet be sets of atoms of size at most ten. We show a deterministic Turing machine that recognises the language: "there exists an atom that appears in an odd number of letters".

The difficulty is with "indicating" the atom that appears in an odd number of letters. As an example, suppose that the input word is:

$$\{\underline{1}, \underline{2}, \underline{3}\}\{\underline{1}, \underline{2}, \underline{4}\}\{\underline{1}, \underline{2}\}\{\underline{1}, \underline{2}, \underline{3}\}\{\underline{1}, \underline{2}\}\{\underline{4}\}.$$

Both atoms $\underline{1}$ and $\underline{2}$ appear in an odd number of letters. However, a deterministic Turing machine cannot "indicate" the atom $\underline{1}$, since it has the same properties as the atom $\underline{2}$.

Here is a solution to the problem. When an input word is fixed, consider two atoms equivalent if they appear in exactly the same letters of the word. In the example above, the equivalence classes are

$$\{\underline{1}, \underline{2}\} \; \{\underline{3}\} \; \{\underline{4}\}.$$

(We ignore the infinite equivalence class that contains all atoms, which do not appear in the input word.) A Turing machine for the language works as follows. First, it computes the equivalence classes (each equivalence class can be stored in a cell of the tape, since it is a set of at most ten atoms). Then it nondeterministically chooses one of these equivalence classes, loads it into its state (to have a deterministic machine, each one can be tried), and sees if it appears in an odd number of positions in the input word. □

9.1 Computational completeness of alternating Turing machines

Recall Theorem 7.9, which described two equivalent characterisations of computable functions from definable sets to definable sets. Let us specialise this concept to the case of languages over a definable alphabet. Assume that the input alphabet Σ is a definable set. It is not difficult to show that every word $w \in \Sigma^*$ is a definable set, since a sequence of definable letters is also definable. View a language as its characteristic function

$$L : \Sigma^* \to \{\text{yes}, \text{no}\}$$

and extend this characteristic function to all definable sets by giving a "no" answer to every definable set which is not in Σ^*. We say that $L \subseteq \Sigma^*$ is *computable* if the function obtained this way is computable in either of the two equivalent senses from Theorem 7.9. Before continuing, let us give an alternative definition of computability which is equivalent, but more adapted to the setting of words and Turing machines.

Recall that by definition, a definable alphabet Σ is given by a set builder expression $\alpha(\bar{x})$ together with a tuple of atoms $\bar{a}$ that instantiates its free variables $\bar{x}$. The expression α cannot be an atom, since otherwise the alphabet has no letters. Therefore, it must be a finite union of set expressions:

$$\alpha(\bar{x}) = \bigcup_{i \in I} \{\alpha_i(\bar{x}\bar{y}_i) : \text{for } \bar{y}_i \text{ such that } \varphi_i(\bar{x}\bar{y}_i)\}.$$

In order to provide a letter in Σ, one needs to indicate $i \in I$ and a tuple of atoms $\bar{b}$ that evaluates the tuple of variables $\bar{y}_i$; the letter is then equal to $\alpha_i(\bar{a}\bar{b})$. Note that the length of the tuple $\bar{b}$ might depend on i. Thanks to the assumption that the atoms are effective, a tuple of atoms can be written down as a bit string. Define a *representation* of a word $w \in \Sigma^*$ to be a bit string describing the sequence of representations of letters as defined above.

Example 53. Consider the equality atoms, represented as bit strings, and the alphabet

$$\underbrace{\{\{y_1, y_2\} : \text{for } y_1, y_2 \text{ such that } y_1 \neq y_2\}}_{\text{expression 1}} \cup \underbrace{\{\underline{2}, \underline{3}\}}_{\text{expression 2}}$$

One possible source of ambiguity in representations is that an element might be captured by two different expressins, e.g. $\{\underline{1}, \underline{2}\}$ in the above example. This problem could be avoided by rewriting the expressions. Another problem, is that even if the expression is known, then two different valuations of the free variables might give the same result. For example, a letter $\{\underline{2}, \underline{5}\}$ is encoded by writing down first the expression number – necessarily expression 1, in this particular example – then giving a valuation of the variables which yields this letter. Note that there are two possible valuations, namely

$$(y_1, y_2) \mapsto (\underline{2}, \underline{5}) \quad (y_1, y_2) \mapsto (\underline{5}, \underline{2})$$

This source of ambiguity could be eliminated by choosing the lexicographically least valuation. Since ambiguity of representation will not play a role in our discussion, we will not make an effort to eliminate it. □

Lemma 9.1 *Assume that the atoms are effectively oligormorphic. If Σ is a definable alphabet, then the following conditions are equivalent for every language $L \subseteq \Sigma^*$:*

1. *L is finitely supported and $\{w \in 2^* : w$ represents a word from $L\}$ is semi-decidable in the usual sense without atoms; and*
2. *L is computed by a definable while program.*

Proof
Using Theorem 7.9. □

The main goal of this section is to give a third equivalent item in the above lemma, namely recognised by a Turing machine. However, in our proof (at least for the general case of arbitrary effective atoms structures), we will need to use *alternating* Turing machines. Already for the simplest equality atoms a deterministic machine will not be enough.

Alternating machines. We begin by describing the alternating model, which we call *definable alternating Turing machines*. The syntax is the same as for normal Turing machines, except that the control states are partitioned into four groups: *existential*, *universal*, *accepting* and *rejecting*. Define a run of an alternating Turing machine to be a well-founded tree whose nodes are labelled by configurations, such that nodes that use existential control states have one child with a successor configuration, nodes that use universal control states have all possible successor configurations as children, and nodes with accepting or rejecting control states are leaves. (By Exercise 57, a run is well-founded if and only if there is some finite bound $n \in \mathbb{N}$ on the length of all paths; this bound is called the *depth* of the run.) An accepting run over an input word is a run where the root has the initial configuration (i.e. the input word with the head over the first position in the initial state) and where all leaves are accepting. The language recognised by a definable alternating Turing machine is those words which admit at least one accepting run.

The main result in this section is the following theorem, which shows that definable alternating Turing machines are computationally complete for languages over definable alphabets. Furthermore, nondeterministic machines are enough when the atoms have quantifier elimination, i.e. every formula of first-order logic is equivalent to a quantifier-free one.

Theorem 9.2 *Assume that the atoms $\mathbb{A}$ are effectively oligomorphic. Then the following conditions are equivalent for every $L \subseteq \Sigma^*$ where Σ is definable:*

1. *L is finitely supported and $\{w \in 2^* : w$ represents a word from $L\}$ is semi-decidable in the usual sense without atoms; and*
2. *L is computed by a definable while program.*

3. L is recognised by a definable alternating Turing machine.

If $\mathbb{A}$ has quantifier elimination, then the above conditions are also equivalent to

4. L is recognised by a definable nondeterministic Turing machine.

The theorem implies that for atoms such as $(\mathbb{N}, =)$ or $(\mathbb{Q}, <)$, nondeterministic Turing machines are computationally complete for languages over definable alphabets. Actually, the assumption on quantifier elimination can be relaxed to obtain item 4, see Exercise 92. A scenario consistent with the author's knowledge, although unlikely, is that item 4 is equivalent to the other items for all effectively oligomorphic structures.

The rest of Section 9.1 is devoted to proving Theorem 9.2. The equivalence of items 1 and 2 is Lemma 9.1. The implication from 3 to 2 is straightforward, by using a definable while program to enumerate candidates for accepting runs. It remains to prove the implication from 1 to 3 (and the implication from 1 to 4 under the additional assumption on quantifier elimination.)

We begin by describing the reason why alternating machines are used: they can evaluate formulas of first-order logic. A formula φ of first-order logic can be encoded as a bit string in a natural way, let us write $\underline{\varphi}$ for this encoding. Alternating Turing machines are a perfect model for evaluating first-order formulas, as shown in the following lemma.

Lemma 9.3 *If the atoms have a finite vocabulary (which is part of the definition of being effective), then the following language over alphabet $\mathbb{A} + \{0, 1\}$ is recognised by an alternating Turing machine:*

$$\{a_1 \cdots a_n \underline{\varphi} : a_1, \ldots, a_n \in \mathbb{A}, \varphi \text{ has } n \text{ free variables, and } \mathbb{A} \models \varphi(a_1, \ldots, a_n)\ \}.$$

For quantifier-free formulas, a deterministic machine is enough.

Proof

The machine simply implements the semantics of first-order logic, using universal states for the universal quantifiers and existential states for the existential quantifiers. The assumption that the vocabulary of the atoms is finite is used in the induction base, for atomic formulas, in which case the relations of the atoms are simply hard-coded into the Turing machine. □

The above lemma gives an alternative solution to Exercise 51, since having only distinct letters can be expressed by a quantifier-free formula. The following lemma shows computational completeness in the special case when the input alphabet is $\mathbb{A}$.

Lemma 9.4 *Assume that the atoms are effective. Let $L \subseteq \mathbb{A}^*$ satisfies item 1 in Theorem 9.2 then it is recognised by a definable alternating Turing machine. If $\mathbb{A}$ has has quantifier elimination, then a deterministic Turing machine is enough.*

Proof

The assumption that L satisfies item 1 says that L is finitely supported, say by some atom tuple $\bar{a}$, and there is a Turing machine M without atoms which recognises the

set $\{w \in 2^* : w$ represents a word from $L\}$. We claim that an alternating Turing machine can reverse the representation up to $\bar{a}$-automorphisms. More precisely, we claim that an alternating Turing machine can compute a function

$$f : \mathbb{A}^* \to 2^*$$

such that for every $w \in \mathbb{A}^*$, the bit string $f(w)$ represents some word in $\mathbb{A}^*$ that is in the same $\bar{a}$-orbit as w. Once we have proved this, we can compose f with M to get a machine recognising L (this works by the assumption that the language L is invariant under $\bar{a}$-automorphisms).

To compute the function f, we use the assumption that the atoms are effectively oligomorphic. Suppose that the input to f is $\bar{b} \in \mathbb{A}^*$. Let n be the length of the tuple $\bar{a}\bar{b}$. Using the assumption on effective oligomorphism, we can compute finitely many formulas with n variables which describe all orbits of n-tuples. Furthermore, if the atoms admit quantifier elimination, then these formulas are quantifier free. Using Lemma 9.3, we can determine which formula φ describes the orbit of $\bar{a}\bar{b}$. Finally, using the assumption that the atoms have decidable first-order model checking, we can find the first bit string that represents some $\bar{c}$ such that $\varphi(\bar{a}\bar{c})$ holds; this word bit string is the value of the function f on input $\bar{b}$. □

We are now ready to complete the proof of Theorem 9.2. Let $L \subseteq \Sigma^*$ be a language which satisfies condition 1. Like for any definable set Σ, there exists some k and a surjective function

$$f : \mathbb{A}^k \to \Sigma$$

whose graph is a definable set. Extend this function to a partial function

$$f^* : \mathbb{A}^* \to \Sigma^*$$

which is defined only on words of length divisible by k and simply applies f to every block of k letters. It is not difficult to see that the inverse image of L under f^* is also computable. Therefore, by Lemma 9.4, the inverse image of L under f^* is recognised by an alternating Turing machine (and a deterministic one in case $\mathbb{A}$ admits quantifier elimination). Using nondeterminism, one can guess a word in $\mathbb{A}^*$ that maps to the input word via f^*, and then run the machine from Lemma 9.4 on this guessed word. This completes the proof of Theorem 9.2.

Exercise 92. Call two structures *equivalent* if they have the same universe and the same automorphism group. Show that following conditions are equivalent for every effectively oligomorphic structure $\mathbb{A}$:

1. nondeterministic Turing machines recognise the same languages as alternating ones;

2. $\mathbb{A}$ is equivalent to a structure where the vocabulary is finite, and for every first-order formula (with free variables) one can compute an equivalent existential one (i.e. one which uses only existential quantifiers in prenex normal form).

9.2 Determinism is weaker than nondeterminism

In Section 9.1 we showed that alternating Turing machines are computationally complete. When the atoms admit quantifier elimination, which is the case e.g. for equality atoms, then already nondeterministic Turing machines are computationally complete. In this section we show that in the equality atoms, deterministic and nondeterministic Turing machines have different expressive power.

Theorem 9.5 *Assume that the atoms are* $(\mathbb{N}, =)$*. There is a definable alphabet* Σ *and a language* $L \subseteq \Sigma^*$ *that is recognised by a nondeterministic definable Turing machine, but which is not recognised by any deterministic definable Turing machine.*

Recall that our notion of recognition is the one corresponding to semi-decidable languages, i.e. the Turing machines are not required to terminate on rejected inputs. In the proof of the theorem, we will see that the nondeterministic machine which recognises L runs in polynomial time, i.e. on every accepted input it has some accepting run with a polynomial number of computation steps. A consequence of the theorem is that, in the equality atoms, NP is not contained in the class of deterministic semi-decidable languages, and hence also NP≠P. However, since computation with atoms is so different from computation without atoms, Theorem 9.5 is unlikely to shed new light on the power of nondeterminism without atoms.

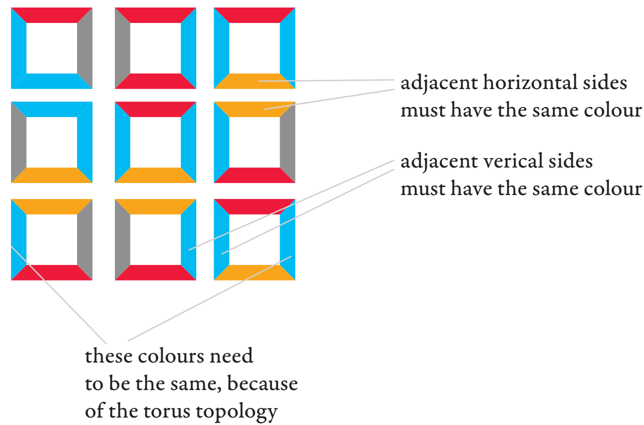
The rest of Section 9.2 is devoted to proving Theorem 9.5.

Tilings. In our proof we a type of tiling problem, and therefore we begin by defining terminology for tilings. If C is a set of colours, then a *tile over colours* C is an element of C^4, which is visualised as a square with each side coloured by a colour from C:

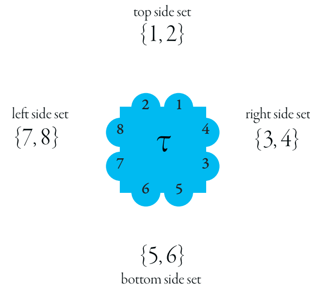
We arrange tiles in square grids with a torus topology, i.e. with the last row adjacent to the first row, and likewise for columns. For $n \in \mathbb{N}$, let us define

$$n \times n \stackrel{\text{def}}{=} \underbrace{\{0, 1, \ldots, n-1\}}_{\text{columns}} \times \underbrace{\{0, 1, \ldots, n-1\}}_{\text{rows}}.$$

The adjacency relation is defined using arithmetic modulo n, i.e. the left neighbour of a grid position is obtained by decrementing the first coordinate modulo n, likewise for right, upper and lower neighbours. If C is a set of colours, then an $n \times n$ *tiling over colours* C is defined to be any function from $n \times n$ to tiles over colours C. A tiling is called *consistent* if adjacent pairs of tiles have the same colour on the shared side, as explained in the following picture for 3×3:



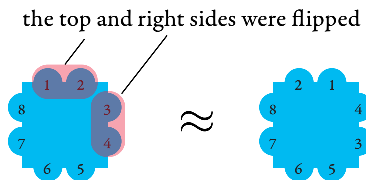
The separating language. For the proof of Theorem 9.5, we will discuss tilings where the set of colours is the set $\mathbb{A}^{(2)}$ of ordered pairs of distinct atoms. To define the separating language, the key is the following equivalence relation on tiles. For a tile τ over colours $\mathbb{A}^{(2)}$, define a *side set* to be a set of two atoms that can be found on one of its four sides, as explained in the following picture:



Define a *flip* of τ to be one of the four tiles that can be obtained from τ by choosing some side set and applying the atom automorphism which swaps the pair of atoms from this side set. We will only apply flips when τ belongs to

$$\Sigma \quad \stackrel{\text{def}}{=} \quad \text{tiles over colours } \mathbb{A}^{(2)} \text{where all four side sets are pairwise disjoint.}$$

and therefore a flip can be seen as affecting only one of the sides. Consider two tiles from Σ to be $\approx$-equivalent if one can go from one to the other by doing an even number of flips. Here is a picture of equivalent tiles, where two flips were used:



An equivalence class of $\approx$ class consists of 8 tiles: 0 flips can be done in 1 way, 2 flips can be done in 6 ways and 4 flips can be done in 1 way. Define $\Sigma_{/\approx}$ to be the set of $\approx$-tiles; this is a definable set. We are now ready to define the separating language. The reason for the name CFI is explained in the bibliographic notes at the end of the section.

Definition 9.6 (CFI property) *Define an $n \times n$ $\approx$-tiling to be a function*

$$\mathcal{T} : n \times n \to \Sigma_{/\approx}.$$

We say that $\mathcal{T}$ satisfies the CFI *property if there exists a consistent tiling*

$$\mathcal{S} : n \times n \to \Sigma$$

such that $\mathcal{T} = \mathcal{S}/_{\approx}$, where is defined by $x \mapsto [\mathcal{S}(x)]_{\approx}$.

Formally speaking, the separating language required for Theorem 9.5 should be a set of words, and not $\approx$-tilings. Therefore we assume some convention on linearly ordering the tiles in an $\approx$-tiling, e.g. the tiles are ordered lexicographically, first by columns then by rows. Under such a convention, an $n \times n$ $\approx$-tiling can be encoded (uniquely) as a word of length n^2 over the alphabet $\Sigma_{/\approx}$. The separating language from Theorem 9.5 is defined to be the set of all encodings of $\approx$-tilings that satisfy the CFI property.

The language L is clearly recognised by a nondeterministic Turing machine. The work alphabet of the machine is $\Sigma_{/\approx} \cup \Sigma$ plus some additional symbols that are used as markers. Given an input word representing some $\approx$-tiling $\mathcal{T}$, the machine uses nondeterminism to guess the consistent tiling $\mathcal{S}$ which witnesses the CFI property. Then, it deterministically checks if the adjacency constraints of a consistent tiling are satisfied by $\mathcal{S}$. This computation can be done in a polynomial number of steps.

L is not recognised by any deterministic Turing machine. To finish the proof of Theorem 9.5, it remains to show that L is not recognised by any deterministic Turing machine. We begin by discussing a natural doubt the reader might have at this point. Given an input representing an $\approx$-tiling $\mathcal{T}$, there are only finitely many (even if exponentially many) possibilities for choosing the witness $\mathcal{S}$ as in Definition 9.6. Why not then use a deterministic algorithm that exhaustively enumerates all the possibilities? The problem is that such an algorithm cannot be implemented as a deterministic Turing machine. The intuitive reason is that even if there are only eight elements in an equivalence class $\tau \in \Sigma_{/\approx}$, one cannot choose deterministically any one of them (i.e. there is no notion of the "first" or "second" element of the equivalence class) to write it down on the tape.

We now proceed to give a formal proof of why language L is not recognised by any deterministic Turing machine. This will be a consequence of Lemma 9.8 below, which says that a deterministic Turing machine, unlike the CFI property, is insensitive to flipping tiles in an $\approx$-tiling.

We begin by giving the definitions that are used in lemma.

For tile $\sigma \in \Sigma$, define $[\sigma]$ to be the tile over colours $\binom{\mathbb{A}}{2}$ which is obtained from σ by forgetting the order information about the pair on each side. It is not difficult to see that $\sigma \approx \sigma'$ implies $[\sigma] = [\sigma']$, and hence it is meaningful to define $[\tau]$ for an $\approx$-tile τ by

$$[\tau] \stackrel{\text{def}}{=} [\sigma] \quad \text{for some, equivalently every, } \sigma \text{ in the equivalence class } \tau$$

We call a $\approx$-tiling $\mathcal{T}$ *weakly consistent* if the tiling

$$[\mathcal{T}] : n \times n \to \text{tiles over colours } \binom{\mathbb{A}}{2} \quad \text{defined by } x \mapsto [\mathcal{T}(x)]$$

is a consistent tiling over colours $\binom{\mathbb{A}}{2}$, and furthermore each atom appears in at most two tiles. If $\mathcal{T}$ is weakly consistent then every atom either does not appear in $\mathcal{T}$, or appears exactly twice, in adjoining side sets corresponding to some edge.

Define the *flip* of an $\approx$-tile $\tau \in \Sigma_{/\approx}$ to be the set

$$\bar{\tau} \stackrel{\text{def}}{=} \{\sigma \in \Sigma : \text{doing an odd number of flips on } \sigma \text{ yields a tile in } \tau\ \}.$$

It is not difficult to see that the above set is also an $\approx$-tile, i.e. flipping can be viewed as an operation on $\approx$-tiles. This operation is an involution, since flipping twice has no effect. The following lemma formalises the statement that the CFI property is sensitive to flips.

Lemma 9.7 *If $\mathcal{T}$ is an $n \times n$ weakly consistent $\approx$-tiling, then for every $x \in n \times n$, the following $\approx$-tiling is weakly consistent and violates the* CFI *property:*

$$\mathcal{T}_x(y) = \begin{cases} \overline{\mathcal{T}(y)} & \text{if } y = x \\ \mathcal{T}(y) & \text{otherwise} \end{cases}$$

Proof
Flips do not modify side sets, and since weak consistency is defined entirely in terms of sides sets, it follows that doing a flip preserves the property of being weakly consistent. To show that a flip results in a violation of the CFI property, we use a parity argument. For $\mathcal{S} : n \times n \to \Sigma$ define the *conflict set* to be the set of edges e in the $n \times n$ grid such that the colours of the two sides adjoining on e are different. Using this terminology, an $\approx$-tiling $\mathcal{T}$ satisfies the CFI property if and only if there exists some $\mathcal{S}$ which has an empty conflict set and such that $\mathcal{T}$ is the $\approx$-equivalence class of $\mathcal{S}$. The key observation is that $\mathcal{S} \approx \mathcal{S}'$ implies that the conflict sets have the same parity (i.e. size modulo two); and furthermore making one flip makes this parity change. □

We are now ready to prove the main lemma which witnesses that L is not recognised by any definable deterministic Turing machine. We use the following notation: if $\mathcal{T}$ is an $\approx$-tiling and M is a definable deterministic Turing machine whose input alphabet $\Sigma_{/\approx}$, then we write $M_{\mathcal{T}}$ for the unique computation of M on the word representing $\mathcal{T}$. We use the formalisation of computations from (12) at the beginning of Section 9, i.e. a computation is a square array of cells, each one containing a letter

of the work alphabet, or a pair (letter of the work alphabet, state of the machine). The following lemma, together with Lemma 9.7 immediately implies that no deterministic definable Turing machine can recognise the language L, thus completing the proof of Theorem 9.5.

Lemma 9.8 *For every definable deterministic Turing machine M with input alphabet $\Sigma_{/\approx}$ there exists $k \in \mathbb{N}$ with the following property. Let*

$$\mathcal{T} : n \times n \to \Sigma_{/\approx} \qquad \textit{with } n > 2k$$

be weakly consistent. Assuming the notation $\mathcal{T}_x$ defined in Lemma 9.7, the following holds for every $i, j \in \mathbb{N}$:

$$M_{\mathcal{T}}(i,j) = M_{\mathcal{T}_x}(i,j) \qquad \textit{for all } x \in n \times n \textit{ with at most } k^2 \textit{ exceptions} \qquad (*)$$

Proof
Let Γ be the work alphabet of the machine, and let Q be its states. Choose k such that for every element s of

$$\Gamma \cup \{\text{blank}\} \cup (\Gamma \cup \{\text{blank}\}) \times Q \qquad (13)$$

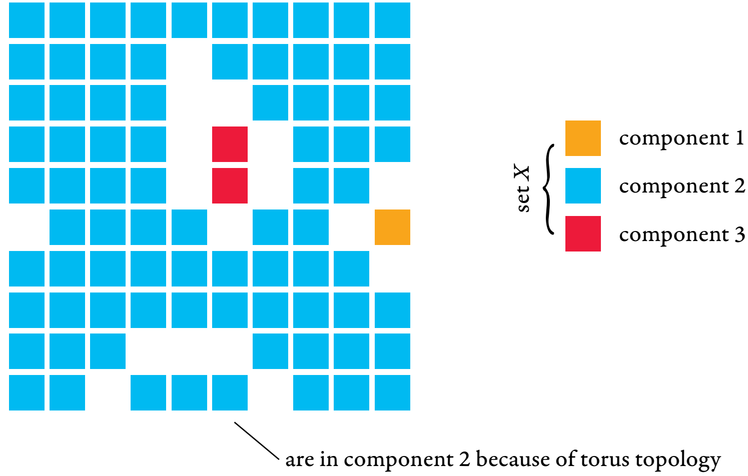
there exists a tuple of at most $k/2$ atoms which supports both s and the Turing machine M. Such a number can be chosen, because M has some fixed finite support, while s ranges over an orbit finite set, and hence there is a common upper bound for the size of supports needed for s.

We prove (*) by induction on i. For the induction base of $i = 0$, we observe that the contents of a cell in time $i = 0$ depend only on the value of the input in at most one grid position, and hence (*) holds with at most one exception.

For the induction step, suppose that (*) is true for $i - 1$ and consider the case of i. In the computation of a Turing machine, the contents of a cell in time i are uniquely determined by the contents of at most two cells in time $i - 1$: the cell in the same column (offset from the beginning of the tape), plus possibly the contents of the unique cell in time $i - 1$ which contains the head of the machine. Hence, using the induction assumption we can conclude the following weaker version of (*), which uses $2k^2$ exceptions instead of k^2:

$$M_{\mathcal{T}}(i,j) = M_{\mathcal{T}_x}(i,j) \qquad \text{for all } x \in n \times n \text{ with at most } 2k^2 \text{ exceptions} \qquad (**)$$

In the rest of this proof, we bring down the number of exceptions to k^2. To do this, we will talk about connected components in the square grid after removing some grid positions. (The square grid refers to the tiling, and not the computation of the machine.) To make these notions precise, we view $n \times n$ as a graph, where the vertices are grid positions, and two grid positions are connected by an edge if they are adjacent in the (torus) grid topology. For a subset $X \subseteq n \times n$, define its connected components to be the connected components in the subgraph of $n \times n$ induced by X. Here is a picture of a set X together with its partition into connected components:



We now resume the proof of the implication from (**) to (*). By choice of k, we can find a tuple of atoms $\bar{a}$ which has size at most $k/2$ and supports both the Turing machine M and $M_{\mathcal{T}}(i,j)$, the latter element belonging to (13). Define the following set of grid positions:

$$Z = \{z \in n \times n : \text{some atom from } \bar{a} \text{ appears in } \mathcal{T}(z)\}.$$

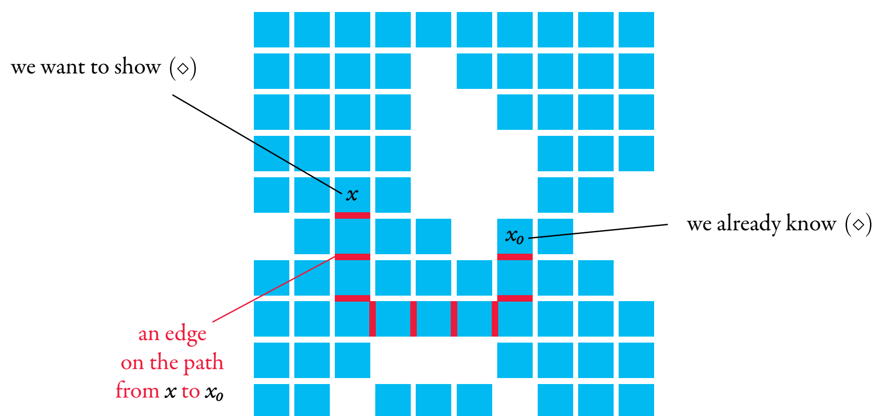
The set Z has at most k grid positions, by the assumption that every atom appears in at most two grid positions from $\mathcal{T}$. From the assumption that $n > 2k$ (actually $n > k$ is enough for this part of the argument) and a straightforward analysis of connectivity in an $n \times n$ grid, one can conclude that the graph corresponding to $n \times n - Z$ has a connected component which contains all grid positions from $n \times n$ with at most k^2 exceptions. Let X be this connected component. Because X omits at most k^2 grid positions, to prove (*), it will suffice to show that

$$M_{\mathcal{T}}(i,j) = M_{\mathcal{T}_x}(i,j) \tag{$\diamond$}$$

holds for every $x \in X$. Observe first a weaker existential version of the above: there exists some $x_0 \in X$ which satisfies ($\diamond$). This is because we have

$$\underbrace{k^2}_{\text{number of exceptions in (**)}} < \underbrace{n^2 - k^2}_{\text{size of } X}$$

which in turn follows from the assumption that $n > 2k$. Let us now show that ($\diamond$) holds for every $x \in X$, not necessarily equal to x_0. Since X is connected and disjoint from Z, in the graph corresponding to $n \times n$ there is a path which goes from x to x_0 and avoids grid positions from Z. Here is a picture:



By the assumption that $\mathcal{T}$ is weakly consistent, to every edge e of $n \times n$ we can associate an unordered set of atoms $S_e \in \binom{\mathbb{A}}{2}$, such that the sets S_e are disjoint for different edges. Define π to be the atom automorphism which flips each unordered set S_e, with e ranging over edges on the path from x to x_0. For each tile except those corresponding to x, x_0, the automorphism flips an even number of side sets, and hence we have:

$$\mathcal{T}_x = \pi(\mathcal{T}_{x_0}). \tag{14}$$

Recall the tuple of atoms $\bar{a}$. The path from x to x_0 was chosen so that all of the sets S_e are disjoint with $\bar{a}$, and therefore π is a $\bar{a}$-automorphism. Because $\bar{a}$ was chosen so that it supports both the machine M and the value $M_{\mathcal{T}}(i,j)$, we have

$$\begin{array}{rcl}
M_{\mathcal{T}_x}(i,j) & = & \text{(by (14))} \\
M_{\pi(\mathcal{T}_{x_0})}(i,j) & = & (\bar{a} \text{ supports } M) \\
\pi(M_{\mathcal{T}_{x_0}}(i,j)) & = & (x_0 \text{ satisfies } (\diamond)) \\
\pi(M_{\mathcal{T}}(i,j)) & = & (\bar{a} \text{ supports } M_{\mathcal{T}}(i,j)) \\
M_{\mathcal{T}}(i,j) & &
\end{array}$$

which completes the proof of (◇) for x and thus also proves the lemma. □

Exercise 93. In the proof of Theorem 7.9, we used an input alphabet which consisted of 8-tuples of atoms modulo some equivalence relation. Improve the proof to use 6-tuples modulo some equivalence relation.

Exercise 94. Assume the equality atoms. Show that if $k \leq 3$ and the input alphabet Σ is k-tuples of atoms modulo some equivariant equivalence relation, then every nondeterministic Turing machine over input alphabet Σ can be determinised.

Exercise 95. Assume the equality atoms and consider the alphabet

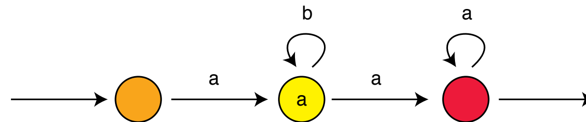
$$\{\{\{a,b,c\},\{d,e,f\}\} : a,b,c,d,e,f \text{ are distinct atoms}\}.$$

Show that Turing machines over this input alphabet do not determinise.

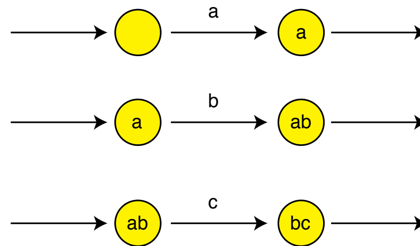
Bibliographic notes for Section 9. Turing machines with atoms were introduced in [13], which also contained the proof of Theorem 9.5. The separating language in Theorem 9.5 is a variant of the Cai-Fürer-Immerman construction [55], it is also motivated by a construction from model theory [21, example on p. 819]. A further study of determinisation of Turing machines was done in [36]; in particular Section 5.1 of that paper gives a solution for the generalisation of Exercise 94 to the optimal value of $k = 5$. Theorem 9.2 on the computational universality of alternating Turing machines is new in these notes.

Part IV
Solutions to the exercises

Solution to Exercise 1.
Consider language 4, i.e. the first data value appears again. The automaton stores the first data value in its unique register and then waits for a repetition to enter an accepting sink state. Here is the picture:



Consider now language 5, i.e. every three consecutive data values are pairwise distinct. The automaton uses two registers to store the last two data values. There is only one control state. Here is the picture:



The unique control state is the yellow state shown above, and thus different occurrences of the yellow state should be seen as self-loops. The picture depicts three kinds of self-loops in this unique control state: a self-loop which goes from zero defined registers to one defined register, a self-loop which goes from one defined register to two defined registers, and a self-loop from two defined registers to two defined registers.

Solution to Exercise 2.
The language is $\{abc : a, b, c \in \mathbb{A} \text{ are distinct}\}$. After reading ab, the automaton should be in the same configuration as after reading ba. This example would go away if automata would have registers that can store unordered pairs of data values. But then we could consider the following language, where addition is done modulo 3:

$$\{a_0 a_1 a_2 a_i a_{i+1} a_{i+2} : a_0, a_1, a_2 \in \mathbb{A} \text{ are distinct}\}.$$

To have a minimal automaton for the above language, we would need registers that store triples of atoms modulo cyclic permutations. Groups other than $\mathbb{Z}_3$ could also be used in examples.

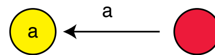
Solution to Exercise 3.
Consider the language

$$\{abc^n : a, b, c \in \mathbb{A} \text{ are distinct and } n \in \mathbb{N}\}$$

One control state and two registers are necessary and sufficient. The automaton begins by loading the first two data values into the two registers. Then the automaton loads c into one of the registers, say the first one. However, one needs to make a design decision: should the second register be erased or not? Both choices lead to nonisomorphic automata. This example would go away if we would allow a register automaton to have a different number of registers depending on the control state.

Solution to Exercise 4.
Language 2 says that some data value appears twice. After reading sufficiently many letters, a deterministic register automaton will necessarily forget one of the previously read letters, in the sense that it will not be in any register. This letter can be read again. How arguments of this type should be written formalised can be seen in the solution to the next exercise, Exercise 5.

Solution to Exercise 5.
The language alphabet is $\mathbb{A}$ and the language, call it L, is "the data value in the last position is only used in the last position". In other words, this is the reverse of the language 4. (And this exercise also shows that nondeterministic automata without guessing are not closed under reverses.) With guessing, the language L can easily be recognised, by simply reversing all arrows in the automaton for language 4 from Exercise 1. The guessing corresponds to this reversed arrow:



Let us prove that L is not recognised by any nondeterministic automaton without guessing. Toward a contradiction, suppose that L is recognised by an automaton with guessing. Let n be strictly bigger than the number of registers. The word $a_1 \cdots a_{n+1}$ consisting of $n + 1$ distinct data values belongs to the language, and hence must admit an accepting run. Decompose this accepting run as $\sigma \cdot t$ where t is the last transition, which reads the letter a_{n+1}, and σ is the rest of the run, which reads the letters $a_1 \cdots a_n$. Since the automaton is not guessing, none of the configurations in the run σ contains a_{n+1}. Furthermore, by assumption on n being greater than the number of registers, some $a \in \{a_1, \ldots, a_n\}$ does not appear in the last configuration of σ. Let π be a bijection of the data values which swaps a with a_{n+1}. Applying π to σ yields a new run $\pi(\sigma)$ which has the same last configuration as σ, since the swapped data values are not present in that configuration. Therefore $\pi(\sigma) \cdot t$ is also an accepting run, but the word it accepts contains the last letter a_{n+1} twice.

Solution to Exercise 6.
Instead of storing a data value that does not appear in the input, use an undefined register with a special marker stored in the control state.

Solution to Exercise 7.

1. - PSPACE membership. A nondeterministic PSPACE algorithm can guess the accepted word. If the automaton has n registers, then data values that are numbers $\{0, \ldots, 2n\}$ are enough.
   - PSPACE hardness. The problem is already hard for automata with a one letter alphabet (therefore the word is uniquely determined by its length). If the state space is n-tuples of atoms, then an arbitrary vector of $n-1$ bits can be encoded by the pattern in which the coordinates $2, \ldots, n$ are equal to the first coordinate. Therefore, one can think of the state as coding vector of $n-1$ bits, which can be used to store the tape contents of a Turing machine. A quantifier-free formula of size polynomial in n can be used to describe the transitions of the machine.

2. - NP hardness. We reduce from the following problem: given a formula

     $$\varphi(a_1, \ldots, a_n, b_1, \ldots, b_n)$$

     which is a Boolean combinations of equalities and inequalities, decide if there is a satisfying assignment where all a_i are pairwise different and all b_i are pairwise different. This is an NP-hard problem, because the pattern of equalities between $\bar{a}$ and $\bar{b}$ can be used to encode an arbitrary vector of n bits (say that bit i is true if and only if the vectors $\bar{a}$ and $\bar{b}$ agree on coordinate i). The above problem is at least as hard as emptiness for register automata, even when there are three orbits of reachable configurations. Indeed, suppose that the automaton has two control states q_0 and q_1, one initial and final, and three orbits of reachable configurations:

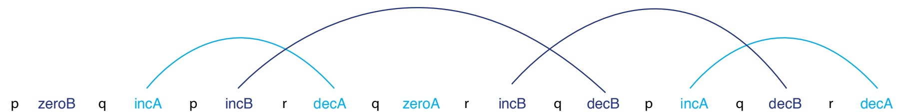
     $$\underbrace{q_0(\bot, \ldots, \bot)}_{\text{orbit 1}} \qquad \underbrace{q_0(\overbrace{a_1, \ldots, a_n}^{\text{distinct data values}})}_{\text{orbit 2}} \qquad \underbrace{q_1(\overbrace{b_1, \ldots, b_n}^{\text{distinct data values}})}_{\text{orbit 3}}$$

     The formula φ could be used as a guard in a transition that goes from the second orbit to the third orbit.
   - NP membership. Consider a graph, where the vertices are orbits of the state space, and there is an edge from orbit Q_1 to orbit Q_2 if and only there is some transition from some state in Q_1 to some state in Q_2. Because the automaton is equivariant, the following conditions are equivalent
     (a) There exists a state q_1 in orbit Q_1 and a state q_2 in orbit Q_2 such that some transition leads from q_1 to q_2 in one step.
     (b) For every state q_1 in orbit Q_1 there exists a state q_2 in orbit Q_2 such that some transition leads from q_1 to q_2 in one step.

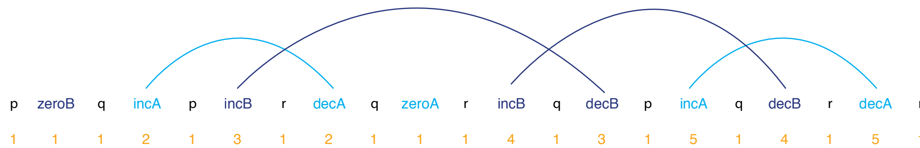
It follows that the automaton is nonempty if and only if the graph described above contains a path from some orbit in the initial states to some orbit in the accepting states. Necessarily such a path has length bounded by the number of orbits. By testing quantifier-free formulas for satisfiability, one can test this in NP.

3. PTIME membership. We do the same argument as in NP membership, only this time the edges of the graph can be computed in polynomial time.

Solution to Exercise 8.
We prove that the (non-)halting problem for Minsky machines reduces to this universality. Recall that a Minsky machine has a finite set of states, two counters storing natural numbers, and a set of transitions which can increment the counters, decrement them and test them for zero. It is an undecidable problem to decide, given a Minsky machine and two control states p, q, if the machine admits a run that goes from p with both counters empty to q with both counters empty. We can view a run of a Minsky machine as a sequence which alternates between control states and counter operations, in a way consistent with the transition relation, as in the following picture:



The counter operations are valid if for every counter $c \in \{A, B\}$, one can pair (the arcs in the picture above) the increments and decrements on counter c such that the increment comes before, and there is no zero test in between. Such a run with a pairing can be encoded as a data word, by adding a unique data value for each arc and using some special data value for positions that are not on arcs (i.e. states or zero tests), as in the following picture:



We claim that a nondeterministic automaton with one register (but with guessing) can recognise the set of data words that are not the encoding of an accepting run with a pairing, and hence undecidability of universality follows in the same way as in Theorem 1.3. The most interesting type of problem is that some arc is wrong: for this the automaton guesses some data value d at the beginning, and checks that this data value is either not used exactly two times, or the first use is not an increment, or the second use is not a decrement of the same counter, or in between there is a test for zero on the appropriate counter.

Solution to Exercise 9.
One can write a formula which is true exactly in the encodings of runs of Turing machines as used in Theorem 1.3. Alternatively, one can write a formula which is true exactly in the encodings of runs of Minsky machines as used in Exercise 8.

Solution to Exercise 10.
Let $\mathcal{A}$ be an alternating register automaton, and define the *dual* of $\mathcal{A}$ to be the same automaton but where we swap universal states with existential states, and we swap accepting states with nonaccepting states. We claim that $\mathcal{A}$ accepts a word if and only if its dual rejects.

We prove that for every configuration c and input data word w, the automaton $\mathcal{A}$ accepts w starting in c if and only if the dual rejects w starting in c. The proof is by induction on the length of the input. For the induction base of empty inputs, we use the fact that accepting and nonaccepting states are swapped. Let us do the induction step. Suppose that the input is aw for some letter a and remaining input w. If the configuration c uses an existential state, then saying that $\mathcal{A}$ accepts aw from c means that there is some transition (c, a, d) such that $\mathcal{A}$ accepts w from d. By the induction assumption, the dual rejects w from d. Since c is universal in the dual, it follows that the dual rejects aw from c, since there is some transition which leads to rejection. The case when c uses a universal state in $\mathcal{A}$ is done the same way.

Solution to Exercise 11.
The right-to-left implication is immediate, because infinite antichains and infinite strictly decreasing sequences are both examples of infinite sequences without infinite monotone subsequences. Let us prove the remaining implication, i.e. in a well quasi-order every infinite sequence has an infinite monotone subsequence.

Let $x_1, x_2, \ldots$ be some sequence in a well quasi-order. Consider the set of minimal elements that appear in the sequence. This set must be finite up to equivalence in the quasi-order, since otherwise we would have an infinite antichain. Furthermore, for every element in the sequence there must be some smaller or equal element that is minimal, since otherwise we would have an infinite strictly decreasing sequence. Cut off a finite prefix of the sequence where all minimal elements are found up to equivalence, and reapply the argument, and continue doing this forever. In the limit we get a partition of the sequence into finite factors

$$\underbrace{x_1, \ldots, x_{i_1}}_{\text{factor 1}}, \underbrace{x_{i_1+1}, \ldots, x_{i_2}}_{\text{factor 2}}, \ldots$$

such that every element from outside the first factor is greater or equal to some element that appears in the previous factor. We can view this factorisation as a directed acyclic graph on the indices $\{1, 2, \ldots\}$ which has an edge from i to j if $x_i \leq x_j$ and i, j are in consecutive factors. This directed acyclic graph has finite degree, because factors are finite, and it has arbitrarily long paths. Therefore it must have an infinite path by König's lemma.

Another solution uses Ramsey's theorem. Take some infinite sequence $x_1, x_2, \ldots$ and colour each pair $i < j$ with "smaller", "bigger or equal" or "incomparable",

depending on the relationship of x_i and x_j. By Ramsey's theorem, there is an infinite subsequence where all pairs get the same colour. This colour has to be "bigger or equal", since the other possibilities would imply an infinite antichain or descending sequence.

Solution to Exercise 12.
Using Exercise 11 it suffices to show that every infinite sequence in $\mathbb{N}^d$ has an infinite monotone subsequence. This is shown by induction on d. The induction base of $d = 1$ is easy to see. For the induction step, consider a sequence

$$x_1, x_2, \ldots \in \mathbb{N}^{d+1}.$$

By the induction assumption, there is an infinite subsequence such that the projection onto the first coordinate is monotone. By induction assumption again, that subsequence has an infinite subsequence where the projection onto the remaining coordinates is monotone, and the result follows.

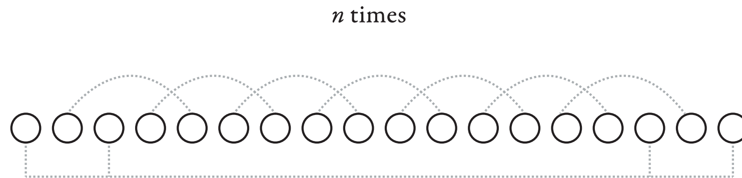
Solution to Exercise 13.
See [51, Exercise 1.10]

Solution to Exercise 14.
The same proof as without order. The only difference is that order-preserving bijections are used in the definition of the quasi-order, and the Higman order is used to show that this is a well quasi-order. More specifically, when proving the variant of Lemma 1.6, instead of using vectors of natural numbers indexed by subsets of Q, we use sequences of subsets of Q, ordered by the Higman ordering.

Solution to Exercise 15.
Using Theorem 1.10 and the Higman ordering on configurations.

Solution to Exercise 16.
This solution is by Klin and Lasota. Let W be the set of words like the one depicted on the following picture, with circles denoting consecutive data values, and dotted lines denoting equality:

n times



Note that the data value in the first letter is special, because it appears four times, and all other atoms appear two times. We claim that if w and v are words in W of different lengths, then w is a not in the same orbit as any subsequence of v. In other words, there is no mapping f from positions of w to positions of v which preserves

the order and equality on data values. Indeed, such a mapping would have to map the first position to the first position (because the first letter contains the special data value that appears four times), and therefore also the third position to the third position. It follows that the second position must be mapped to the second position, and therefore also the fifth position to the fifth position. Arguing inductively, we see that the i-th position needs to be mapped to the i-th position. In other words, w needs to be mapped to a prefix of v. This cannot be, because, the last position of w is mapped to the last position of v.

Solution to Exercise 17.
Consider the set W of words in the solution to Exercise 16. Let $W_P \subseteq W$ be the subset of words that have a prime number of different atoms. Finally, let L be the upward closure of W_P under the Higman order. We claim that this language is not recognised by a nondeterministic register automaton. Otherwise, such an automaton would need to tell the difference between words from W that have prime and non-prime length. By choosing some non-computable set of numbers instead of the prime numbers, we can get a language that is not computable.

Solution to Exercise 18.
Storing the data value from position i in the register can be seen as storing a pointer to position i. The automaton can increment such pointers, test them for equality, and it can move its head to a pointer. Using this one can implement simple arithmetic on pointers.

Solution to Exercise 19.
It will be easier to work with a slightly more general model, called *two-way automata with regular lookaround*, where the transitions can ask about regular queries about the sequence of labels to the left (or to the right) of the head. For example, the automaton could empty its register conditionally on the property "the number of b labels to the left of the head is even". From now on, when talking about two-way register automata we assume it has one register, it is nondeterministic, but it is allowed to use regular lookaround.

A configuration if a two-way register automaton is called *local* if the register is either empty or its content is equal to the data value under the head. Call a two-way register automaton local if every change of registers is done only in local configurations (i.e. the automaton can either load the current data value into the register assuming the register was previously empty, or it can empty the register assuming that the register previously stored the data value under the head). One first shows that every two-way register automaton can be made local without affecting the expressive power on data words with pairwise distinct data values. For this, the automaton nondeterministically guesses the last local configuration before emptying the register and does the emptying at that moment.

It remains to prove the exercise for two-way register automata that are local. For

a data word

$$\frac{b_1}{a_1}\frac{b_2}{a_2}\cdots\frac{b_n}{a_n}$$

and control states p, q, we say that the automaton admits a (p, q)-loop in position $i \in \{1, \ldots, n\}$ if it can start in position i in the local configuration (p, a_i) and then do a finite number of transitions that do not change the register and lead back to position i in the local configuration (q, a_i). It is not difficult to see that the existence of a (p, q)-loop depends only on the label under the head and the regular properties of the labels to the left and right of the head. Therefore, the instead of doing a (p, q)-loop, the automaton could simply do an ϵ-transition conditional on some regular lookaround. After eliminating (p, q)-loops this way, we are left with a two-way automaton which has the property that whenever it loads something into a register, it empties the register in the next step. For such automata, the register is superfluous, and we are left with a two-way automaton without registers, which recognise only regular properties of the labels.

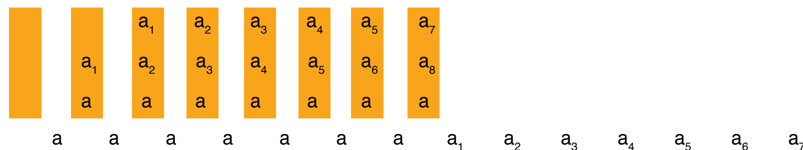
Solution to Exercise 20.
This solution comes from the Master's thesis of Tomasz Wysocki. Consider the following language over $\mathbb{A}$:

$$\{a^n a_1 \cdots a_n : n \in \mathbb{N} \text{ and } a, a_1, \ldots, a_n \in \mathbb{A} \text{ are all distinct}\}$$

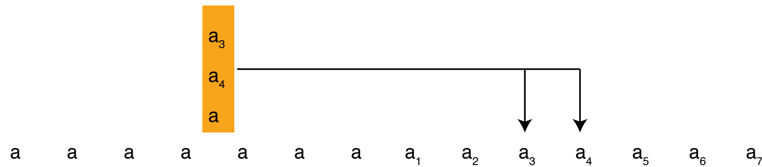
Let us first argue that an alternating register automaton without guessing cannot recognise the language. After reading a prefix of the form a^n, the bag can only have a in its registers. Since there are finitely many possibilities for such bags, there must be some $n < m$ such that the set of reachable bags after reading a^n is the same as the set of reachable bags after reading a^m. Therefore, if the automaton accepts $a^n a_1 \cdots a_n$, then it also accepts $a^m a_1 \cdots a_n$.

Let us recognise this language with guessing. An alternating automaton can easily check that a word is of the form $a^n a_1 \cdots a_m$ for distinct data values $a, a_1, \ldots, a_m$. The challenge is to check that $n = m$. Since languages recognised by alternating automata are closed under intersection, we assume that the input is of the form $a^n a_1 \cdots a_m$.

We only present the main idea using pictures. The automaton has three registers. A main thread of the automaton will read the first n letters, and after reading the i-th letter it will be in a configuration with the initial state and register values a, a_{i-1}, a_i as in the following picture (the orange boxes represent these configurations, with the first two boxes being corner cases):



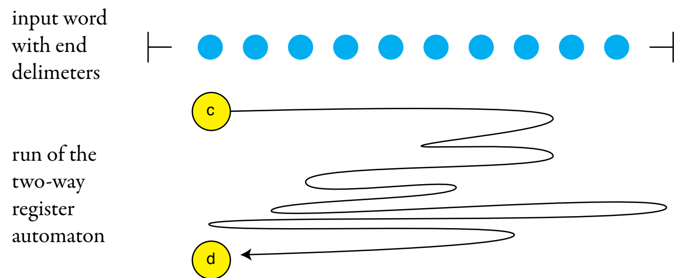
The contents of the registers are above are guessed, but they are verified using alternation: the initial state is universal, and in each step it spawns off a parallel thread that checks if the current configuration corresponds to two consecutive data values in the future, as in this picture:



Solution to Exercise 21.
This solution comes from the Master's thesis of Tomasz Wysocki. Consider a two-way nondeterministic automaton $\mathcal{A}$, where the states are Q and the registers are R. For two configurations of this automaton

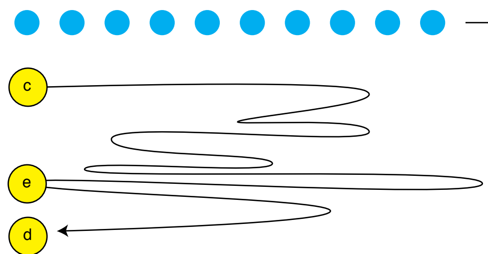
$$c, d \in Q \times (\mathbb{A} \cup \{\bot\})^R$$

we say that a word admits a (c, d)-loop if there is a run of the automaton which begins in configuration c, ends in configuration d, and never tries to move to the left beyond the first position of the word. Here is the picture, note how the run is allowed to revisit the first position or the end delimiter $\dashv$ but it is not allowed to see the start delimiter $\vdash$.



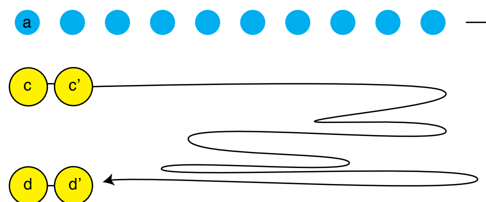
The crucial point is to recognise loops: we will sketch that there is an alternating register automaton, such that if is initialised in a that stores both c and d, then it accepts if and only if there is a (c, d)-loop. Once loops are recognised, it is not difficult to simulate the two-way automaton (one needs to deal with the initial configuration and visiting the start marker $\vdash$.) To recognise loops, we observe that a data word admits a (c, d)-loop if and only if one of the following conditions holds:

- There is some intermediate configuration e such that the word admits a (c, e)-loop and an (e, d)-loop, as in this picture.



To check this, the simulating alternating register automaton does an ϵ-transition where it guesses e and temporarily stores it in the registers. Next it universally branches by into threads for (c, e) and (e, d). Guessing is crucial, because e might contain data values from the future of the word, and ϵ-transitions are used because the two-way automaton might revisit the first position an unbounded number of times.

- The loop does not revisit the first position, as in the following picture:



The simulating alternating register automaton guesses the two configurations c', d', subject to the transition requirement, and advances to the next position.

Solution to Exercise 22.
We want a language that is two-way deterministic, also one-way nondeterministic, but not one-way alternating without guessing. This language is:

$$\{w \in \mathbb{A}^* : \text{some letter appears exactly once}\}.$$

The language is clearly recognised by a one-way nondeterministic automaton, by guessing the letter which appears exactly once. Let us now find a deterministic two-way automaton which does this language. The automaton implements the following procedure:

1. Put the head on the first letter.
2. Check if the letter under the head appears exactly once. If yes, accept immediately, otherwise return the head to its previous position (this can be done by a subroutine which first searches to the left for a duplicate, then searches to the right for a duplicate, and returns after finding the first duplicate).

3. If the head is on the last position, reject, otherwise move the head one step to the right and goto 2.

It remains to show that the language cannot be done by an alternating one-way automaton without guessing. This, honestly speaking, is just a conjecture.

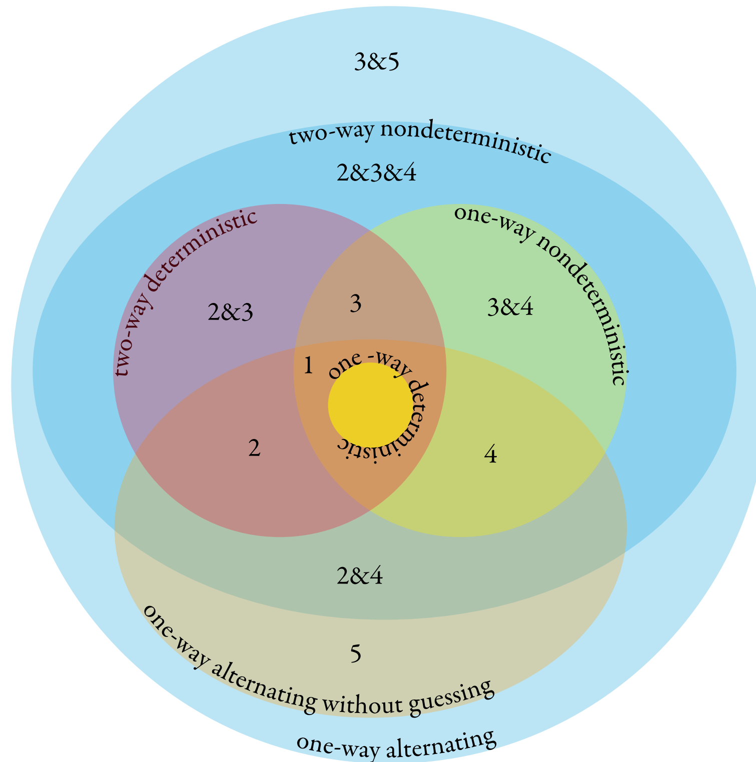
Solution to Exercise 23.
We want a language that is one-way nondeterministic and one-way alternating without guessing, but not two-way deterministic. For this, consider the set of even length sequences of data values

$$a_1 b_1 \cdots a_n b_n$$

such that there is a path from 1 to n in the graph whose vertices are $\{1, \ldots, n\}$ and where the edge relation contains all pairs $i \to j$ such that $i < j$ and $b_i = a_j$. This language is clearly seen to be recognised by a one-way nondeterministic register automaton without guessing (and therefore also by an alternating one). However, if the language were recognised by a two-way deterministic register automaton, then the language would be in deterministic logspace. However, every instance of directed graph reachability can be encoded as a membership question in this language, and therefore we would get that directed graph reachability is in deterministic logspace, thus implying that logspace can be determinised.

Solution to Exercise 24.
We want a language that is one-way alternating without guessing but which is not two-way nondeterministic. We use the same type of graph problem as in Exercise 23, except that instead of graph reachability we use alternating graph reachability. Since alternating graph reachability is complete for polynomial time, the language cannot be done by a nondeterministic two-way automaton, since otherwise nondeterministic logspace would be equal to polynomial time.

Solution to Exercise 25.
Suppose that L is a language that can be done by model A but not B, and K is a language that can be done by model A but not C. Define $L\&K$ to be the concatenation of L and K separated by a fresh symbol. As long as A, B, C are one of the six models in the figure, then the language $L\&K$ can be done by model A but neither by B or C. Using this idea, we can find examples for all coloured areas as in the following picture:



Solution to Exercise 26.
If there was closure under Kleene star, then we would have undecidable emptiness, by finding a data automaton recognising the encodings of computations of Minsky machines used in Exercise 8. Since data automata are closed under intersections, it suffices to find data automaton recognising just one counter with zero tests. If there was closure under Kleene star, then we could check one counter with zero tests: the zero tests can be performed only when the star proceeds to the next iteration.

Solution to Exercise 27.
Suppose that the transitions are

$$\delta_1, \ldots, \delta_n \in \mathbb{Z}^d.$$

There is a run (which can use negative coordinates) that goes from $v \in \mathbb{Z}^d$ to $w \in \mathbb{Z}^d$ if and only

$$v = w + a_1\delta_1 + \cdots + a_n\delta_n \qquad \text{for some } a_1, \ldots, a_n \in \mathbb{N}$$

This is an instance of integer linear programming, and it is known that such instances can be solved in NP. Another answer is that integer linear programming is a special case of Pressburger arithmetic, which is decidable.

Solution to Exercise 28.
Languages recognised by data automata are closed under inverse images of the following operations on data words: "remove the first position" and "keep only positions divisible by k". Therefore, we can apply Lemma 2.6 to get the desired result.

Solution to Exercise 29.
Let us use the name *enriched data automaton* for the model from this exercise, and the name *standard data automaton* for the original model. To prove the exercise, we introduce an intermediate model, called a *semi-enriched data automaton*. In semi-enriched model, there is some k such that only the following information is stored about each block of question marks: the exact length if the block has length $\leq k$, and the remainder modulo k if it the block has length $\geq k$. It is not difficult to see that the enriched and semi-enriched models have the same expressive power. To show that the semi-enriched model has the same expressive power as the standard one, we use Exercise 28 and a labelling of every position by its offset from the beginning modulo k.

Solution to Exercise 30.
With the more powerful model from this exercise, one can recognise computations of counter machines as used in Exercise 8. This would contradict Theorem 2.5 that emptiness is decidable for data automata.

Solution to Exercise 31.
A position is called *opening* if it is the first chosen position in its interval, and a *closing* position if it is the last chosen position in its interval. The following lemma characterises the language in the statement of the exercise in terms of a condition that can clearly be recognised by a data automaton. Therefore, to solve the exercise it remains to prove the lemma.

Lemma 2.7 *A data word belongs to the language in the exercise if and only if:*

1. *every class string satisfies the following expression:*

$$\Big(\big(\underbrace{\mathit{open}}_{\text{opening but not closing}}\ \overbrace{\mathit{middle}^*}^{\text{remaining cases}}\ \underbrace{\mathit{close}}_{\text{closing but not opening}}\big) + \underbrace{\mathit{clopen}}_{\text{opening and closing}} + \underbrace{\bot}_{\text{not chosen}}\Big)^*$$

2. *one can colour the intervals with four colours so that:*
   (a) *for every opening position, the previous position with the same data value does not exist or is in an interval with a different colour;*
   (b) *for every closing position, the next position with the same data value does not exist or is in an interval with a different colour.*

Proof
We begin with the bottom-up implication. Suppose that conditions 1, 2 hold. We show membership in the language:

- *All chosen positions in the same interval have the same data value.* By induction on the left-to-right order on positions, we prove that every chosen position x has the same data value as all earlier chosen positions in its interval. If x is an opening position, then this statement vacuously true, since there are no earlier chosen positions in the same interval. Assume then that x is chosen but not opening. By condition 1, x cannot be the first position in its class. Let y be the previous position in the class of x. We need to show that y is in the same interval as x. By condition 1, y is a chosen but not closing position. Therefore, there must be a closing position in the interval of y, call it z, which is strictly after y. We cannot have $y < z < x$ since then we could apply the induction assumption to z and show that it is in the same class as x, contradicting the choice of y as the previous position in the class. Therefore $z \geq x$, and thus x is in the same interval as y.

- *There is no non-chosen position which has the same data value as some chosen position in the same interval.* Consider an interval. If the interval has no chosen position, the condition is vacuously true. Otherwise, let d be the data value in the chosen positions, which is unique by the previous item. There cannot be any non-chosen position in the interval with data value d that is before the opening position, since otherwise we would get a contradiction with 2a). A symmetric argument holds for non-chosen positions after the closing position. Between the opening and closing position there cannot be non-chosen positions by condition 1.

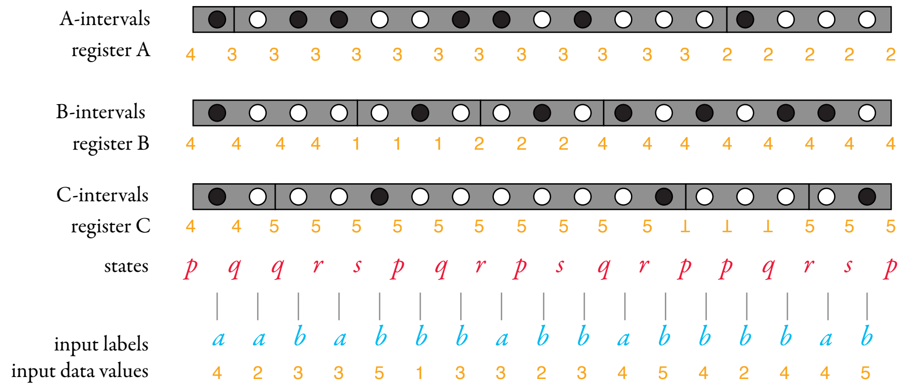
We now show that top-down implication. Condition 1 is easy to see, so we focus on condition 2. We say that two intervals I and J are in conflict if I contains the class predecessor (i.e. previous position in the same class) of the opening position in J. Condition 2a) says that conflicting intervals have different colours. The key observation is that the conflict relation is a forest, because every interval has at most one opening position, and every opening position has at most one class predecessor. Every forest can be coloured with two colours so that no edge is monochromatic, which shows that two colours are enough to satisfy 2a). A symmetric argument shows that two colours are enough to satisfy 2b), and therefore the product colouring with four colours will satisfy both 2a) and 2b). □

Solution to Exercise 32.
For the sake of this exercise, we consider a model of register automaton where undefined registers are not allowed. The initial configuration has the initial state and the first data value in all the registers. It is not difficult to see that this model is equivalent to the original model of register automata.

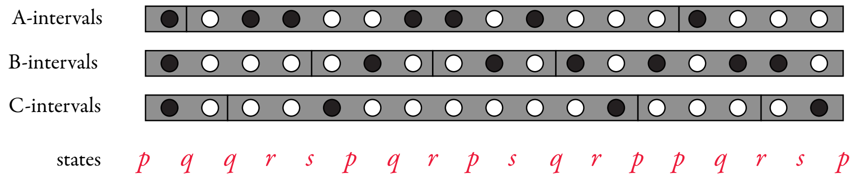
Take some nondeterministic register automaton where undefined registers are not allowed, in the sense described above. Without loss of generality, we assume that it is weakly guessing in the sense of Exercise 6. Consider a run of this automaton.

For a register r, an *r-interval* is a maximal connected set of positions in the input word such that every transition in the interval has the same contents of r in its source configuration. Define the *r-chosen* data value of an r-interval, which may be

undefined, to be the contents of register r that is used throughout the interval (in the source configurations). Call a position r*-chosen* if the input data value is equal to the r-chosen data value of the containing r-interval. Here is a picture of a run for an automaton with registers {A, B, C} together with the corresponding intervals and their chosen positions.



In the picture above, the chosen positions are marked by black circles and the non-chosen positions are marked by white circles. To describe the run, the data automaton uses nondeterminism to guess this part of the above picture:



Using the solution to Exercise 31, the data automaton checks for each register r that for every r-interval, all r-chosen positions have the same data value, and all non-r-chosen positions have a different data value than the r-chosen ones. Since the automaton is weakly guessing, every r-interval contains some r-chosen position. The above picture is sufficient to reconstruct the entire run including the register contents, in a way which can be checked by the finite states of the transducer in the data automaton.

Solution to Exercise 33.
If the run of the transducer is given explicitly in the data word, and every position is labelled by the state in the run of an automaton recognising the class language, then the correctness of such a labelling can be checked by a formula of the logic.

Solution to Exercise 34.

1. Every data value appears exactly twice.

2. Let us use the name *middle* for the second appearance of the data value in the first position. Every data value before the middle appears also after or at the middle. Every data value after the middle appears also before the middle.

3. Regardless of the choices by player $\forall$, the following procedure is bound to terminate by reaching the last position in step (c).

   (a) Player $\forall$ chooses a position x before the middle.

   (b) Let x' be the position after or at the middle with the same data value as x.

   (c) If x is the last position, then terminate. Otherwise, player $\forall$ chooses some position $y > x'$.

   (d) Let x be the position before the middle with the same data value as y.

   (e) Goto (b)

It is not difficult to see that the conditions above are satisfied by every word from the language. For the converse, we prove that if a word does not belong to the language, then items 1 and 2 imply that 3 does not hold. Suppose that 1 and 2 hold, which means that the word is of the form

$$a_1 \cdots a_n a_{\pi(1)} \cdots a_{\pi(n)}$$

for some distinct data values $a_1, \ldots, a_n$ and some permutation π of $\{1, \ldots, n\}$. In particular, there must be some $i < j$ such that $\pi(i) > \pi(j)$. In step (a), player $\forall$ chooses a_j before the middle, and in step (b) player $\forall$ chooses a_i after the middle.

Solution to Exercise 35.
For undecidability, we could extend the idea from Exercise 34 to recognise use the encoding of Turing machine computations from Theorem 1.3. For the decidability, we use a data automaton. The data automaton guesses for each position what are the states from which this position would be accepted, i.e. from which states would player $\exists$ win if the game started in that position. Then only a local consistency check is needed, in the spirit of Theorem 2.8.

Solution to Exercise 36.
We refer to the two logics in the exercise as "two variable logic" and "modal logic". Let us only do the more interesting direction from two variable logic to modal logic, the opposite translation is simply a formalisation of the definition. We claim that for every formula of two variable logic $\varphi(x)$ with one free variable there is a formula of the modal logic that selects the same positions. Once this claim has been proved, the result follows, since a sentence (i.e. formula without free variables) of two variable is of the form $\exists x\ \varphi(x)$, and therefore the corresponding formula of modal logic is

$$\varphi' \vee \bigvee_m \langle m \rangle \varphi'$$

where φ' is the formula of modal logic equivalent to $\varphi(x)$ and m ranges over all modalities. The proof of the claim is by induction on formula size. The interesting case is when $\varphi(x)$ uses a quantifier, say

$$\varphi(x) = \exists y \; \psi(x, y)$$

The formula $\psi(x, y)$ is a Boolean combination of formulas that are predicates, negations of predicates, or begin with a quantifier. By converting this Boolean combination to DNF, and distributing disjunction across $\exists$ we can see that $\varphi(x)$ is equivalent to a disjunction of formulas of the form

$$\exists y \bigwedge_{i \in I} \psi_i(x, y)$$

where each ψ_i is either a predicate or its negation, or it begins with a quantifier. Let $J \subseteq I$ be those indices where ψ_i has two free variables, this corresponds to the binary predicates and their negations. Without loss of generality we can assume that if $i \in I - J$, then the free variable of φ_i is y, since otherwise we could move φ_i out of the scope of the quantifier $\exists y$. Therefore we have rewritten the formula as

$$\exists y \quad \bigwedge_{j \in J} \psi_j(x, y) \;\wedge\; \bigwedge_{i \in I - J} \psi_i(y)$$

To the second conjunction we can apply the induction assumption, yielding an equivalent formula of the modal logic, call it ψ. If the first conjunction is inconsistent, then the whole formula can be replaced by some unsatisfiable formula of the modal logic. If the first conjunction is consistent, then it is equal to a disjunction of modalities $m_1 \vee \cdots \vee m_n$, in which case the entire formula is equivalent to

$$\bigvee_{i \in \{1, \ldots, n\}} \langle m_i \rangle \psi$$

Solution to Exercise 37.

There are only four equivariant (having empty support) binary relations on atoms, namely the empty and full relations, the equality relation, and the disequality relation:

$$\emptyset \qquad \mathbb{A} \times \mathbb{A} \qquad \{(a, a) : a \in \mathbb{A}\} \qquad \{(a, b) : a \neq b \in \mathbb{A}\}.$$

It suffices to show that if an equivariant relation contains some equality pair (a, a) then it contains all other equality pairs as well, and if it contains some disequality pair (a, b) with $a \neq b$, then it contains all other disequality pairs as well. The reason is that every equality pair can be mapped to every other equality pair by an automorphism of the equality atoms, likewise for disequality pairs. The reader will easily generalise this argument to show that an n-ary relation is equivariant if and only if it can be defined by a quantifier-free formula that uses only equality.

Solution to Exercise 38.
All of the four equivariant relations mentioned in Example 37 are still valid. (In general, when the atoms gain structure, there are more equivariant sets.) However, there are four new binary relations, which refer to the total order, namely:

$$\{(a,b) : a < b\} \qquad \{(a,b) : a \leq b\} \qquad \{(a,b) : a > b\} \qquad \{(a,b) : a \geq b\}.$$

Observe again these are exactly the binary relations that can be defined by quantifier-free formulas.

Solution to Exercise 39.
By unravelling the definition, the commuting diagram says that

$$(\pi(x), \pi(f(x))) \in f \qquad \text{for every } x \in X$$

which is equivalent to

$$(x, f(x)) \in \pi^{-1}(f) \qquad \text{for every } x \in X.$$

Since applying an automorphism, such as π^{-1}, to the function f results in a function, the above is equivalent to saying that the functions f and $\pi^{-1}(f)$ are identical, for every $\bar{a}$-automorphism π. This is the same thing as saying that f is supported by $\bar{a}$.

Solution to Exercise 40.
Note first that if there would be order in the atoms, then we could use max. We need to show that there is no finitely supported function

$$f : \mathsf{P}_2(\mathbb{A}) \to \mathbb{A}$$

which chooses an element for each set. Toward a contradiction, suppose that f is such a function, and has finite support. Choose a two element set $\{a, b\}$ of atoms that do not appear in the support. Without loss of generality, assume that $f(\{a, b\}) = a$. Let π be the transposition which swaps a with b, and is the identity on all other atoms, in particular on the support of f. This transposition does not change the input of the function, but changes its output, and therefore it does not preserve the function f, contradicting the assumption.

This exercise touches on the origins of sets with atoms, which are in set theory. In 1922, Abraham Fraenkel showed that, when the atoms have equality only, the sets with atoms:

- fail the axiom of choice, as shown in this exercise, but
- satisfy axioms similar to the Zermelo-Fraenkel axioms of set theory.

The axioms satisfied by sets with atoms are not the real Zermelo-Fraenkel axioms, e.g. extensionality fails because every atom has the same elements as the empty set. The independence of the axiom of choice from the real Zermelo-Fraenkel axioms had to wait for Cohen and forcing.

Solution to Exercise 41.
The vertices are ordered pairs of atoms. From a vertex (a,b) there is exactly one edge, which connects it to (b,a). This graph is bipartite, so it admits a two-colouring. A finitely supported two-colouring, say by colours blue and yellow, would give a choice function, namely map a set $\{a,b\}$ to the unique pair in $\{(a,b),(b,a)\}$ which is coloured by blue.

Solution to Exercise 42.
Suppose that $<$ is a partial order, and a,b are atoms outside the support. Choose π to be the transposition that swaps a and b; in particular π is the identity on the support of $<$. It follows that $<$ is preserved when π is applied to its arguments, and therefore $a<b$ is equivalent to $a>b$. By antisymmetry, neither property can hold.

Solution to Exercise 43.
Suppose that R is a binary relation on the atoms with finite support. Let c be the smallest atom in the finite support. If $a_1 < b_1$ and $a_2 < b_2$ are atoms which are smaller than c, then R selects the pair (a_1,b_1) if and only if it selects the pair (a_2,b_2), because these pairs can be mapped to each other by an automorphism of the rational numbers that fixes all rational numbers greater or equal to c. It follows that for atoms smaller than c, the order imposed by R is either that of the rational numbers or its opposite, neither of which is well-founded.

This example goes back to Andrzej Mostowski, who was one of the main figures in sets with atoms, which is why they are sometimes called Fraenkel-Mostowski sets. The example shows that in sets with atoms there exist sets which can be totally ordered, but not in a well-founded way.

Solution to Exercise 44.
This exercise might be connected to [41, Section III.9], but I'm not sure.

We first observe that the choice of enumeration is not important. This is because the topology on bijections does not depend on the enumerations. In other words, the notion of convergent sequence (of bijections) does not depend on the enumeration: a sequence of bijections is convergent if and only if it is pointwise ultimately constant, i.e. for every argument, all but finitely many bijections give the same result.

The equivalence in the exercise says that the following conditions are equivalent:

1. if a sequence of bijections $\pi_1, \pi_2, \ldots$ of atom automorphisms is pointwise ultimately constant, then the sequence of bijections $f_1, f_2, \ldots$ on X defined by $f_n(x) = \pi_n(x)$ is also pointwise ultimately constant.

2. every element of X is finitely supported.

For the bottom up implication, suppose that $\pi_1, \pi_2, \ldots$ is pointwise ultimately constant. To show that $f_1, f_2, \ldots$ is pointwise ultimately constant, take some element $x \in X$. By assumption 2, there is some finite atom tuple $\bar{a}$ that supports x. By assumption on $\pi_1, \pi_2, \ldots$ being pointwise ultimately constant, it follows that all but finitely many of the automorphisms $\pi_1, \pi_2, \ldots$ give the same result on the tuple $\bar{a}$. This implies that all but finitely many of the functions $f_1, f_2, \ldots$ give the same result on x.

For the top-down implication, suppose that some $x \in X$ does not have finite support. Let $a_1, a_2, \ldots$ be an enumeration of $\mathbb{A}$. Since x does not have finite support, it follows that for every $n \in \{1, 2, \ldots\}$ there is some atom automorphism π_n which is the identity on $a_1, a_2, \ldots, a_n$ but is not the identity on x. Consider the sequence

$$\pi_1, \mathrm{id}, \pi_2, \mathrm{id}, \pi_3, \mathrm{id}, \ldots.$$

This sequence is pointwise ultimately constant (its limit is the identity). However, if we apply the atom automorphisms from the sequence to x, then on even numbered positions we will get x, and on even numbered positions we will not get x.

Solution to Exercise 45.
Define $S \supseteq R$ to be those pairs which can be obtained by taking some first coordinate of R and pairing it with some second coordinate of R. The set S is obtained from R by taking the product of the projections of R onto the first and second coordinates. Since projection is an equivariant function, it follows from Fact 3.6 that S is orbit-finite. Choose some tuple $\bar{a}$ which supports both R and S. It is easy to see that the transitive closure does not increase the support, and therefore the transitive closure of R is a subset of S that is union of $\bar{a}$-orbits. Since S is orbit-finite, this union must be finite.

Solution to Exercise 46.
First observe that the assumption that the atoms have finitely many $\emptyset$-orbits is necessary to get the converse. As an example, take some infinite structure without any automorphisms, e.g. $\mathbb{A} = (\mathbb{N}, <)$. In this case every $\bar{a}$-orbit is a singleton, regardless of the choice of the atom tuple $\bar{a}$, and therefore the two conditions in the statement of the theorem are equivalent.

Let us now prove the statement in the exercise. By induction on $n \in \{1, 2, \ldots\}$ we show that $\mathbb{A}^n$ has finitely many $\bar{a}$-orbits for every tuple of atoms $\bar{a}$. The induction base is the assumption from the exercise. Let us now do the induction step, i.e. consider $\mathbb{A}^{n+1}$. Take some n-tuple of atoms $\bar{a}$. By the induction base, there are finitely many $\bar{a}$-orbits in $\mathbb{A}$, which means that there a finite finitely many $\emptyset$-orbits in $\mathbb{A}^{n+1}$ that contain tuples which begin with $\bar{a}$. We have just shown that the mapping

$$f : \mathbb{A}^n \to \mathsf{P}(\mathbb{A}^{n+1})$$

which maps a tuple $\mathbb{A}$ to those $\emptyset$-orbits in $\mathbb{A}^{n+1}$ that contain a tuple beginning with $\bar{a}$ always produces finite families of subsets. Since every tuple in $\mathbb{A}^{n+1}$ must begin with some tuple in $\mathbb{A}^n$, it follows that the family of $\emptyset$-orbits in $\mathbb{A}^{n+1}$ is

$$\bigcup_{\bar{a} \in \mathbb{A}^n} f(\bar{a})$$

It is also easy to see that f is equivariant, and therefore its value only depends on the $\emptyset$-orbit of the argument. Therefore, in the union above we could take only one tuple $\bar{a}$ for every $\emptyset$-orbit of $\mathbb{A}^n$, of which there are finitely many by induction assumption. Therefore, the family of $\emptyset$-orbits in $\mathbb{A}^{n+1}$ is finite, as a finite union of finite families. We have shown that $\mathbb{A}^{n+1}$ has finitely many $\emptyset$-orbits. By the assumptions that the

two conditions in Theorem 3.3 are equivalent, it follows that $\mathbb{A}^{n+1}$ has finitely many $\bar{a}$-orbits for every tuple of atoms.

Solution to Exercise 47.
Suppose that X is orbit-finite. Choose some support $\bar{b}$ of X. For every tuple of atoms $\bar{a}$, there are finitely many $\bar{a}\bar{b}$-orbits of X. If an element $x \in X$ is supported by $\bar{a}$, then its $\bar{a}\bar{b}$-orbit is a singleton, hence there are finitely many elements of X supported by $\bar{a}$.

Solution to Exercise 48.
Consider the set of all non-repeating tuples of atoms. Since tuples can have arbitrarily large dimensions, and atom automorphisms preserve dimensions of tuples, the set is not orbit-finite. Nevertheless, a given tuple of atoms can only support finitely many tuples, namely those tuples that are contained in it (and possibly reordered).

Solution to Exercise 49.
Call a set $\bar{a}$-representable if it satisfies the conclusion of the exercise. By oligomorphism, if a set is orbit-finite and supported by $\bar{a}$, then it is a finite union of $\bar{a}$-orbits. It is not difficult to see that $\bar{a}$-representable sets are closed under disjoint unions, by encoding additional information in the equality type of a tuple of atoms. Therefore, it suffices to show that every $\bar{a}$-orbit is $\bar{a}$-representable. Take then some $\bar{a}$-orbit, i.e. a set of the form

$$X = \{\pi(x) : \pi \text{ is an } \bar{a}\text{-automorphism}\}$$

for some set with atoms x. Let $\bar{b}$ be some support of x, and let n be the dimension of $\bar{b}$. The relation

$$f = \{(\pi(\bar{b}), \pi(x)) : \pi \text{ is an } \bar{a}\text{-automorphism}\}$$

is an $\bar{a}$-supported partial function from $\mathbb{A}^n$ to X. Functionality, i.e. one argument can only give one result follows from the definition of support. Define the *kernel* of f to be the partial equivalence relation on $\mathbb{A}^n$ which identifies two tuples if they are both in the domain of f and have same values under f. It is not difficult to see that the $\mathbb{A}^n$ quotiented by this kernel is in $\bar{a}$-supported bijection with X.

Solution to Exercise 50.
Choose some $x \in X$ and some atom tuple $\bar{b}$ which supports x. Consider the set of pairs

$$\{(\pi(\bar{b}), \pi(x)) : \pi \text{ is an } \bar{a}\text{-automorphism.}\}$$

This set of pairs is a surjective function from atom tuples to X, by the assumption that $\bar{b}$ supports x. The domain of the function is the $\bar{a}$-orbit of $\bar{b}$, which is an orbit-finite set. From Fact 3.6 it follows that the range of the function is orbit-finite.

Solution to Exercise 51.
Suppose that X and f are supported by an atom tuple $\bar{a}$. Since orbit-finite sets are

clearly closed under finite unions, it suffices to consider the case when X is one $\bar{a}$-orbit. Choose some $x \in X$, and let $\bar{b}$ be an atom tuple which supports $f(x)$. Since $f(x)$ is orbit-finite, it is a union of finitely many $\bar{b}$-orbits, and therefore one can choose $y_1, \ldots, y_n$ so that every element of $f(x)$ is obtained from some y_i by applying some $\bar{b}$-automorphism. It follows that an element belongs to the union in the exercise if and only if it can be obtained by taking some y_i, applying some $\bar{b}$-automorphism, and then applying some $\bar{a}$-automorphism. The result then follows from Exercise 50.

Solution to Exercise 52.
Suppose that X is an orbit-finite set, and $f : X \to X$ is an injective function. It is not difficult to see that if $\bar{a}$ is a support of f, then f maps injectively $\bar{a}$-orbits to $\bar{a}$-orbits. In particular, since X has finitely many $\bar{a}$-orbits, then the image of f must have the same number of $\bar{a}$-orbits, and is therefore the whole set X.

Solution to Exercise 53.
Before giving the solution, we remark that Dedekind finiteness can be used to characterise orbit-finite sets, but one needs to use the (finitely supported) powerset. The following theorem, which is given here without proof, was shown by Andreas Blass.

Theorem 3.7 *For every choice of atoms, not necessarily oligomorphic ones, a set is all-support orbit-finite if and only if its powerset is Dedekind finite.*

Let us now solve the exercise. Consider the equality atoms and the set

$$\mathbb{A}^{(*)} \stackrel{\text{def}}{=} \bigcup_n \mathbb{A}^{(n)},$$

i.e. the set of non-repeating tuples of arbitrary lengths. This set is not orbit-finite, yet we claim that it is Dedekind finite, i.e. that every finitely supported injection

$$f : \mathbb{A}^{(*)} \to \mathbb{A}^{(*)}$$

is a bijection. Suppose that f is supported by a finite tuple of atoms $\bar{a}$. For a tuple in $\mathbb{A}^{(*)}$ define its $\bar{a}$-dimension to be the number of atoms in the tuple, not counting the atoms from $\bar{a}$. All tuples in a single $\bar{a}$-orbit have the same $\bar{a}$-dimension, and therefore it makes sense to talk about the $\bar{a}$-dimension of an $\bar{a}$-orbit.

Claim 3.7.1 *For every $\bar{a}$-orbit Z, the image $f(Z)$ is a $\bar{a}$-orbit with the same $\bar{a}$-dimension.*

Proof
The image under f of an $\bar{a}$-orbit in $\mathbb{A}^{(*)}$ is also an $\bar{a}$-orbit. The $\bar{a}$-dimension cannot increase when applying f, since the function is $\bar{a}$-supported, but it cannot decrease as well (since the inverse of f is also $\bar{a}$-supported). □

The key property is that for every $n \in \mathbb{N}$, the set $\mathbb{A}^{(*)}$ has finitely many $\bar{a}$-orbits of $\bar{a}$-dimension n. It follows that for every n, f is a bijection between $\bar{a}$-orbits of $\bar{a}$-dimension n, and therefore f is a bijection.

Solution to Exercise 54.
The left-to-right implication is clear. For the converse implication, if X is not finite, then the family of finite subsets of X is directed but has no maximal elements.

Solution to Exercise 55.
Let us introduce a further condition: (**) there is a maximal element in every set of atoms $\mathcal{X} \subseteq \mathsf{P}X$ which is a chain (i.e. totally ordered by inclusion) and uniformly supported. We will show that orbit-finiteness, (*) and (**) are all equivalent. The implication from (*) to (**) is immediate. For the implication from (**) to orbit-finiteness of X, choose some support $\bar{a}$ of X. If X had infinitely many $\bar{a}$-orbits, then we could construct a uniformly supported infinite chain without a maximal element, by successively adding these orbits. For the implication from orbit-finiteness to (*), suppose that $\mathcal{X}$ is a uniformly supported directed family of subsets of an orbit-finite set X. Let $\bar{a}$ a tuple of atoms that supports every set in $\mathcal{X}$. The union $\bigcup \mathcal{X}$ is a finitely supported subset of X, and therefore must be orbit-finite by oligomorphism. The union partitions into finitely many $\bar{a}$-orbits, call them $X_1, \ldots, X_n$. Every set from $\mathcal{X}$ is simply a union of some of the $\bar{a}$-orbits $X_1, \ldots, X_n$, and therefore $\mathcal{X}$ must contain $X_1 \cup \cdots \cup X_n$, a maximal element.

Solution to Exercise 56.
Let us begin with a counterexample for $(\mathbb{Q}, <)$. The set of all atoms is orbit-finite, but it admits a chain of subsets without a maximal element, namely the family of all downward closed intervals.

We now prove that the statement in the exercise is true in the equality atoms. We will show that (***) is equivalent to (**) from the solution of Exercise 55 and therefore it is equivalent to orbit-finiteness. Actually, we show a stronger property.

Lemma 3.8 *Consider the equality atoms. If a set with atoms* $(X, \leq)$ *is a total order, then some tuple of atoms supports all elements of* X.

Proof
We use the following property of the equality atoms:

> (†) Every finite partial automorphism of the atoms can be extended to a complete automorphism that is the identity on almost all atoms.

The above property is not true in $(\mathbb{Q}, <)$ but it true e.g. in the random graph that will be discussed in Section 6.

Let $\bar{a}$ be a support of both X and the total order, which we denote by $\leq$. We show that every element $x \in X$ is supported by $\bar{a}$. Let then π be some $\bar{a}$-automorphism of the atoms. We need to show that $\pi(x) = x$. Let $\bar{b}$ be a finite support of x (eventually we will show that x is supported by $\bar{a}$). Since supports are closed under adding elements, assume that that all atoms in $\bar{a}$ appear also in $\bar{b}$. By property (†), there must be some automorphism of the atoms σ, which agrees with π on $\bar{b}$, but which is the identity on almost all atoms. Since π and σ agree on the support of x, it follows that $\pi(x) = \sigma(x)$. Also, σ is an $\bar{a}$-automorphism since it agrees with π on $\bar{b}$ which contains all elements of $\bar{a}$.

Since X is supported by $\bar{a}$, it follows that $\sigma(x)$ belongs to X. Since $\leq$ is a total order, x and $\sigma(x)$ must be comparable under $\leq$. Without loss of generality, we assume that

$$x \leq \sigma(x).$$

Since $\leq$ is supported by $\bar{a}$, we can apply the $\bar{a}$-automorphism σ to both sides of the inequality, yielding

$$\sigma(x) \leq \sigma^2(x).$$

By doing this a finite number of times, we get

$$x \leq \sigma(x) \leq \cdots \leq \sigma^n(x)$$

Since σ is the identity on almost all atoms, there must be some n for which σ^n is the identity. Therefore, we see that $x \leq \sigma(x) \leq x$, and therefore $x = \sigma(x)$, which is the same as $\pi(x)$. □

Solution to Exercise 57.
Same proof as for the standard lemma, plus this observation: the depth of a subtree is invariant under applying atom automorphisms.

Solution to Exercise 58.
A simple formalisation (especially if one uses the variant of the definition that uses syntactic equivariance).

Solution to Exercise 59.
A Minsky machine in a special case. The constant 0, even though it is not present in the vocabulary of the atoms, can still be used because definable sets can use constants.

Solution to Exercise 60.
Consider a nondeterministic definable automaton $\mathcal{A}$ which recognises a language L supported by $\bar{a}$. Define a new automaton, which is a disjoint union of all automata of the form $\pi(\mathcal{A})$ where π is an $\bar{a}$-automorphism. This new automaton is also definable. It is also supported by $\bar{a}$. Finally, the recognised language is the same, because all automata in the disjoint union recognise the same language, namely the original language L.

Solution to Exercise 61.
In Exercise 58 we showed that every register automaton is a special case of a definable automaton. Let us show the converse. Suppose that we have a definable automaton $\mathcal{A}$ which recognises an equivariant language over an alphabet $\mathbb{A} \times \Sigma$. Using Exercise 60, we can assume that $\mathcal{A}$ is equivariant. By Exercise 49, we can assume that the state space Q of $\mathcal{A}$ is equal to $\mathbb{A}^n/_\sim$ where $\sim$ is some equivariant partial equivalence relation on $\mathbb{A}^n$. Define a new automaton $\mathcal{B}$ by "de-quotienting" $\mathcal{A}$, i.e. the input

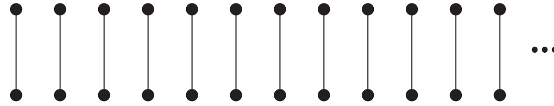
alphabet is the same, the state space is $\mathbb{A}^n$, the transition relation is defined as

$$\underbrace{\bar{a} \xrightarrow{a} \bar{b}}_{\text{in } \mathcal{B}} \qquad \text{iff} \qquad \underbrace{\bar{a}/_{\sim} \xrightarrow{a} \bar{b}/_{\sim}}_{\text{in } \mathcal{A}}$$

and the initial and final states are defined the same way. It is not difficult to see that $\mathcal{B}$ accepts the same word as $\mathcal{A}$. We have just defined an equivariant automaton which recognises the original language, whose state space is exactly $\mathbb{A}^n$. This almost the same thing as a register automaton with one control states and n register. The only difference is that in a register automaton, the initial state is supposed to be of the form (initial control state, completely undefined register valuation) and accepting states are determined based on the control state. These differences can be easily removed, by adding two control states to $\mathcal{B}$.

Solution to Exercise 62.
See [12, Corollary 9.5].

Solution to Exercise 63.
Suppose that the atoms are a graph with infinitely many edges that do not share any nodes.



This structure is oligomorphic, actually it is homogeneous (see Section 6). Every atom is supported by itself, or the other side of its edge.

Solution to Exercise 64.
Let $\bar{a}$ be the least support of the group, i.e. the tuple consisting of theorbit finiteerse, the multiplication operation and the inverse function $g \mapsto g^{-1}$. (Actually,orbit finite supports the multiplication operation, then it supports the universe and the inverse.) For an element g of the group, define $[g]$ to be its least support minus the atoms that appear in $\bar{a}$. It is not difficult to see every elements g, h of the group satisfy

$$[gh] \subseteq [g] \cup [h] \qquad [g^{-1}] \subseteq [g] \tag{2}$$

Take some g in the group which maximises the size $[g]$. Such a maximum exists, since the size of $[g]$ depends on that $\bar{a}$-orbit of g, of which there are finitely many. Since we are dealing with the equality atoms, we can choose an orbit finiteomorphism π so that

$$\pi([g]) \cap [g] = \emptyset \tag{3}$$

We have

$$g = \pi(g)\pi(g)^{-1}g.$$

Combining this with (2) we get

$$[g] \subseteq [\pi(g)] \cup [\pi(g)^{-1}g]$$

Combining this with (3), we get

$$[g] \subseteq [\pi(g)^{-1}g]$$

By maximality of $[g]$ the above is actually an equality. Using a similar reasoning applied to

$$\pi(g)^{-1} = g^{-1}\pi(g)\pi(g)^{-1}$$

we conclude that

$$[\pi(g)] = [\pi(g)^{-1}] \subseteq [g^{-1}\pi(g)] = [\pi(g)^{-1}g] = [g].$$

where the first and second equalities hold because taking the inverse does not affect the value of $[_]$. From (3) it follows that $[\pi(g)]$ is empty. Therefore $[g]$ must also be empty, since $[_]$ commutes with $\bar{a}$-automorphisms. By maximality of $[g]$ it follows that all elements of the group have value $\emptyset$ under $[_]$ which implies that all elements of the group are supported by $\bar{a}$. In an orbit-finite set there can only be finitely many elements with a given support (Exercise 47). Therefore the group is finite.

The same proof would work for some other atoms, e.g. $(\mathbb{Q}, <)$. I do not know if it works for all oligomorphic atoms.

Solution to Exercise 65.
If two tuples of atoms $\bar{a}$ and $\bar{b}$ are in the same orbit, then the sizes of their generated substructures are the same.

Solution to Exercise 66.
Consider a finite directed graph G, and the randomly chosen graph H. With probability one, every partial isomorphism between G and H extends to an embedding. Since there are countably many possibilities for G it follows that with probability one the graph H has the property from Lemma 6.5. By Lemma 6.6 there is one graph that we get with probability one, up to isomorphism, call this the random directed graph. By applying Lemma 6.6 with both $\mathfrak{H}_1$ and $\mathfrak{H}_2$ being the random directed graph, we see that the random directed graph is homogeneous. The age is the set of all finite directed graphs.

Solution to Exercise 67.
Let us revisit the proof in Exercise 56. The proof uses property (*), which fails in the universal equivalence relation or in the random graph. However, the proof only needs a weaker property, namely:

> (**) Every finite partial automorphism f of the atoms can be extended to a complete automorphism π, such that for some $n \in \mathbb{N}$:
>
> $$\pi^n(a) = a \qquad \text{for every } a \text{ in the domain of } f.$$

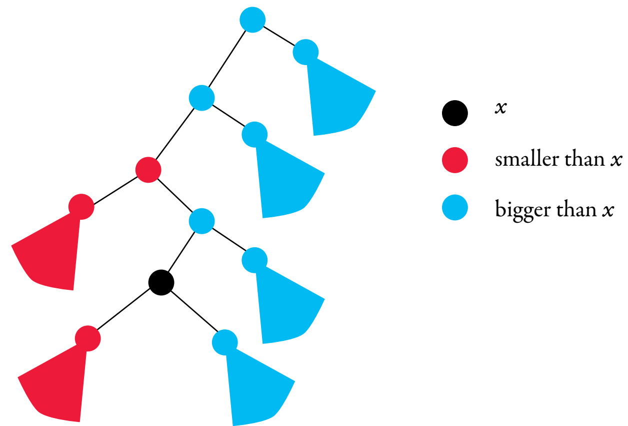
We show that property (**) holds in the Fraïssé limit of undirected graphs. We use a theorem of Hrushovski [32]:

Theorem 6.10 *Every finite (undirected) graph G embeds in some finite graph H such that every partial automorphism of G extends to a complete automorphism of H.*

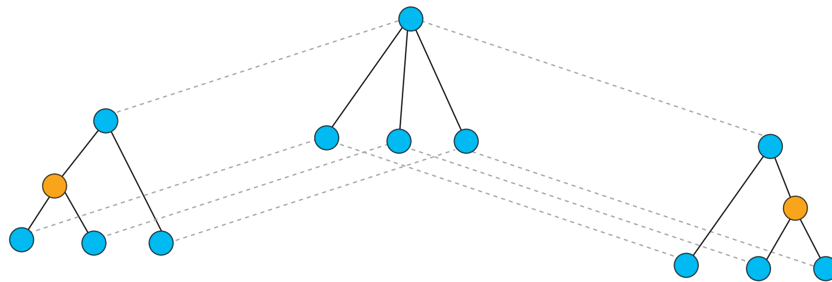
To prove property (**), let G be the subgraph of the random graph induced by the domain and co-domain of f. Apply Theorem 6.10, yielding some H. By the universality property of the random graph, we may assume that H is a finite induced subgraph of the random graph. By the theorem, f extends to some full automorphism of H, and that automorphism extends to a full automorphism π of the random graph, which preserves H as a set. It follows that there is some n such that π^n is the identity when restricted to H.

Solution to Exercise 68.
In a homogeneous structure, two tuples are in the same orbit if they satisfy the same quantifier-free formulas. By the assumption that the vocabulary is relational (i.e. has no function symbols) and finite, up to logical equivalence there are finitely many quantifier-free formulas over a given set of variables, and they can be computed. By the assumption on $\mathscr{A}$ having decidable membership, one can decide which quantifier-free formulas are satisfiable in the Fraïssé limit $\mathbb{A}$. Furthermore, one can effectively eliminate quantifiers, i.e. for every first-order formula (possibly with free variables) over the vocabulary of $\mathbb{A}$, one can compute an equivalent one which is quantifier-free. Using this observation, it follows that the first-order theory of $\mathbb{A}$ is decidable. Furthermore, $\mathbb{A}$ is effectively oligomorphic, in the sense of Exercise 77, since the "same orbit" formula is the quantifier-free formula which checks that the same predicates from the finite vocabulary are satisfied. Therefore, $\mathbb{A}$ satisfies the assumptions of Exercise 77 and is thus an effective structure.

Solution to Exercise 69.
A rational number can be viewed as node in Rabin's tree $\{0, 1\}^*$ as follows

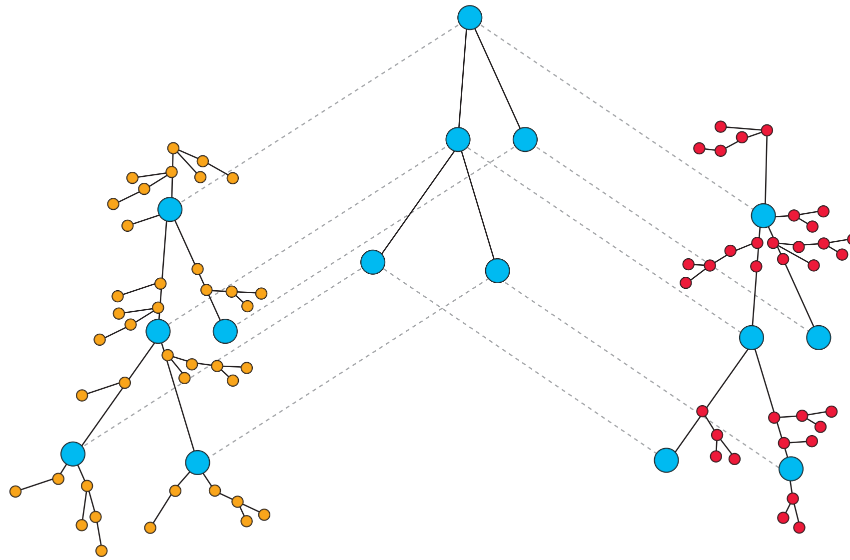


Solution to Exercise 70.
Here is the instance of amalgamation which has no solution:

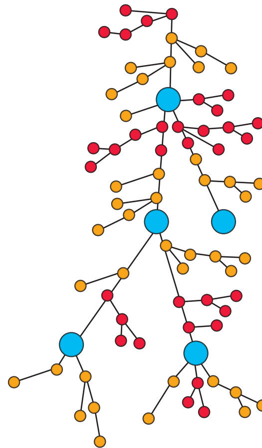


Solution to Exercise 71.
Proof by picture:

an instance of amalgamation



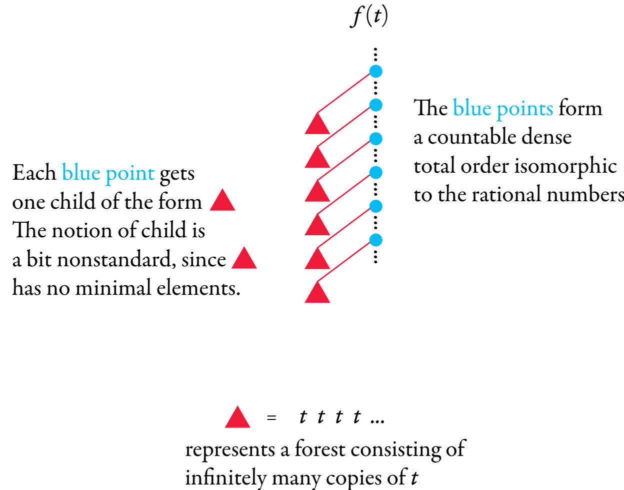
its solution



Solution to Exercise 72.
Fix some atom a. Define the equivalence relation to be $b \sim c$ if the closest common ancestor of b, c is a proper descendant of a.

Solution to Exercise 73.

Solution to Exercise 74.
We will show how this random tree can be interpreted in the complete binary tree using MSO. To prove this, consider a transformation f which inputs a tree t (i.e. a structure with the closest common ancestor function where for each element, the ancestors are a totally ordered) and outputs the tree depicted in the following picture:



Define t_∞ to be a limit of this procedure, i.e. a tree satisfying

$$t_\infty \quad \text{is isomorphic to} \quad f(t_\infty).$$

We will show that t_∞ is the random tree. To show this, we need to show that it is a) homogeneous; and b) it contains every tree as an induced substructure. Both properties are not difficult to show. Using the idea from Exercise 69 and the recursive nature of Rabin's tree, one can show that the structure $(\mathbb{Q}^*, \preceq)$ consisting of sequences of rational numbers ordered lexicographically has decidable MSO theory. The tree t_∞ can be described in terms of $(\mathbb{Q}^*, \preceq)$, with nodes being coded as odd length sequences of rational numbers (i.e. every second level of $(\mathbb{Q}^*, \preceq)$ is used), and the descendant relation defined using an MSO formula $\varphi(x, y)$ which uses the definition of the tree t_∞ in terms of the function f. This implies that t_∞ has decidable MSO theory, since it can be interpreted inside a structure with decidable MSO theory.

Solution to Exercise 75.
The truth value of an MSO formula $\varphi(x_1, \ldots, x_n)$ depends only on the orbit of the free variables. The random directed graph is homogeneous and without functions, and therefore by Exercise 68 one can decide if a first-order formula with free variables has at least one satisfying assignment. Since the tuples which satisfy φ form an equivariant set, this set is definable in first-order logic by Theorem 4.4. If the translation to first-order logic were computable, then one would be able to decide the MSO

theory of the random directed graph. Since the random directed graph contains all finite directed graphs as induced subgraphs, e.g. all directed grids, it has undecidable MSO theory.

Solution to Exercise 76.
See [17, Proposition 2].

Solution to Exercise 77.

Let $\mathbb{A}$ be a structure that is effectively oligomorphic and has decidable first-order theory. Our goal is to extend the vocabulary of the structure with constants $c_1, c_2, \ldots$ such that the the structure $(\mathbb{A}, c_1, c_2, \ldots)$ has decidable first-order theory, and every element of the universe is represented by some constant.

For $k \in \{0, 1, \ldots\}$ we say that a tuple of atoms $a_1 \ldots a_n$ is k-saturated if it is non-repeating, and every k-tuple of atoms is in the same equivariant orbit as some tuple of the form

$$a_{n_1} a_{n_2} \cdots a_{n_k} \qquad \text{for some } n_1, \ldots, n_k \in \{1, \ldots, k\}.$$

If $k = 0$, then the condition is trivially satisfied. If $\mathbb{A}$ is oligomorphic, then every k-saturated tuple can be extended to one which is $(k+1)$-saturated. It follows that there is an infinite sequence of atoms $a_1, a_2, \ldots$ which is ω-saturated in the following sense: for every k, some finite prefix $a_1, \ldots, a_n$ is a k-saturated tuple of atoms.

Claim 7.2.1 *If a sequence of atoms $a_1, a_2, \ldots$ is ω-saturated then the structure $\mathbb{A}$ is isomorphic to its substructure induced by $\{a_1, a_2, \ldots\}$.*

Proof
One shows that Duplicator can win the infinite round Ehrenfeucht-Fraïssé game between $\mathbb{A}$ and the induced substructure, which implies isomorphism. □

Using the assumption that the formulas for the "same orbit" equivalence relation can be computed, it follows that there is some infinite ω-saturated sequence of atoms which is computable in the following sense: for every n, one can compute a first-order formula $\varphi_n(x_1, \ldots, x_n)$ which defines the equivariant orbit of the first n elements in the infinite sequence. Fix some ω-saturated and computable infinite sequence of atoms $a_1, a_2, \ldots$. Let $\mathbb{B}$ be the substructure of $\mathbb{A}$ induced by this sequence, extended with constants $c_1, c_2, \ldots$ representing the atoms $a_1, a_2, \ldots$. By the computable assumption, $\mathbb{B}$ has decidable first-order theory, and by the claim it is isomorphic to $\mathbb{A}$.

Solution to Exercise 78.
Clearly 1 implies 2. Let us show that 2 implies 1. By assumption 2, we can compute the number k of $\emptyset$-orbits of $\mathbb{A}^n$. By Lemma 4.3, each such orbit has a different first-order theory. Therefore, it suffices to find k inequivalent formulas with n free variables, these formulas can be found using brute force.

It is not difficult to see that 1 implies 3: if we can axiomatise each orbit by a formula, then being in the same orbit boils down to satisfying the same axiomatising formula, for which there are finitely many possibilities.

Let us show that 3 implies 2. We want to count the number of $\emptyset$-orbits in $\mathbb{A}^n$. Consider the following procedure. Let $A \subseteq \mathbb{A}^n$ be a finite set of tuples of atoms, which are in pairwise different orbits. Initially, A is empty. Using assumption 3 and decidable model checking, we can decide if there exists a tuple $\bar{a} \in \mathbb{A}^n$ which is in a different orbit than all tuples in A. If there exists no such tuple, then we have found the number of orbits. Otherwise, we can find such a tuple, by enumerating through all possible candidates. We add this tuple to A and continue. The algorithm is bound to stop because of oligomorphism.

Solution to Exercise 79.
Induction on the size of the formula φ. The Boolean operations are straightforward, since definable sets are closed under unions and complementation relative to the definable set A^n. For the quantifiers, it suffices to deal with $\exists$, and this step corresponds the simple observation that if $X \subseteq A^n$ is definable, then so is its projection onto the first $n-1$ coordinates. (The observation is actually a bit tedious if one wants to properly deal with the definition of tuples using Kuratowski pairing.)

Solution to Exercise 80.
Using the Symbol Pushing Lemma.

Solution to Exercise 81.
We first It is not difficult to see that if X is definable, then

$$\{\cup x : x \in X\} \quad \text{and} \quad \bigcup X$$

are both definable sets. Furthermore, both of these sets have smaller rank than X, i.e. smaller maximal nesting of set brackets. Therefore, by induction on rank one shows that the union-member closure is also definable.

Solution to Exercise 82.

Solution to Exercise 83.
Instead of natural numbers, we could use the positive rational numbers, and the answer to emptiness would be the same. This is because a run that uses positive rational numbers can be changed into a run that uses natural numbers, by scaling. After assuming that the counters store positive rational numbers, we end up with a special case of nondeterministic orbit-finite automata, over the total order atoms. (The automaton is not equivariant, since it uses the constant 0.) As we shall prove later on, emptiness for such automata is decidable.

Solution to Exercise 84.
The standard proof works. This is because the standard proof does not change the state space, only it adds transitions, initial states and final states. The new bigger set of transitions is still finitely supported.

Solution to Exercise 85.
Suppose that we have a union

$$\bigcup_{i \in I} L_i$$

where I is an orbit-finite set and each L_i is recognised by an nondeterministic orbit-finite automaton with state space Q_i. Then the union is recognised by an automaton with state space

$$\biguplus_{i \in I} Q_i$$

which is an orbit-finite set by Exercise 51.

Solution to Exercise 86.
Intuitively speaking, the problem is that intersection corresponds to product on automata, and we cannot do orbit-finite products. Here is the counterexample. For every $a \in \mathbb{A}$, the language "a appears at most once" is recognised by a (deterministic) orbit-finite automaton. If we could intersect all these languages, then we would get a nondeterministic automaton for the language "all letters are distinct". By Exercises 61, this would mean that "all letters are distinct" could be recognised by a register automaton, which is not the case.

Solution to Exercise 87.
The problem is that when the automaton reads the first letter of the input, say the unordered set $\{\underline{1}, \underline{2}\}$, then it cannot load any atoms into its registers, since this would require a form of choice.

Solution to Exercise 88.
(TODO) This model is taken from Murawski and Tzevelekos.

Solution to Exercise 89.
Before giving the solution, we point out that without atoms, emptiness is decidable for higher order pushdown automata, even for orders ≥ 3. For undecidability it suffices to have a stack of at most two stacks. We assume that ϵ-transitions are available, which changes the expressive power of the model, but does not influence decidability of emptiness.

We only show that such an automaton can recognise

$$L = \{(w\#)^n : w \in \mathbb{A}^* \text{ has no repetitions and } n \in \mathbb{N}\}$$

over the alphabet $\mathbb{A} \cup \{\#\}$. The same construction can be modified so that the automaton checks that consecutive blocks between # symbols, instead of being equal as in L, are consecutive configurations of a Turing machine.

In a first phase, the automaton puts w into the (first) stack and checks that it has no repetitions. This is done as follows. For every new letter a, the automaton stores a in its state. Then it duplicates the stack, and searches if a appears on the duplicated

stack, destroying the duplicate in the process. If it does not find a on the duplicated stack, it pushes a onto the first stack, and proceeds to the next input letter.

Once it has checked that w has no repetitions, and stored w on the stack, the automaton proceeds to the second phase, which checks that the rest of the input consists of copies of w separated by $\#$ symbols. The second phase is done essentially the same way as the first. For every two consecutive letters a and b in the rest of the input the automaton does the following.

> If $a = \#$ then b must be the first letter of w, which is stored in the state. If $b = \#$, then a must be the last letter of w, which is stored in the state. Finally, suppose that neither a nor b are $\#$. The automaton needs to check that a and b are consecutive letters in w. To do this, the automaton duplicates the stack, and searches through this stack to check that a and b are consecutive symbols on the stack.

Maybe the above undecidability argument shows that our definition of higher-order pushdown automata for atoms is the wrong one. If it is wrong, then which one is right?

Solution to Exercise 90.
The proof is the same as the standard proof for normal sets.

- When transforming a context-free grammar into a pushdown automaton, we do not need any assumptions on the atoms. The classical construction works. The automaton keeps a stack of nonterminals. It begins with just the starting nonterminal, and accepts when all nonterminals have been used up. In a single transition, it replaces the nonterminal on top of the stack by the result of applying a rule. Here it is useful to assume that the grammar is in Chomsky normal form.

- When transforming a pushdown automaton into a context-free grammar, we use the assumption that the atoms are homogeneous over a finite vocabulary, since we need closure of orbit-finite sets under products. The nonterminals are going to be

$$\mathcal{N} = Q \times \Gamma \times Q.$$

  (Here we need the assumption on the atoms, since $\mathcal{N}$ is a product of orbit-finite sets.) The language generated by a nonterminal (p, γ, q) is going to be the set of words which take the automaton from a configuration with state p and γ on top of the stack, to another configuration with state p and γ on top of the stack, such that during the run the symbol γ is not removed from the stack. The rules of the grammar are as in the classical construction; it is easy to see that the set of rules is orbit-finite.

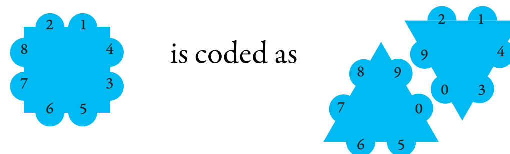
Solution to Exercise 91.
The language is odd length palindromes where the first letter is equal to the middle

letter. If it were generated by an orbit-finite context-free grammar with finitely many terminals (but possibly an orbit-finite set of rules), then the language would have the following property for some tuple of atoms $\bar{a}$ (the support of the hypothetical grammar), which it does not have:

> For every sufficiently long w, there is a decomposition $w = w_1 w_2 w_3$, with w_2 and $w_1 w_3$ nonempty such that
>
> $$w_1(\pi \cdot w_2)w_3$$
>
> is a palindrome for every $\bar{a}$-automorphism π.

Solution to Exercise 92.

Solution to Exercise 93.



Solution to Exercise 94.
By looking at the finitely many possible equivariant definable relations, and then doing a deatomisation procedure in each case.

Solution to Exercise 95.
See [36, Example 2.5 and discussion at the end of Section 5].

References

[1] *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE Computer Society, 2015.

[2] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.

[3] Rajeev Alur, Pavol Černý, and Scott Weinstein. *Algorithmic Analysis of Array-Accessing Programs*, pages 86–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[4] Vince Bárány, Mikołaj Bojańczyk, Diego Figueira, and Pawel Parys. Decidable classes of documents for xpath. In D'Souza et al. [26], pages 99–111.

[5] Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5):702–715, 2010.

[6] Andreas Blass. Power-dedekind finiteness. http://www.math.lsa.umich.edu/ ablass/pd-finite.pdf, 2013.

[7] Mikolaj Bojanczyk. Data monoids. In *STACS*, pages 105–116, 2011.

[8] Mikołaj Bojańczyk. Nominal monoids. *Theory Comput. Syst.*, 53(2):194–222, 2013.

[9] Mikolaj Bojanczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. Towards nominal computation. In *POPL*, pages 401–412, 2012.

[10] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011.

[11] Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. Automata with group actions. In *LICS*, pages 355–364, 2011.

[12] Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10(3), 2014.

[13] Mikołaj Bojańczyk, Bartek Klin, Slawomir Lasota, and Szymon Toruńczyk. Turing machines with atoms. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 183–192, 2013.

[14] Mikołaj Bojańczyk and Slawomir Lasota. An extension of data automata that captures xpath. *Logical Methods in Computer Science*, 8(1), 2012.

[15] Mikołaj Bojańczyk and Slawomir Lasota. A machine-independent characterization of timed languages. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, pages 92–103, 2012.

[16] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3):13:1–13:48, 2009.

[17] Mikołaj Bojańczyk, Luc Segoufin, and Szymon Toruńczyk. Verification of database-driven systems via amalgamation. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 63–74, 2013.

[18] Mikołaj Bojańczyk and Szymon Toruńczyk. Imperative programming in sets with atoms. In D'Souza et al. [26], pages 4–15.

[19] Giuseppe Castagna and Andrew D. Gordon, editors. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 2017.

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

[21] G. Cherlin and A.H. Lachlan. Stable finitely homogeneous structures. *Trans. AMS*, 296(2):815–850, 1985.

[22] Lorenzo Clemente and Slawomir Lasota. Reachability analysis of first-order definable pushdown systems. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, pages 244–259, 2015.

[23] Lorenzo Clemente and Slawomir Lasota. Timed pushdown automata revisited. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015* [1], pages 738–749.

[24] Thomas Colcombet, Clemens Ley, and Gabriele Puppis. Logics with rigidly guarded data tests. *Logical Methods in Computer Science*, 11(3), 2015.

[25] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009.

[26] Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors. *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, volume 18 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[27] Gian Luigi Ferrari, Ugo Montanari, and Marco Pistore. Minimizing transition systems for name passing calculi: A co-algebraic formulation. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, pages 129–158, 2002.

[28] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[29] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[30] Stefan Göller, Christoph Haase, Ranko Lazic, and Patrick Totzke. A polynomial-time algorithm for reachability in branching VASS in dimension one. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 105:1–105:13, 2016.

[31] W. Hodges. *Model Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1993.

[32] Ehud Hrushovski. Extending partial isomorphisms of graphs. *Combinatorica*, 12(4):411–416, 1992.

[33] Florent Jacquemard, Luc Segoufin, and Jerémie Dimino. Fo2($<$, $+1$, $\sim$) on data trees, data tree automata and branching vector addition systems. *Logical Methods in Computer Science*, 12(2), 2016.

[34] Marcin Jurdzinski and Ranko Lazic. Alternating automata on data trees and xpath satisfiability. *ACM Trans. Comput. Log.*, 12(3):19:1–19:21, 2011.

[35] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[36] Bartek Klin, Slawomir Lasota, Joanna Ochremiak, and Szymon Toruńczyk. Turing machines with atoms, constraint satisfaction problems, and descriptive complexity. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 58:1–58:10, 2014.

[37] Eryk Kopczynski and Szymon Toruńczyk. LOIS: an application of SMT solvers. In Tim King and Ruzica Piskac, editors, *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories affiliated with the International Joint Conference on Automated Reasoning, SMT@IJCAR 2016, Coimbra, Portugal, July 1-2, 2016.*, volume 1617 of *CEUR Workshop Proceedings*, pages 51–60. CEUR-WS.org, 2016.

[38] Eryk Kopczynski and Szymon Toruńczyk. LOIS: syntax and semantics. In Castagna and Gordon [19], pages 586–598.

[39] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 267–281, 1982.

[40] Jérôme Leroux. The general vector addition system reachability problem by presburger inductive invariants. *Logical Methods in Computer Science*, 6(3), 2010.

[41] S. Mac Lane and I. Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory*. Springer, 1992.

[42] Ernst W. Mayr. An algorithm for the general petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984.

[43] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. Learning nominal automata. In Castagna and Gordon [19], pages 613–625.

[44] Ugo Montanari and Marco Pistore. Finite state verification for the asynchronous pi-calculus. In *TACAS*, pages 255–269, 1999.

[45] Ugo Montanari and Marco Pistore. History-dependent automata: An introduction. In *SFM*, pages 1–28, 2005.

[46] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Reachability in pushdown register automata. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, pages 464–473, 2014.

[47] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Bisimilarity in fresh-register automata. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015* [1], pages 156–167.

[48] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

[49] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[50] James Schmerl. A decidable ${}_0$-categorical theory with a non-recursive Ryll-Nardzewski function. *Fundamenta Mathematicae*, 98(2):121–125, 1978.

[51] Sylvain Schmitz and Philippe Schnoebelen. Algorithmic Aspects of WQO Theory. Lecture, August 2012.

[52] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, pages 41–57, 2006.

[53] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. 1990.

[54] Tomasz Wysocki. Automaty alternujące z rejestrami na skończonych słowach. Master's thesis, University of Warsaw, 2013.

[55] Jin yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identifications. *Combinatorica*, 12(4):389–410, 1992.