Turing Machines with Atoms

Mikołaj Bojańczyk, Bartek Klin, Sławomir Lasota, Szymon Toruńczyk
University of Warsaw, e-mail: {bojan,klin,sl,szymtor}@mimuw.edu.pl

***Abstract*—We study Turing machines over sets with atoms, also known as nominal sets. Our main result is that deterministic machines are weaker than nondeterministic ones; in particular, P≠NP in sets with atoms. Our main construction is closely related to the Cai-Fürer-Immerman graphs used in descriptive complexity theory.**

I. Introduction

Motivation. Perhaps the first computational complexity result learned by a student of Computer Science is the $n \log n$ lower bound on sorting in the "comparison model". Although widely known, this fact is unusual from the standpoint of mainstream computation theory: in the comparison model, arbitrarily large numbers can be manipulated in a single step of computation, but they can only be accessed by checking whether they are greater, equal or smaller than other numbers. This contrasts with the main tool of computation theory that is Turing machines; there, complex objects such as numbers are normally encoded as strings over a finite alphabet (so that, e.g., comparing two numbers requires several steps of computation), but these encodings are then open to arbitrary manipulation (so that e.g. numbers can be added).

Turing machines. In this paper, we study Turing machines that operate over infinite alphabets that can only be accessed in limited ways. As an initial step, we restrict attention to alphabets whose letters are finite structures built of *atoms* (taken from a fixed countably infinite set) that can only be tested for equality. The set of all atoms is denoted $\mathbb{A}$. Individual atoms will be written down as underlined positive integers $\underline{1}, \underline{2}$, etc; the underlining is used to distinguish the atoms from integers, since atoms have no structure (like order or successor) except for equality.

For example, an input or work alphabet of a Turing machine may contain letters of the following shape:

- atoms themselves,
- (ordered) pairs (or, in general, n-tuples) of distinct atoms,
- (unordered) sets of atoms of size 2.

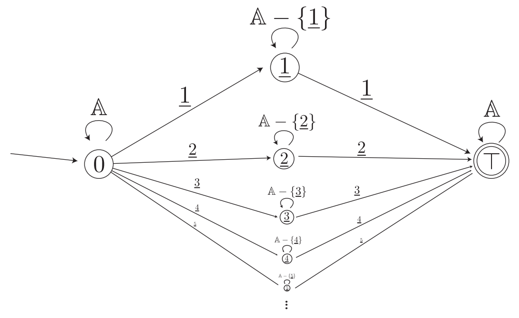
More complex examples are used in the following sections. Note that each shape of letters comes with an obvious action of bijective atom renaming. Sometimes this action is trivial even if the renaming is not; for example, the permutation that swaps $\underline{3}$ and $\underline{5}$ does not alter the set $\{\underline{3}, \underline{5}\}$.

We are interested in Turing machines that operate on alphabets of such shapes, as well as store their letters as parts of their state. The set of letters of a given shape is normally infinite, so we need to speak of machines with infinite state spaces. However, we shall restrict the behaviour of such machines by requiring that their transition relations are invariant with respect to bijective atom renaming. For example, if a machine M in a state that stores a set of atoms $\{\underline{3}, \underline{5}\}$, upon seeing the atom $\underline{3}$ on the tape, moves to a state where just the atom $\underline{5}$ is stored, then M in a state that stores $\{\underline{8}, \underline{2}\}$, upon seeing $\underline{2}$ must move to a similar state where just $\underline{8}$ is stored. This property, formalized later using sets with atoms, corresponds exactly to the intuition of a machine that "only cares for atom equality".

Example I.1. Over the input alphabet of atoms $\mathbb{A}$, consider the language of words where some letter appears at least twice:

$$L = \{a_1 \cdots a_n \in \mathbb{A} \colon a_i = a_j \text{ for some } i < j\}$$

This language is easily recognized by a nondeterministic Turing machine (indeed, a left-to-right nondeterministic automaton) with atoms, with (infinite) state space $\{0, \top\} + \mathbb{A}$ where 0 is initial, $\top$ is accepting and the transition relation is defined by the graph:



Alternatively, this may be seen as a machine with three "control states" $0, \top$ and A, where in the state A a single atom is additionally stored. This is a machine that reads the input word from the left, nondeterministically guesses the first occurrence of a letter that appears more than once, and then checks that it indeed appears afterwards.

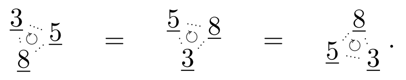
The language L can also be recognized by a deterministic Turing machine that, for each letter in the input word, stores it in the state and checks whether it appears more than once. An extended work alphabet is needed for this to allow marking of the currently processed letter; more importantly, a deterministic machine cannot process the input word in one left-to-right pass. □

The above is quite similar to how classical Turing machines recognize the counterpart of L in the world of finite alphabets without atoms. The next example shows an additional aspect of nondeterminism that appears in the presence of atoms.

Example I.2. Let the input alphabet consist of sets of the form:

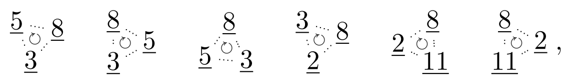
$$\{(a, b, c), (b, c, a), (c, a, b)\} \text{ for } a, b, c \in \mathbb{A} \text{ distinct.} \quad (1)$$

Such a letter is a triple of atoms up to cyclic shift, and it can be visualized as a rotating, but oriented, triangle on a plane, for example:

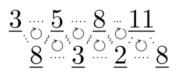


(2)

Consider the language of those sequences of such triples:



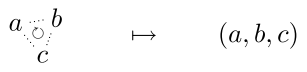
where every two consecutive triples share at least two atoms, that can be glued together in a matching chain like this:



(3)

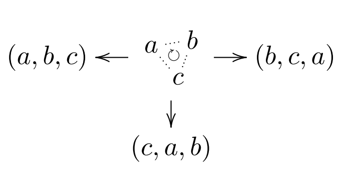
To recognize this language, a Turing machine first nondeterministically "freezes" the leftmost triangle in some position (or, equivalently, chooses an atom from it that shall not touch the second triangle in the chain), and then progresses to the right deterministically, checking that each subsequent letter can be affixed to the emerging chain (if it can, there is a unique way to do it, since the atoms in each letter are distinct).

One might think that a deterministic Turing machine can recognize this language by trying each of the three possible fixed positions of the leftmost letter one by one, much as nondeterminism is resolved in the classical world. However, this is impossible in our model! In particular, a transition function that maps a rotating triangle to a fixed triple of atoms:



(4)

is not invariant with respect to bijective atom renaming. Indeed, the cyclic permutation $(a \mapsto b \mapsto c \mapsto a)$ does not change the triangle, but it does change the resulting triple. Intuitively, a function is unable to distinguish one of the three possible outcomes, as it has only access to equality tests on atoms; therefore "freezing" a rotating triangle in a fixed position is an act of nondeterminism, and it cannot be done by a deterministic machine. On the other hand, a *non*deterministic transition relation:



is fine (i.e., if a triangle and a triple are related then they are so after any bijective atom renaming). □

The language in the above example *can* be recognized by a deterministic machine: one that stores in its state all three atom triples arising from the leftmost letter, and processes them in parallel. However, this does not generalize: one of our main results is that, with a more complex alphabet, deterministic machines with atoms are weaker than nondeterministic ones.

Our contribution. We model atoms by using an alternative model of set theory called sets with atoms (or nominal sets, or Fraenkel-Mostowski sets). Turing machines with atoms are defined by interpreting the standard definition in the alternative model. The focus of our study is on the difference between determinism and nondeterminism. Our main contributions are:

1) Theorem III.1 says that in the presence of atoms, deterministic decidability is weaker than nondeterministic decidability. Even more, there is a language that is decidable in nondeterministic polynomial time, but not deterministically decidable (even not deterministically semi-decidable). In particular, P≠NP in sets with atoms[1], and PSPACE≠NPSPACE. Our proof may be adapted to show that no interesting nondeterministic complexity class is contained in deterministically semi-decidable languages. The main construction used in Theorem III.1 is closely related to Cai-Fürer-Immerman graphs [15].
2) Corollary V.3 says that even though they are weaker than nondeterministic machines, deterministic machines still have some good properties, in particular closure under orbit-finite union, which is a form of guessing a fixed number of atoms.
3) Theorem VI.3 characterises those input alphabets for which deterministic and nondeterministic decidability coincide.
4) Atoms are a natural way to speak of data that can be accessed by an algorithm only in a limited way, e.g. by testing equality. We briefly mention atoms equipped with more structure, such as a linear order. This case turns out to be simpler than the equality atoms; in particular, nondeterminism can be eliminated just as in the classical world. A more interesting example is studied in Theorem VII.2, which shows that checking the linear independence of binary vectors requires exponential time when vectors are equipped only with addition and zero test. (Guassian elimination tests independence in polynomial time, but it uses more than just addition and zero test.)

II. Sets and machines with atoms

We define sets with atoms following [10], and following [4] for orbit-finiteness.

Consider a countably infinite set, denoted by $\mathbb{A}$, whose elements we call *atoms*. For most of the paper, we assume that the atoms have no structure except for equality, and therefore we use the name *atom automorphism* for any permutation of the atoms. Occasionally, we call $\mathbb{A}$ the *equality atoms* (to distinguish from atoms with more structure which will be studied in Section VII; there not all permutations are automorphisms.) A *set with atoms* is any set that can contain atoms or other sets with atoms, in a well-founded way. Formally, sets with atoms are defined by ordinal induction: the empty set is the only set at level 0, and sets at level α either are atoms (which contain no elements) or contain sets at levels smaller than α.

Examples of sets with atoms include:

(a) any classical set without atoms,
(b) an atom $\underline{3}$, an ordered pair of atoms $(\underline{3}, \underline{5})$ (encoded as a set in a standard way, e.g. $\{\{\underline{3}\}, \{\underline{3}, \underline{5}\}\}$),

[1]In [5], we mistakenly conjectured that the P vs. NP problem with atoms is equivalent to the classical one.

(c) $\{(\underline{3},\underline{5},\underline{8}),(\underline{5},\underline{8},\underline{3}),(\underline{8},\underline{3},\underline{5})\}$, i.e. the triple $(\underline{3},\underline{5},\underline{8})$ considered up to cyclic shift,

(d) the set $\mathbb{A}$, the set $\mathbb{A}^n$ of n-tuples of atoms, the set $\mathbb{A}^{(n)}$ of n-tuples of distinct atoms, the set $\binom{\mathbb{A}}{n}$ of sets of atoms of size n, etc.

One can perform standard set-theoretic constructions on sets with atoms, including union, intersection, Cartesian product, powerset etc.

Legal sets with atoms. For X a set with atoms and π an atom automorphism, by $\pi(X)$ we denote the set obtained by application of π to the elements of X (this definition is recursive; formally, this is again defined by ordinal induction). We say that a set $S \subseteq \mathbb{A}$ *supports* X if $X = \pi(X)$ for every π which is the identity on S; such π is called an *S-automorphism*. For example, (a) and (d) above are supported by the empty set, and $\{\underline{3},\underline{5},\underline{8}\}$ supports (c). A set with atoms is called *legal* if it has some finite support, each of its elements has some finite support, and so on recursively. For example, the legal subsets of $\mathbb{A}$ are precisely those which are either finite or cofinite. The full powerset $\mathcal{P}\mathbb{A}$ is not legal, but the set $\mathcal{P}_{\text{fs}}\mathbb{A}$ of finitely supported sets of atoms is legal. Trivially, every element of a legal set is legal.

We are only interested in sets with atoms that are legal. From now on all sets with atoms are assumed to be legal.

Every legal set X has the *least* finite support with respect to set inclusion (see e.g. [10, Prop. 3.4]). By *the* support of X, we implicitly mean the least support.

A legal set supported by the empty set is called *equivariant*. For example, $\mathbb{A}$ is equivariant, but $\{\underline{3}\} \subseteq \mathbb{A}$ is not.

Relations and functions between sets with atoms can be seen as sets with atoms themselves, as their graphs. A relation $R \subseteq X \times Y$ is supported by $S \subseteq \mathbb{A}$ iff xRy implies $\pi(x)R\pi(y)$ for every S-automorphism π. Similarly, a function $f : X \to Y$ is supported by S iff $\pi(f(x)) = f(\pi(x))$, for every S-automorphism π. It follows that $f(x)$ is supported by the union of S with the least support of x.

The general rule is that a relation (function) is supported by S (equivariant) if it can be defined without mentioning any atoms outside of S (resp. any atoms at all). For example, the only equivariant function from $\mathbb{A}$ to $\mathbb{A}$ is the identity, and the only other equivariant relations on $\mathbb{A}$ are the full and the empty relations and the complement of the equality relation. A constant function from $\mathbb{A}$ to $\mathbb{A}$ with value $a \in \mathbb{A}$ is supported by $\{a\}$. The only equivariant functions from $\mathbb{A}^2$ to $\mathbb{A}$ are the projections; the only equivariant function in the other direction is the diagonal. There is no equivariant function from $\binom{\mathbb{A}}{2}$ to $\mathbb{A}$, but

$$\{(\{a,b\},a) \colon a,b \in \mathbb{A}\}$$

is a legal equivariant relation between $\binom{\mathbb{A}}{2}$ and $\mathbb{A}$. (Note that it relates e.g. $\{\underline{3},\underline{5}\}$ both to $\underline{3}$ and $\underline{5}$.)

Orbit-finite sets. For X a set with atoms and $S \subseteq \mathbb{A}$, the *S-orbit of X* is

$$\{\pi(X) : \pi \text{ is an } S\text{-automorphism}\}.$$

For every S, the S-orbits form a partition of all sets with atoms. The definition of support can be phrased using S-orbits: a set with atoms is supported by S if and only if it is a union (possibly infinite) of S-orbits. As S grows, the partition of sets with atoms into S-orbits becomes more refined. However, the following fact shows that having a finite number of S-orbits does not depend on the choice of S:

Fact II.1. *For every $S \subseteq T$ finite sets of atoms, every S-orbit is a finite union of T-orbits.*

As a result, it is meaningful to define a set with atoms X to be *orbit-finite* if it is partitioned into finitely many S-orbits by some (or, equivalently, every) S that supports X. The sets $\mathbb{A}$, $\mathbb{A}^n$, $\mathbb{A}^{(n)}$ and $\binom{\mathbb{A}}{n}$ are all orbit-finite. Sets like $\mathbb{A}^*$ and $\mathcal{P}_{\text{fs}}\mathbb{A}$ are not orbit-finite.

Turing machines. The definition of a Turing machine with atoms is exactly the same as the classical one, but with finite sets replaced by (legal) orbit-finite sets with atoms. Thus the ingredients of a machine are: states Q, distinguished subsets of initial and accepting states, an input alphabet A, a work alphabet $B \supseteq A$, and a (legal) transition relation:

$$\delta \;\subseteq\; Q \times B \times Q \times B \times \{-1,0,1\}$$

(elements of the set $\{-1,0,1\}$ encode directions of the head move[2]). All these ingredients are required to be orbit-finite sets with atoms. An input is a finite word $w \in A^*$ over the input alphabet. Then one defines configurations, transitions between configurations, runs as sequences of configurations, acceptance, language recognized by a machine, etc., exactly as in the classical case. A machine is deterministic if it has one initial state and its transition relation is a partial function.

The model we have defined is an atom version of a one-tape machine. Two- or three-tape machines would not contribute anything new. One could think about machines with tapes indexed by atoms, or a tape where the directions for the head movement are indexed by atoms. We do not study such models here.

Observe that we do not stipulate rejecting states and therefore do not assume machines to be total. Thus we focus on semi-decidability, not on decidability, in this paper.

Example II.1. 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. (This is the complement of the language from Example I.1, with the additional condition on $\underline{5}$ thrown in to illustrate non-equivariant machines.) The idea is that the machine iterates the following procedure until the tape is empty: if the first letter on the tape is different than $\underline{5}$, replace it by a blank and load it into the state, scan the word to check that the just-erased letter does not appear again, and go back to the beginning of the tape. Formally speaking, the machine is defined as follows.

– The work alphabet is the input alphabet plus the blank symbol (a designated symbol with empty support, used for demarcating the input word). The work alphabet has two $\emptyset$-

[2]Integers can be defined as sets without atoms, so they are also legal sets with atoms. It is in this sense that we use $-1, 0, 1$ in the definition of the transition relation. In particular, 1 is the von Neumann number $\{\emptyset\}$, and it should not be confused with the atom $\underline{1}$.

orbits: one orbit for the atoms, and one singleton orbit for the blank symbol.

– There are three states $init, return$ and $accept$ with empty support, and a set of states $\mathbb{A} - \{\underline{5}\}$. A state a from this set is denoted $scan(a)$. Altogether there are four orbits for the states: singleton $\emptyset$-orbits for $init, return$ and $accept$, and one $\{\underline{5}\}$-orbit for the $scan$ states. One can think of a state $scan(a)$ as consisting of a control component, namely the word $scan$, and a register storing the atom a.

– The state $init$ is initial and the state $accept$ is accepting.

– The transition relation is actually a partial function, and therefore the machine is deterministic. The following set of transitions (which form a single $\{\underline{5}\}$-orbit) corresponds to loading the first letter into the state, erasing it, and moving the head to the right:

$$(init, a) \to (scan(a), blank, 1) \qquad \text{for } a \in \mathbb{A} - \{\underline{5}\}$$

The following set of transitions makes the head move to the end of the tape as long as the atom seen in the first letter does not reappear:

$$(scan(a), b) \to (scan(a), b, 1) \qquad \text{for } a \neq b \in \mathbb{A} - \{\underline{5}\}.$$

The set above also has one $\{\underline{5}\}$-orbit, since every pair of distinct atoms that are both different than $\underline{5}$ can be mapped to any other such pair by a $\{\underline{5}\}$-automorphism of the atoms. The following set of transitions (two orbits) makes the head return to the beginning of the tape:

$$\begin{aligned} (scan(a), blank) &\to (return, blank, -1) \\ (return, a) &\to (return, a, -1) \end{aligned} \qquad \text{for } a \in \mathbb{A}.$$

Note that when $a = \underline{5}$, then the transitions above are never used. The following transition (one transition) makes the machine repeat the procedure

$$(return, blank) \to (init, blank, 1),$$

and the following transition accepts once the tape has been emptied (or if the input was empty to begin with)

$$(init, blank) \to (accept, blank, 0). \qquad \square$$

A Turing machine is a set with atoms (recall that tuples are encoded as sets), therefore it makes sense to talk about the support of a machine. For instance, the machine in the example above is supported by $\{\underline{5}\}$. In general, if a machine is supported by a set of atoms S, then its language is also supported by S. The reason is that the function $M \mapsto L(M)$ which maps a Turing machine to its language is equivariant (its definition does not refer to any specific atoms) so $L(M)$ is supported by the least support of M.

Example II.2. We explain the nondeterministic machine sketched in Example I.2 in some more detail. It is actually a nondeterministic automaton, i.e., it does not write to the tape and always moves the head right. Its state space is

$$\{0\} \ \cup \ \{\triangle(a, b), \triangledown(a, b) : a, b \in \mathbb{A} \text{ distinct}\}$$

so it has three orbits: one singleton orbit, and two orbits isomorphic to $\mathbb{A}^{(2)}$. For better illustration we write ${\triangle}^{a}_{b}$ and ${\triangledown}^{a}_{b}$ instead of $\triangle(a, b)$ and $\triangledown(a, b)$, respectively. In the initial state 0, the automaton inspects the leftmost input letter and nondeterministically chooses a next state according to the set of transitions:

$$\left(0, \begin{smallmatrix} a & & b \\ & \circlearrowright & \\ & c & \end{smallmatrix}\right) \to \left({\triangledown}^{a}_{b}, \begin{smallmatrix} a & & b \\ & \circlearrowright & \\ & c & \end{smallmatrix}, 1\right) \qquad \text{for } a, b, c \in \mathbb{A} \text{ distinct}$$

(recall (2) to see that this defines three outgoing transitions for any input letter).

The machine continues deterministically according to transitions:

$$\left({\triangledown}^{a}_{b}, \begin{smallmatrix} a & & c \\ & \circlearrowright & \\ & b & \end{smallmatrix}\right) \to \left({\triangle}^{a}_{c}, \begin{smallmatrix} a & & c \\ & \circlearrowright & \\ & b & \end{smallmatrix}, 1\right) \qquad \text{for } a, b, c \in \mathbb{A} \text{ distinct}$$

$$\left({\triangle}^{a}_{b}, \begin{smallmatrix} a & & c \\ & \circlearrowright & \\ & b & \end{smallmatrix}\right) \to \left({\triangledown}^{c}_{b}, \begin{smallmatrix} a & & c \\ & \circlearrowright & \\ & b & \end{smallmatrix}, 1\right) \qquad \text{for } a, b, c \in \mathbb{A} \text{ distinct}$$

This defines a partial mapping, as it requires the state and the next letter share at least two atoms in a consistent way.

Each of the three kinds of transitions above forms an equivariant one-orbit set. $\square$

Complexity classes. A language over an orbit-finite alphabet is called *deterministically semi-decidable* if there is a deterministic Turing machine with atoms that recognises it (i.e. accepts the words in the language and does not accept the other words). Likewise, we can talk of a *nondeterministically semi-decidable* language. (We will prove that these notions are not equivalent.)

Even in the presence of atoms, the number of letters in an word and the number of computation steps are natural numbers (without atoms). Therefore it makes sense to talk of time and space resources. This leads to definitions of the classes P and NP with atoms, or other complexity classes, such as PSpace.

When the input alphabet does not contain atoms, say the input alphabet is $\{0, 1\}$, using atoms is not beneficial to the machine. More precisely, when L is a language over an alphabet without atoms, then L is recognised by a deterministic Turing machine with atoms if and only if it is recognised by a deterministic Turing machine without atoms. (A deterministic Turing machine with empty support, given an input word without atoms, cannot produce any atoms during its run, as it transition function has empty support. Similarly, a Turing machine with support S can only produce a bounded number of atoms.) Therefore, over alphabets without atoms, there is only one notion of deterministic semi-decidable language. The same holds for nondeterministic semi-decidable, P and NP.

Prior work. Sets with atoms appear in the literature under various other names: Fraenkel-Mostowski models [2], nominal sets [11], sets with urelements [1], permutation models [12].

Sets with atoms were introduced in the context of set theory by Fraenkel in 1922, and further developed by Mostowski, which is why they are sometimes called Fraenkel-Mostowski sets. They were rediscovered for the computer science community by Gabbay and Pitts [11]. It turns out that atoms are a good way of describing variable names in programs or logical formulas, and the automorphisms of atoms are a good way of describing renaming of variables. Since then, sets with atoms, under the name of nominal sets, have become a lively topic in semantics, see e.g. [14], [13]. Recently, sets with atoms have been investigated from the point of view of automata theory [4], [6], [7] and computation theory [3], [8]. The present paper is a continuation of the latter line of research.

III. Determinism is weaker than nondeterminism

In this section we show that, with atoms, the deterministic and nondeterministic models are not equivalent. What is more, already nondeterministic polynomial time is not included in deterministic semi-decidable languages. This illustrates the weakness of the deterministic model.

Theorem III.1. *In sets with equality atoms, there is a language that is decidable in nondeterministic polynomial time, but not deterministically semi-decidable.*

A consequence of the theorem is that, with atoms, P is not equal to NP. It is not our intention to play up the significance of this result. In a sense, the theorem is too strong for its own good: it shows that computation with atoms is so different from computation without atoms, that results on the power of nondeterminism in the presence of atoms are unlikely to shed new light on the power of nondeterminism without atoms.

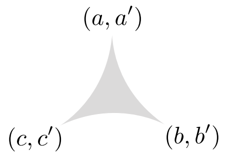
The rest of Section III is devoted to proving Theorem III.1. In Section III-A, we define a language L that witnesses the difference between NP and deterministic decidability; we also show that L is in NP. Then we prove that L is not deterministically semi-decidable. The proof is in two steps. In Section III-B, we define orbit-finite algebras and show that every deterministic Turing machine can be simulated by an orbit-finite algebra. In Section III-C we show that no orbit-finite algebra can recognise L.

A. The language

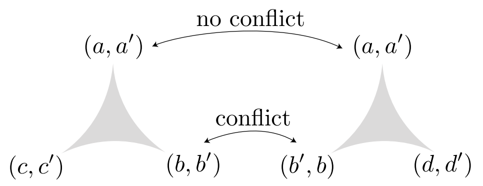
The input alphabet. We begin by defining the input alphabet. We use the name *triangle* for an unordered triple of ordered pairs of atoms, where all six atoms are distinct. In other words, a triangle is a set

$$\{(a,a'),(b,b'),(c,c')\} \qquad \text{where } a,a',b,b',c,c' \text{ are distinct.}$$

We define the *side sets* of the triangle to be the unordered pairs $\{a,a'\}$, $\{b,b'\}$ and $\{c,c'\}$. We denote triangles by τ. The set of all triangles is a one-orbit set. We visualise a triangle as an unordered hyperedge that connects orientations of its side sets:



Suppose that we have several triangles. A *conflict* is a side set which appears in two triangles in different orientations. For instance, the following two triangles have one conflict:



Triangles $\tau_1,\dots,\tau_n$ are called *nonconflicting* if they have no conflicts. Consider a triangle which has a side set $\{a,a'\}$. The *swap on* $\{a,a'\}$ changes the orientation of the side, i.e. changes (a,a') to (a',a) and vice versa. Note that doing a swap (on any side set) does not change the set of side sets. Swaps are a way of resolving conflicts. For instance, in the picture above, doing a swap on the side set $\{b,b'\}$ in the left (or right, but not both) triangle will remove the conflict.

We say that two triangles are *$\approx$-equivalent* if one can go from one to the other by an even number of swaps (i.e. swapping zero or two side sets). We use the name *$\approx$-triangle* for a $\approx$-equivalence class of triangles; each such a class contains exactly four triangles. Doing a single swap changes the $\approx$-class of a triangle, doing another swap comes back to the original class. Therefore, when the side sets are fixed, there are exactly two $\approx$-triangles with these side sets. These two $\approx$-triangles are said to be *dual*, and changing a $\approx$-triangle to its dual is called *flipping*. Note, however, that the set of all $\approx$-triangles is a one-orbit set.

The language. We now define a language that witnesses Theorem III.1. The input alphabet is $\approx$-triangles. The language is the projection, by taking $\approx$-equivalence classes, of the nonconflicting sequences of triangles.

$$L = \{[\tau_1]_\approx \cdots [\tau_n]_\approx : \tau_1,\dots,\tau_n \text{ are nonconflicting triangles}\}$$

Observe that membership in L does not depend on the order or repetition of letters, and therefore it makes sense to talk about a set of $\approx$-triangles belonging to L.

We will show that the language L is a witness for Theorem III.1: it is in NP but not deterministically decidable. Membership in NP is straightforward: the machine has a work alphabet that contains triangles (and not just $\approx$-triangles), and uses nondeterminism to guess $\tau_1,\dots,\tau_n$. Once the actual triangles are given, the machine deterministically compares every two of them to see if they conflict.

There are just exponentially many possibilities for $\tau_1,\dots,\tau_n$. One could ask why there is no deterministic algorithm recognizing L, by exhaustively enumerating all the possibilities? The reason is essentially the same as in (4) in Example I.2: there is no function that would map a $\approx$-triangle to some triangle that belongs to it. In particular, the set of all pairs of the form $([\tau]_\approx,\tau)$ is a relation, not a function.

Remark: The language L is a variant of the Cai-Fürer-Immerman (CFI) construction [15] from Descriptive Complexity Theory. There, it is used to show that a certain logic $\mathcal{C}^\omega_{\infty\omega}$ cannot express a property of (unordered) graphs which is, however, decidable in polynomial time. That result can also be deduced from Theorem III.3 below. The details will appear in the full version of this paper.

Our inspiration for the language L came from yet another source: it is a generalization of a construction from Model Theory ([9], Example on p. 819).

B. Algebras as a model of local computation

The reason why a deterministic Turing machine cannot recognise the language L is that it has only a local view of the computation: the decision for the next step is taken based on the state of the machine, and one cell of the tape. In particular, the decision depends only on the small set of atoms that is found in the state and one cell; the size of this set is fixed by the machine, and does not depend on the input. Our proof will show that any computation model of this kind will not recognise the language L. To model locally based decisions, we use the notion of algebras and terms (similar to circuits). Terms in an algebra are evaluated in a local fashion: the result of a bigger term depends only on a single operation applied to its subterms. By using terms and algebra, our proof will not need to depend on the technical details of Turing machines such as end-of-tape markers, the position of the head, etc.

An *orbit-finite algebra* consists of:

- an orbit-finite *universe* A,
- a finite set of finitely supported *operations* of finite arity:

$$f_1 : A^{n_1} \to A \qquad , \dots , \qquad f_k : A^{n_k} \to A.$$

We require the set of operations to be finite, although an orbit-finite set of operations would also be natural. In this paper we use algebras only as a technical tool, and we choose a truly finite set of operations for technical convenience.

A term in an algebra is defined as usual: it is a finite tree where internal nodes are operations, and the leaves are variables or constant operations. Given a term t, and a valuation val which maps its variables to the universe of the algebra, we write $t[\mathrm{val}] \in A$ for the value of the term under the valuation.

If there is some implicit order $x_1, \dots, x_n$ on the variables of a term, then we can also evaluate the term in a word $w \in A^n$, by using the valuation that maps the i-th variable to the i-th position. We denote this value by $t[w]$.

Recognising a language. We now define what it means for an algebra to recognise a language $L \subseteq A^*$. To input a word from A^*, we require the universe of the algebra to contain the input alphabet, but it can also contain some other elements, which can be seen as a work alphabet. Finally, we require the universe to contain the Boolean values *true* and *false*, so that it can say when a word belongs to the language. We say that such an algebra (non-uniformly) recognises L if for every input length n there is a term t_n with n variables such that

$$t_n[w] = \begin{cases} \textit{true} & \text{if } w \in L \\ \textit{false} & \text{if } w \notin L \end{cases} \qquad \text{for every } w \in A^n.$$

Theorem III.2. *For every deterministic Turing machine, there is an algebra that recognises its language.*

Note that the statement does not require the machine to be total: it is allowed to have non-terminating computations on non-accepted words. In other words, the theorem says that (deterministically) semi-decidable languages are recognised by algebras. The proof is basically an unraveling of the definition of a Turing machine, and is presented in Appendix A.

C. Algebras do not recognise L

By Theorem III.2, in order to show that L is not deterministically semi-decidable, it suffices to show the following:

Theorem III.3. *No orbit-finite algebra recognises L.*

The rest of this section is devoted to proving this theorem.

Triangulations and parity. A set of triangles is called a *triangulation* if

- side sets in the triangulation are either disjoint or equal,
- every side set appears in exactly two triangles.

This definition also makes sense for sets of $\approx$-triangles.

For a set of triangles $\mathcal{T}$, we define

$$[\mathcal{T}]_\approx \stackrel{\text{def}}{=} \{[\tau]_\approx : \tau \in \mathcal{T}\}.$$

We say that two triangles are *neighbouring* if they share a side set. A set of triangles is called *connected* if every triangle can be reached from every other via a sequence of neighbouring triangles. The following shows that, for connected triangulations, membership in L is a parity-type property.

Lemma III.4. *Let $\mathcal{T}$ be a set of triangles that is a connected triangulation. Then $[\mathcal{T}]_\approx \in L$ iff $\mathcal{T}$ has an even number of conflicts.*

The idea is that if two conflicts are connected via a path of triangles, then appropriately performing two swaps in each triangle along the path gives a set of triangles $\mathcal{T}'$, two conflicts fewer than $\mathcal{T}$, such that $[\mathcal{T}']_\approx = [\mathcal{T}]_\approx$.

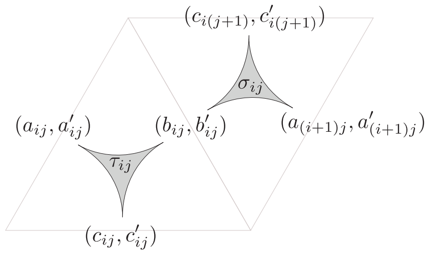
Torus. To construct an input that will confuse an algebra, we will place triangles in a torus-like arrangement. Let $n \in \mathbb{N}$. An n-torus is a set of $2n^2$ $\approx$-triangles defined as follows. Let us begin with $6n^2$ distinct atoms

$$a_{ij}, a'_{ij} \qquad b_{ij}, b'_{ij} \qquad c_{ij}, c'_{ij} \qquad \text{for } i, j \in \{0, \dots, n-1\}.$$

Define a *proper n-torus* to be the following set of $\approx$-triangles:

$$\mathcal{T}_n = \{[\tau_{ij}]_\approx, [\sigma_{ij}]_\approx : i, j \in \{0, \dots, n-1\}\},$$

where the triangles τ_{ij} and σ_{ij} are as follows:



We adopt the convention that all indices are counted modulo n, so that e.g. $a_{in} = a_{i0}$. This means that the neighbourhood graph of a proper n-torus, illustrated in Figure 1, indeed resembles a torus: the triangles on the left are neighbours of the triangles on the right, and likewise for the top and bottom.

An *n-torus* is obtained from a proper n-torus by flipping any of its $\approx$-triangles in any way.

Toruses are difficult for algebras. We now complete the proof of Theorem III.3. The key step is the following lemma.

Fix an algebra $\mathcal{A}$. Let t be a term of $\mathcal{A}$ and $\mathcal{T}$ be an n-torus. For a valuation $\mathrm{val}\colon \textit{variables}(t) \to \mathcal{T}$ and a $\approx$-triangle $\tau \in \mathcal{T}$, we say that t and val *ignore* τ, if

$$t[\mathrm{val}] = t[\mathrm{val}_\tau],$$

where val_τ is defined like val, but with the value of τ flipped.

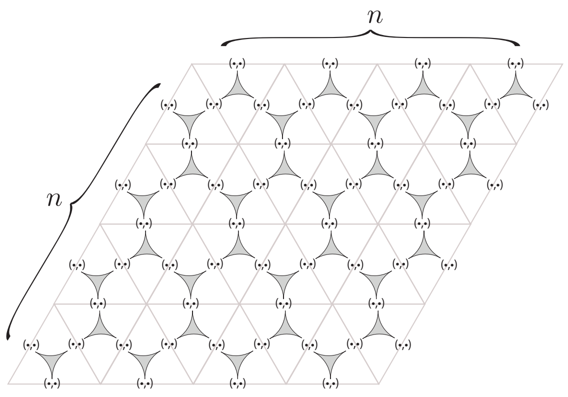


Fig. 1. An n-torus.

Lemma III.5. *There is some $k \in \mathbb{N}$ (depending only on $\mathcal{A}$) such that for every n-torus $\mathcal{T}$ with a sufficiently large n, for every term t in $\mathcal{A}$ and for every valuation* val *with values in $\mathcal{T}$, t and* val *ignore all but at most k elements of $\mathcal{T}$.*

The lemma implies Theorem III.3. Indeed, suppose that an algebra recognizes L, and consider terms t_n recognizing L over words of length n. Since (by Lemma III.4) flipping a single input letter affects membership in L, for every valuation val and every $\tau \in \mathcal{T}$ we have that t and val do not ignore τ. This holds true for every n, and if $2n^2 > k$, this contradicts Lemma III.5, since an n-torus is built of $2n^2$ $\approx$-triangles.

Proof: Let r be the maximal arity of all operations in $\mathcal{A}$. The proof of Lemma III.5 proceeds by induction on the size of the term t. The base case is when the term is a variable, and a variable ignores all $\tau \in \mathcal{T}$ except for one. For the induction step, fix some val. The topmost operation in t has arity at most r, so by the inductive assumption, there are at most $k \cdot r$ elements $\tau \in \mathcal{T}$ which are not ignored by t and val. We need to show, however, that there are actually only at most k such elements τ. The argument has a geometric flavor, and builds on the following easy observation that it is hard to decompose a torus into small pieces:

Fact III.6. *After removing m triangles in an n-torus, there remains a connected component of at least $2n^2 - m^2$ triangles.*

Define $m = 2(k_1 + k_2)$, where k_1 is the size of the least support of all (finitely many) operations in the algebra $\mathcal{A}$ and k_2 is the maximal size of a least support of an element of the universe of $\mathcal{A}$. We now reveal the value of k; we put $k = m^2$.

Let S be a set which supports all operations in the algebra $\mathcal{A}$ and the value $t[\text{val}]$. By induction on the size of the term t, one can show that

$$t[\pi(\text{val})] = t[\text{val}] \qquad \text{for any } S\text{-automorphism } \pi. \tag{5}$$

Without losing generality, S can be chosen so that it has at most $k_1 + k_2$ elements. As every atom appears in at most two $\approx$-triangles, there are at most m elements in $\mathcal{T}$ whose least support intersects S.

Assume now that n is sufficiently large; specifically, we need that $2n^2 > k \cdot r + k$. By Fact III.6, there is a connected subset $C \subseteq \mathcal{T}$ such all elements of C have least supports disjoint with S, and the size of C is at least $2n^2 - k$, so it is bigger than $k \cdot r$. By the inductive assumption, some $\tau \in C$ is ignored by t and val. For the proof of Lemma III.5 it is enough to prove that *every* $\tau \in C$ is ignored by t and val; indeed, there are at most k elements outside of C in the torus. To this end, since C is connected, it is now enough to show:

Lemma III.7. *If some $\tau \in C$ is ignored by t and* val*, then every neighbor $\tau' \in C$ of τ also is.*

To prove this, note that applying the atom automorphism π that swaps the atoms in the side set shared by τ and τ', has exactly the same effect on the torus $\mathcal{T}$ as flipping both τ and τ'. (For this we use the assumption that these atoms do not appear elsewhere in $\mathcal{T}$.) In consequence, flipping τ' has the same effect as applying π and then flipping τ. As the side set is disjoint from S, π is an S-automorphism so, by (5), it does not change the value of $t[\text{val}]$. As a result, flipping τ' has the same effect on $t[\text{val}]$ as flipping τ. ■

IV. The Church-Turing thesis

Theorem III.1 shows that deterministic and nondeterministic Turing machines lead to different notions of decidability. Does this mean that there is no Church-Turing thesis for atoms, i.e. no robust notion of decidability? In this section we argue that there *is* one, with many equivalent formulations; it is just that deterministic Turing machines are not one of them.

A. Representations

A robust notion of computation with atoms not only should have several equivalent definitions, but it should also have a connection to the standard notion of computation without atoms. To make this connection, we represent objects with atoms by using data structures without atoms, which can be written down as bit strings. Atoms themselves can be represented as natural numbers. Using the representation for the atoms, finite permutations of atoms (i.e., those that are the identity on almost all atoms) can be also represented, say as lists of pairs of the form $a_1 \to b_1, \ldots, a_n \to b_n$.

Suppose that X is a set with atoms. A *representation function* for X is an injective function

$$encode : X \to 2^*,$$

which maps an element of X to its (unique) representation so that there is an algorithm solving the following problem:

- *Input.* A finite permutation of atoms π, and an element $x \in X$, both given by representations.
- *Output.* The representation of $\pi(x)$, or an error if $\pi(x) \notin X$.

In [4] it is shown that every orbit-finite set has a representation function. Note that a representation function cannot be finitely supported. If it were supported by S, then it would assign the same bit strings to all arguments in the same S-orbit of X.

B. The Church-Turing thesis

Suppose that A is an orbit-finite alphabet with atoms, and $encode$ is a representation function. A string $a_1 \cdots a_n \in A^*$

is represented as a list of representations of the individual letters. For $L \subseteq A^*$, we write $encode(L)$ to denote the set of all encodings of words in L. Since $encode(L)$ is a set of bit strings, it makes sense to recognise it using a standard Turing machine without atoms.

Theorem IV.1. *Let A be an orbit-finite alphabet, and let $encode$ be a representation function. For a finitely supported language $L \subseteq A^*$, the following conditions are equivalent:*

(1) *$encode(L)$ is recognised by a nondeterministic Turing machine without atoms;*
(2) *L is recognised by a nondeterministic Turing machine with atoms.*

Since (2) does not depend on the representation function, a corollary of this is that the choice of representation function $encode$ does not affect the notion of computability. Observe also that in (1) we could have used any other Turing-complete mechanism without atoms, such as deterministic Turing machines. The proof of the theorem is Appendix B.

Programming languages. As more evidence for the atom version of the Church-Turing thesis, nondeterministic Turing machines with atoms have the same power as two programming languages designed for sets with atoms: a functional language from [3] called $N\lambda$, and an imperative language from [8] called *while programs with atoms*. Both languages can process objects that are richer than simply strings over an orbit-finite alphabet, e.g. they can transform orbit-finite sets into other orbit-finite sets. Even if it is not their principal design goal, both Nλ and while programs with atoms can be used as language recognisers: when A is an orbit-finite alphabet, one can use both programming languages to define functions of the form $A^* \to \{true, false\}$. It turns out that, as language recognisers, both programming languages are equivalent to nondeterministic Turing machines.

Theorem IV.2. *The conditions in Theorem IV.1 are also equivalent to the following conditions:*

3) *L is recognised by a program of $N\lambda$;*
4) *L is recognised by a while program with atoms.*

V. Eliminating restricted nondeterminism

In this section, we show how a deterministic Turing machine can, to some extent, simulate nondeterministic guessing of atoms. As we know from Theorem III.1, in general, such guessing cannot be eliminated. We show that one can eliminate guessing of atoms which are *fresh*, i.e. which do not belong to the least support of the input word or configuration. This simulation preserves computation time and space. We also show that guessing a bounded number of non-fresh atoms can be eliminated too. In particular, deterministic Turing machines can simulate guessing a bounded number of (fresh or non-fresh) atoms.

A Turing machine with the *fresh oracle* is a Turing machine which, at any moment of the run, may consult the oracle to obtain an atom which is fresh with respect to the current configuration. The acceptance condition is defined existentially: the machine accepts an input word if the oracle can respond in such a way that the resulting run is accepting. The precise definition is in Appendix C – there, the oracle provides a fresh atom by writing it at the current head position; however, any other reasonable definition would give an equivalent model.

It is not difficult to see that acceptance does not depend on the responses of the oracle, as long as they are fresh atoms. This observation underlies the proof of the following result, stating that machines with the fresh oracle can be simulated by usual Turing machines, preserving computation time and space. We say that a machine with the fresh oracle is *deterministic* if, apart from the choices made by the oracle, its transitions are deterministic.

Proposition V.1. *Let M be a deterministic Turing machine with the fresh oracle. Then there exists a deterministic Turing machine M' (without the fresh oracle) over the same input alphabet, such that:*

1) *M and M' accept the same input words;*
2) *M and M' have the same supports;*
3) *The computation time and space used by M' is the same as used by M, up to a constant multiplicative factor.*

The proof relies on the notion of *abstraction sets*, introduced by Gabbay and Pitts [11] in their nominal framework for variable binding in semantics. We relegate the proof to Appendix C. We only sketch the rough idea in the case when M invokes the fresh oracle only once, in the first step of the computation. The general statement can be deduced from this special case by an appropriate nesting of Turing machines. The idea is that M' stores in its state an *abstraction*, which is roughly a set I of implications of the following form:

> If the atom returned by the oracle is a, then the state of the machine M is q.

Similar sets are used to represent each tape symbol. It is important that not all the implications in I need to be true, but the ones that involve fresh atoms do. Abstractions form an orbit-finite set. Moreover, they behave well with respect to applying functions, in particular, the transition function of M.

On the other hand, we define a Turing machine with the *choice oracle*, which, at any moment of the run, may nondeterministically obtain an atom from the least support of the tape symbol under the current head position. Similarly as for the fresh oracle, the machine accepts an input word if the choice oracle can respond in such a way that the resulting run is accepting. Contrary to the case of the fresh oracle, acceptance of the run may depend on the answers of the oracle.

Example V.1. Consider an input alphabet $\binom{\mathbb{A}}{n}$, i.e. sets of n atoms. We show how a Turing machine M with the choice oracle can linearly order the atoms in a single letter. The work alphabet of the machine is sets of at most n atoms. Given a letter X, the machine invokes the choice oracle to choose some atom $a \in X$. It writes this atom in one cell of the tape, and in another cell it writes the set $X - \{a\}$. The procedure is then repeated with $X - \{a\}$ in place of X, until X becomes empty.

The choice oracle can be used to order the atoms in the least support of a letter b from an arbitrary orbit-finite alphabet: simply apply the above construction to $X = \text{supp}(b)$, where supp(b) is the least support of b; the mapping $b \mapsto \text{supp}(b)$ is legal, so it can be carried out by a deterministic machine.

Theorem III.1 implies that the choice oracle cannot be eliminated in general, but according to the following result, a bounded number of calls can be eliminated.

Proposition V.2. *Let n be a number and let M be a deterministic Turing machine with the choice oracle, such that in every run of M, M invokes the choice oracle at most n times. Then there exists a deterministic Turing machine M' over the same input alphabet, such that:*

1) *M and M' accept the same input words;*
2) *M and M' have the same supports;*
3) *The computation time and space used by M' is the same as used by M, up to a constant multiplicative factor.*

The proof is in Appendix C; the rough idea is similar as in the proof of Proposition V.1.

A family of objects $\{x_i\}_{i \in I}$ is modelled as a legal function $i \mapsto x_i$. As a corollary from Proposition V.2, deterministic Turing machines are closed under orbit-finite unions:

Corollary V.3. *Let I be an orbit-finite set, and let $\{M_i\}_{i \in I}$ be a family of deterministic Turing machines over a common input alphabet. Then the language $\bigcup_{i \in I} L(M_i)$ is recognizable by a deterministic Turing machine.*

Proof: For every single-orbit set I there is some $n \in \mathbb{N}$ and a surjective legal partial function $f : \mathbb{A}^n \to I$. Therefore, it is is enough to consider the case when $I \subseteq \mathbb{A}^n$ for some n. Guessing an n-tuple of atoms can be simulated by invoking the fresh oracle or the choice oracle at most n times. ■

VI. A DICHOTOMY FOR INPUT ALPHABETS

To separate deterministic and nondeterministic computation in Theorem III.1, a rather complex input alphabet was needed. For simpler alphabets, such as that of atoms, nondeterministic Turing machines do determinize. As it turns out, there is a dichotomy on input alphabets:

Theorem VI.1. *Every input alphabet A is either:*

1) **Nonstandard.** *There is a language over A that is in* NP *but not deterministically semi-decidable, or*
2) **Standard.** *For languages over A,*
   a) *deterministic semi-decidable equals nondeterministic semi-decidable.*
   b) *the answer to* P=NP *is the same as classically.*

The proof (to be found in Appendix D) also shows that over a standard alphabet many complexity questions have the same answer as classically, including any equality concerning the classes P, NP, PSPACE and EXPTIME. Conversely, over nonstandard alphabets, any complexity class that allows an unbounded number of nondeterministic guesses (e.g. nondeterministic logarithmic space) is not included in the deterministically semi-decidable languages.

A canonical language. Whether an alphabet is standard or not can be traced to one kind of language. For a finite set of atoms S, define

$$L_{A,S} = \{wv \in A^* : w = \pi(v) \text{ for an } S\text{-automorphism } \pi\}.$$

Note that $w = \pi(v)$ implies that w and v have the same length.

Lemma VI.2. *The language $L_{A,S}$ is in* NP.

Proof sketch: Given input wv, a nondeterministic Turing machine can guess the S-automorphism π, by nondeterministically writing on the tape pairs of the form $(a, \pi(a))$ such that a is in the least support of the word w and $\pi(a)$ is in the least support of the word v. Once π is written on the tape, the condition $w = \pi(v)$ can be verified in deterministic polynomial time. ■

The following theorem shows that the language $L_{A,S}$ contains all the difficulties for deterministic computation: if it can be recognised by a deterministic Turing machine, then all nondeterministic Turing machines can be determinised[3].

Theorem VI.3. *For every finite set S of atoms and every orbit-finite alphabet A, the following conditions are equivalent:*

1) *the language $L_{A,S}$ is in* P*;*
2) *the language $L_{A,S}$ is deterministically semi-decidable;*
3) *the language $L_{A,S}$ is recognized by some orbit-finite algebra;*

Furthermore, for a fixed A, if one (or all) conditions above hold for some S, then they hold for every S. Finally, the alphabet A is standard if the conditions above hold, and nonstandard otherwise.

We conclude with some examples of standard and nonstandard alphabets.

Example VI.1. To see that the alphabet $\mathbb{A}$ is standard, by Theorem VI.3, it is enough to show that the language $L_{\mathbb{A},\emptyset}$ is in P. A word $a_1 \cdots a_n b_1 \cdots b_n$ belongs to this language if and only if

$$a_i = a_j \quad \text{iff} \quad b_i = b_j \qquad \text{for every } i, j \in \{1, \ldots, n\}.$$

This is easily checked in deterministic polynomial time. The same argument works for alphabets of the form $\mathbb{A}^n$.

Another example of a standard alphabet is $\binom{\mathbb{A}}{2}$, i.e. two-element sets of atoms. To test if two given words of equal length are in the same $\emptyset$-orbit, it is sufficient to check that the intersection of every three letters has the same cardinality in both words. (This implies that the intersection of any subset of letters has the same cardinality in both words.) This decision procedure generalizes easily to sets of atoms of cardinality greater than two.

On the other hand, the alphabet used in Section III-A is nonstandard, by Theorem III.1.

VII. ATOMS WITH STRUCTURE

So far, we have assumed that the only structure on the atoms is equality. To study atoms with some additional structure, following [4], we can model the atoms as a relational structure (as in model theory). We can then define (legal) sets with atoms and orbit-finite sets in the same way as before, with the notion of atom automorphism now inherited from the relational structure. The atoms with equality that have been discussed so far correspond to the relational structure $(\mathbb{N}, =)$,

[3]Note, however, that we do not claim NP-completeness of $L_{A,S}$.

or any other countable set with equality. Sets without atoms correspond to the empty relational structure.

In this section we show two other examples of relational structures for the atoms, and see what happens to determinisation of Turing machines with those atoms.

A. Total order atoms

Consider the atom structure where the universe is the rational numbers with order: $(\mathbb{Q}, \leq)$. We use the name *sets with total order atoms* for sets with atoms built on this relational structure. In sets with total order atoms, Turing machines behave the same way as without atoms.

Theorem VII.1. *Consider the total order atoms. For every input alphabet:*

- *deterministic semi-decidable is equal to nondeterministic semi-decidable; and*
- *the answer to* P=NP *is the same as classically.*

The intuition is that having a linear order on the atoms, the choice oracle can be easily eliminated, simply by choosing the minimal element of the least support of a symbol – in presence of a linear order, this is an equivariant function.

B. Bit vector atoms

We now present another example of atoms, where deterministic polynomial time is weaker than nondeterministic polynomial time (as in the equality atoms), but deterministic decidable is equal to nondeterministic decidable (unlike in the equality atoms). The example also shows how atoms can be used to model limited access to the input data (in this case, the data is a vector space).

We use the name *bit vector* for an infinite sequence of zeros and ones which has finitely many ones. By ignoring the trailing zeros, a bit vector can be represented as a finite sequence such as 00101001. Bit vectors can be added modulo two, and multiplied by 0 or 1. Equipped with this structure, we get a vector space over the two element field. The dimension of this vector space is countably infinite, an example basis consists of bit vectors which have a 1 on exactly one coordinate:

$$1, 01, 001, 0001, \dots$$

The bit vectors can be seen as a relational structure, with a ternary predicate $x + y = z$ for addition modulo 2 and a unary predicate $0(x)$ for distingushing the zero vector. Using the automorphisms of this structure (such an automorphism is required to preserve addition; it is uniquely defined by a mapping from one basis to another), we can define *sets with bit-vector atoms.*

Theorem VII.2. *Over sets with bit-vector atoms, for the input alphabet of (bit-vector) atoms,*

- *deterministic semi-decidable is equal to nondeterministic semi-decidable; and*
- P $\neq$ NP.

The problem that separates P from NP is testing linear dependence of vectors, i.e. the following language:

$$D \stackrel{\text{def}}{=} \{a_1 \cdots a_n \in \mathbb{A} : a_1, \dots, a_n \text{ are linearly dependent}\}.$$

It is easy to see that the language D is recognised by a nondeterministic polynomial time Turing machine. The machine uses nondeterminism to guess a linear combination that witnesses dependence:

$$c_1 \cdot a_1 + \cdots + c_n \cdot a_n = 0$$

The coefficients $c_1, \dots, c_n \in \{0, 1\}$ are guessed as the machine moves through the input tape, after i steps the machine stores in its state the partial sum of a subset of the first i vectors. (To do this, the state space Q of the machine needs to be built using atoms.)

In the appendix, we show that checking all linear combination is inevitable: every deterministic Turing machine will need an exponential number of computation steps to recognise the language D.

Conclusions

We mostly study computation over atoms whose structure allows only for equality tests. It turns out that deterministic Turing machines are strictly less expressive than nondeterministic ones, the reason being that the state of the machine can only store a bounded number of atoms. A natural question is to find a deterministic model for computation with atoms which is expressively equivalent to the nondeterministic machines. Two such models are mentioned in this paper: the functional programming language Nλ, and while programs with atoms. Is there such a model in the spirit of Turing machines?

We also initiate the study of computation over atoms whose structure is richer than just equality tests. It would be interesting to see for what structures one can prove lower bounds similar as in Theorem VII.2, or that P$\neq$NP.

References

[1] Jon Barwise. *Admissible sets and structures*. Springer-Verlag, Berlin, 1975. An approach to definability theory, Perspectives in Mathematical Logic.

[2] Jon Barwise, editor. *Handbook of Mathematical Logic*. Number 90 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1977.

[3] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin, and Sławomir Lasota. Towards nominal computation. In *POPL*, pages 401–412, 2012.

[4] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata with group actions. In *LICS*, pages 355–364, 2011.

[5] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets. Submitted, 2012.

[6] Mikołaj Bojańczyk and Sławomir Lasota. A machine-independent characterization of timed languages. In *ICALP (2)*, pages 92–103, 2012.

[7] Mikołaj Bojańczyk and Thomas Place. Toward model theory with data values. In *ICALP (2)*, pages 116–127, 2012.

[8] Mikołaj Bojańczyk and Szymon Toruńczyk. Imperative programming in sets with atoms. In *FSTTCS*, pages 4–15, 2012.

[9] G. Cherlin and A.H. Lachlan. Stable finitely homogeneous structures. *Trans. AMS*, 296(2):815–850, 1985.

[10] M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[11] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[12] Thomas Jech. *The Axiom of Choice*. North-Holland, 1973.

[13] Andrzej S. Murawski and Nikos Tzevelekos. Algorithmic nominal game semantics. In *ESOP*, pages 419–438, 2011.

[14] David Turner and Glynn Winskel. Nominal domain theory for concurrency. In *CSL*, pages 546–560, 2009.

[15] 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.

Appendix A
Proofs for Section III

A. *Proof of Theorem III.2*

Let M be a deterministic Turing machine with atoms. Define the universe to be the disjoint union of the work alphabet B of the machine, the state space Q of the machine, the Booleans, and an undefined value:

$$B \cup Q \cup \{\mathit{true}, \mathit{false}\} \cup \{\text{undefined}\}.$$

A partial function on the universe is encoded by using the undefined value. The machine is encoded in the algebra in the most straightforward way:

- A constant for the initial state.
- A function for the accepting states:

  $$accept : Q \to \{\mathit{true}, \mathit{false}\}.$$

  When given an argument outside Q, the result is undefined. The same encoding for partiality is done for the functions below.
- The transition function is represented by the following functions. The new state is given by a function

  $$state : Q \times B \to Q.$$

  The symbol written on the tape is given by

  $$write : Q \times B \to B.$$

  The direction of the head is given by two functions:

  $$moveleft : Q \times B \to \{\mathit{true}, \mathit{false}\}$$
  $$moveright : Q \times B \to \{\mathit{true}, \mathit{false}\}$$

  If both functions above say *false*, then the head does not move.

There are constants for true and false. Finally, to simulate control flow, the algebra contains an if-then-else construction:

$$\text{ifthenelse} : \{\mathit{true}, \mathit{false}\} \times U \times U \to U.$$

This construction yields, among other things, the standard Boolean operations. This completes the definition of the algebra.

We now show that the algebra recognises the language of the Turing machine. We will show that for every input length n and $i, j \in \mathbb{N}$, there are the following terms:

- There is a term $state_{n,i}$, with values in Q, which evaluated in a word w returns the state of the machine after i steps of computation on w.
- There is a term $tape_{n,i,j}$, with values in B, which evaluated in a word w returns the contents of the j-th cell of the tape after i steps of computation on w.
- There is a term $head_{n,i}$, with values in the Booleans, which evaluated in a word w returns true if the head of the machine is over cell j after i steps of computation on w.

The terms are defined by induction on the time i of computation, by simply formalising the definition of a Turing machine's run.

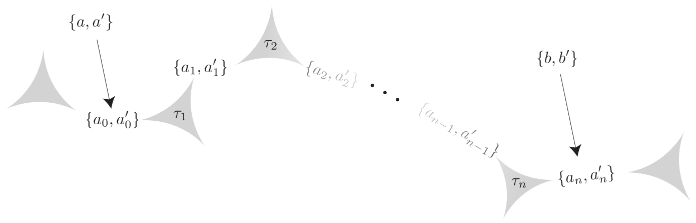
To complete the proof, we observe that for each input length n, there is a maximal running time i_n that is used by the machine over accepted inputs of length n. (The function $n \mapsto i_n$ might not be decidable, but we are using a non-uniform model, so this is not a problem.) Therefore, the term for input length n uses Boolean operations to test if for some $i \in \{1, \dots, i_n\}$, the machine reaches an accepting state.

B. *Proof of Lemma III.4*

Lemma III.4 says that if $\mathcal{T}$ is a set of triangles that is a connected triangulation, then $[\mathcal{T}]_\approx \in L$ iff $\mathcal{T}$ has an even number of conflicts.

We begin with the left to right implication, which does not use the assumption on connectedness. Since every side set appears in exactly two triangles, doing a swap on a single triangle in $\mathcal{T}$ changes the number of conflicts by one: either it adds one or removes one. Changing a single triangle to a $\approx$-equivalent does an even number of swaps, and therefore preserves the parity of the number of conflicts. If $[\mathcal{T}]_\approx \in L$, then there is some $\mathcal{S}$ which has zero (an even number) of conflicts, and $[\mathcal{T}]_\approx = [\mathcal{S}]_\approx$. It follows that $\mathcal{T}$ has an even number of conflicts.

We now do the right to left implication, which uses the assumption on connectedness. Suppose that $\mathcal{T}$ is a connected triangulation with an even (but nonzero) number of conflicts. We will find a set of triangles $\mathcal{S}$ that has two inconsistencies less, and satisfies $[\mathcal{T}]_\approx = [\mathcal{S}]_\approx$. By iterating this procedure, we eventually reach a set in L. Let then $\{a, a'\}$ and $\{b, b'\}$ be two conflicting sides of two triangles in $\mathcal{T}$. By the assumption on connectedness, there are neighbouring triangles $\tau_1, \dots, \tau_n \in \mathcal{T}$ that connect $\{a, a'\}$ with $\{b, b'\}$ as in the following picture:



For each $i \in \{0, \dots, n\}$, let σ_i be the triangle obtained from τ_i by swapping the side sets $\{a_{i-1}, a'_{i-1}\}$ and $\{a_i, a'_i\}$. Since two side sets are swapped it follows that $\tau_i \approx \sigma_i$. Let $\mathcal{S}$ be the set of triangles obtained from $\mathcal{T}$ by replacing $\tau_1, \dots, \tau_n$ with $\sigma_1, \dots, \sigma_n$. Since two or zero swaps are done on each letter, we have $[\mathcal{S}]_\approx = [\mathcal{T}]_\approx$. We claim that $\mathcal{S}$ has two less conflicts than $\mathcal{T}$. Indeed, the side $\{a, a'\}$ is no longer an inconsistency, since we swapped the atoms a and a' on one of the triangles incident to it, namely the triangle τ_0, but not on the other triangle. The same argument holds for the side $\{b, b'\}$. On the other hand, for each $i \in \{1, \dots, n-1\}$, the side $\{a_i, a'_i\}$ is an inconsistency in $\mathcal{T}$ if and only if it was an inconsistency in $\mathcal{S}$, because we swapped the atoms a_i and a'_i on both triangles that contain the edge set.

Appendix B
Appendix to Section IV

A. *Proof of Theorem IV.1*

The theorem says that the following conditions are equivalent for a finitely supported language L over an orbit-finite alphabet A, under a representation function $encode$:

1) $encode(L)$ is recognised by a nondeterministic Turing machine without atoms;
2) L is recognised by a nondeterministic Turing machine with atoms.

Other operations on representation functions. Before proving Theorem IV.1, we show some simple results on representation functions.

Lemma B.1. *Let $encode : X \to 2^*$ be a representation function. The following problem is semi-decidable:*

- *Input. Elements $x, y \in X$, and a finite set S of atoms, given by representations.*
- *Output. The representation of a partial S-permutation π of atoms such that $\pi(x) = y$, or an error if there is none.*

Proof: By enumerating all partial S-permutations of atoms. ■

Using the same argument as in the above lemma, we can show:

Lemma B.2. *Let $encode : X \to 2^*$ be a representation function. For every finitely supported relation $R \subseteq X^n$, the set of representations of tuples in R is semi-decidable.*

Proof: A finitely supported relation on an orbit-finite set is orbit-finite, as a set of tuples. Therefore, R is orbit-finite. Let S be a support of R, and let

$$\bar{x}_1, \dots, \bar{x}_k \in X^n$$

be tuples such that every other tuple of R is equal to one of them, up to S-permutations. Since S-permutations can be enumerated, for every $i \in \{1, \dots, n\}$, the representations of the S-orbit of x_i form a semi-decidable set of bit strings. Therefore, the representations of all tuples in R form a semi-decidable set of bit strings. ■

The above "algorithms" are very inefficient, but have the advantage of only using the abstract definition of a representation function. The representation functions that are shown in [4] realise the operations above in polynomial time.

The following lemma shows that representation functions can be easily extended to representation functions for bigger sets.

Lemma B.3. *Let $encode : X \to 2^*$ be a representation function. For every orbit-finite $Y \supseteq X$, there is a representation function $encode' : Y \to 2^*$ such that*

$$encode'(x) = 0 \cdot encode(x) \qquad \textit{for every } x \in X.$$

Proof
The set $Y - X$ is also orbit-finite, and therefore by [4] it has a representation function, call it $encode''$. The representation function for Y is obtained by combining the two:

$$y \in Y \qquad \mapsto \qquad \begin{cases} 0 \cdot encode(y) & \text{if } y \in X \\ 1 \cdot encode''(y) & \text{otherwise.} \end{cases}$$

□

We now proceed to the proof of Theorem IV.1.

Implication from 2) to 1). Fix a nondeterministic Turing machine with atoms M, which recognises the language L. Let B be the work alphabet of the machine, and Q the state space. By Lemma B.3, we can assume without loss of generality that the representation function for the input alphabet extends to a representation function for the disjoint union $B \sqcup Q$, which we also denote by $encode$:

$$encode : B \sqcup Q \to 2^*.$$

We now prove that $encode(L)$ is recognised by a nondeterministic Turing machine without atoms.

Let us denote configurations of M by letters c and d. We write $encode(c)$ for an encoding of a configuration: first we encode the contents of the tape (letter by letter), then we encode the state of the machine, and then we write a number indicating the position of the head. We claim that the encodings of the one-step transition relation, namely

$$\{(encode(c), encode(d)) : M \text{ can go from } c \text{ to } d \text{ in one step}\},$$

is a decidable language. Indeed, we only need to test that the configuration has changed in the vicinity of the head in a way consistent with the transition function. For this, all we need is that the following set of tuples of bit strings is semi-decidable:

$$\{(encode(a), encode(q), encode(b), encode(p), dir) :$$
$$(a, q \to b, p, dir) \text{ is a transition in the machine } M\}.$$

This follows from Lemma B.2.

Implication from 1) to 2). The main idea in this implication is that a nondeterministic Turing machine with atoms can produce an encoding of its input. In a literal sense, this is not true. For instance, it is impossible to write a nondeterministic Turing machine which recognises the language

$$\{w\#encode(w) : w \in \mathbb{A}^*\} \tag{6}$$

The problem is that the language above is not finitely supported, for the same reasons that the $encode$ function is not finitely supported. Because of this issue, we can only hope for a machine that produces and encoding of its input, or some other input in the same orbit. This machine is described in the following lemma.

Lemma B.4. *Let S be a finite set of atoms, and A an S-supported orbit-finite alphabet. The language*

$$\{w\#encode(\pi(w)) : w \in A^* \textit{ and } \pi \textit{ is an } S\textit{-automorphism}\}$$

is recognised by a nondeterministic Turing machine with atoms.

Before we prove the lemma, we show how it yields the implication from 1) to 2) in Theorem IV.1.

Proof of implication 1) to 2) in Theorem IV.1: Let S be a finite set of atoms that supports L, and let M be a Turing machine without atoms that recognises the language $\mathit{encode}(L)$. Below we describe a nondeterministic Turing machine with atoms that recognises L. Given an input $w \in A^*$ the machine uses Lemma B.4 to nondeterministically guess a word v such that $v = \mathit{encode}(\pi(w))$ for some S-automorphism π. Because L is supported by S, we know that

$$w \in L \qquad \text{iff} \qquad \pi(w) \in L.$$

Then, we run the machine M to test if $\pi(w)$ belongs to L. ■

Proof of Lemma B.4: We first prove the lemma for the case when A is the atoms. An input to the machine is a sequence of atoms $a_1, \ldots, a_n$, followed by a separator $\#$, followed by a list of bit strings $b_1, \ldots, b_n$. The machine first tests if the input is actually legal, i.e. the bit strings $b_1, \ldots, b_n$ are encodings of atoms. Then it accepts if:

- The sequences $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ have the same equality type. In other words, for every $i, j \in \{1, \ldots, n\}$, the letters a_i and a_j are equal if and only if the bit strings b_i and b_j encode the same atom.
- For every $i \in \{1, \ldots, n\}$, if $a_i \in S$ then $b_i = \mathit{encode}(a_i)$.

Both conditions above can be easily tested by a Turing machine. (For the second condition it is necessary that the machine is actually S-supported.)

This completes the proof of the lemma for the case when A is the atoms. The same proof works for the case when A consists of tuples of atoms. We now proceed to the case when A is an arbitrary orbit-finite set.

Every orbit-finite set A is a surjective image of non-repeating tuples of atoms. This means that there is some $k \in \mathbb{N}$ and a surjective function

$$\alpha : \mathbb{A}^{(k)} \to A.$$

When A is S-supported, then the function α can also be assumed to be S-supported.

Choose some representation function for $\mathbb{A}^{(k)}$, which by abuse of notation we will also denote by encode. (Strictly speaking, we apply Lemma B.3 and get a common representation function for $\mathbb{A}^{(k)} \cup A$.)

Let $\hat{\alpha} : 2^* \to 2^*$ be the lifting of α to encodings, i.e. the function

$$x \qquad \mapsto \qquad \mathit{encode}(\alpha(\mathit{encode}^{-1}(x))$$

Using the definition of a representation function, one can show that $\hat{\alpha}$ is semi-decidable in the usual sense, on bit strings. We now describe the machine from the statement of the lemma. Suppose that the input word is

$$a_1 \cdots a_n \in A^*$$

The machine first uses nondeterminism to guess letters

$$b_1 \in \alpha^{-1}(a_1) \qquad \cdots \qquad b_n \in \alpha^{-1}(a_n)$$

and stores these letters on the tape. Since we have already proved the lemma for alphabets of the form $\mathbb{A}^{(k)}$, the machine can now guess strings

$$c_1, \ldots, c_n \in 2^*$$

such that for some S-automorphism π, we have

$$c_1 \cdots c_n = \mathit{encode}(\pi(b_1 \cdots b_n)).$$

We can now compute

$$\hat{\alpha}(c_1) \cdots \hat{\alpha}(c_n).$$

To complete the lemma, it suffices to show that

$$\hat{\alpha}(c_1) \cdots \hat{\alpha}(c_n) = \mathit{encode}(\pi(a_1 \cdots a_n)). \tag{7}$$

The proof of (7) is a simple calculation:

$$\hat{\alpha}(c_i) = \mathit{encode}(\alpha(\mathit{encode}^{-1}(c_i))) = \\ \mathit{encode}(\alpha(\mathit{encode}^{-1}(\mathit{encode}(\pi(b_i))))) = \\ \mathit{encode}(\alpha(\pi(b_i)))$$

Because α is S-supported and π is an S-automorphism, the above becomes

$$\mathit{encode}(\pi(\alpha(b_i))) = \mathit{encode}(\pi(a_i)),$$

as required by (7). ■

B. Proof of Theorem IV.2

The theorem says that the conditions from Theorem IV.2 are equivalent to:

3) L is recognised by a program of Nλ;
4) L is recognised by a while program with atoms.

Proof: The implication from 4) to 1) is a special case of Theorem 5 in [8]. For the implications from 2) to 3) and from 3) to 4), we only give a rough sketch, since a more detailed proof would require a mored detailed description of the programming languages involved.

The basic idea behind the implication from 2) to 3) is that Nλ, as a language which processes orbit-finite sets, can simply implement the definition of Turing machine acceptance. Suppose that M is a nondeterministic Turing machine with atoms. Given an input word w, the set of configurations of the machine that can be reached in n steps is an orbit-finite set, call it $\mathit{conf}(w, n)$. In Nλ one can write a program that inputs w and n and outputs $\mathit{conf}(w, n)$. This set can then be searched for an accepting configuration, since Nλ allows exhaustive search of orbit-finite sets.

The basic idea behind the implication from 3) to 4) is the same as above: while programs with atoms, as a language which processes orbit-finite sets, can simply implement the operational semantics of the language Nλ.

■

Appendix C
Appendix to Section V

Before proving the results concerning the oracles, we introduce some preliminary notions, which originate from the work of Gabbay and Pitts [11].

For an atom or set with atoms X, we denote by $\mathrm{supp}(X)$ the least support of X.

A. Freshness

Let X be a set. For $(a, x), (a', x') \in \mathbb{A} \times X$, we write $(a, x) \sim (a', x')$ if there is a permutation π such that:

- $\pi \cdot (a, x) = (a', x')$
- $\pi(b) = b$ for each $b \in \text{supp}(x) - \{a\}$.

The relation $\sim$ is easily seen to be an equivalence relation. We write $[a].x$ for the equivalence class of a pair (a, x); such equivalence classes are called *abstraction classes*. We write $[\mathbb{A}]X$ for the set of equivalence classes of the relation $\sim$. Observe that if X is orbit-finite, then so is $[\mathbb{A}]X$, as the image of the orbit-finite set $\mathbb{A} \times X$. Also, if $X \subseteq Y$, then $[\mathbb{A}]X \subseteq [\mathbb{A}]Y$.

Example C.1. Let $X = \mathbb{A}^3$. Then $(\underline{3}, (\underline{1}, \underline{2}, \underline{3})) \sim (\underline{4}, (\underline{1}, \underline{2}, \underline{4}))$. Both pairs define the equivalence class $[\underline{3}].(\underline{1}, \underline{2}, \underline{3})$.

Remark: The definition of the equivalence relation $\sim$ is borrowed from [11]. There, the equivalence class of (a, x) is denoted simply $a.x$.

The following proposition lists several important properties of $[\mathbb{A}]$. The first two suggest that $[a].x$ is a way of "variation of the parameter" a in x.

Proposition C.1 ([11]).
1) *For any $a \in \mathbb{A}$ and $x \in X$,* $\text{supp}([a].x) = \text{supp}(x) - \{a\}$.
2) *Each equivalence class $\tilde{x} \in [\mathbb{A}]X$ is a subset of $\mathbb{A} \times X$ which is a partial mapping from $\mathbb{A}$ to X with domain $\mathbb{A} - \text{supp}(\tilde{x})$, such that for $a \in \mathbb{A} - \text{supp}(\tilde{x})$,*
$$\tilde{x}(a) = x \quad \textit{if and only if} \quad \tilde{x} = [a].x.$$
3) *There is a bijection $[\mathbb{A}](X \times Y) \simeq [\mathbb{A}]X \times [\mathbb{A}]Y$, which maps $[a].(x, y)$ to $([a].x, [a].y)$.*
4) *If $f \subseteq X \times Y$ is a function from X to Y, then $[\mathbb{A}]f \subseteq [\mathbb{A}](X \times Y) \simeq [\mathbb{A}]X \times [\mathbb{A}]Y$ is a function from $[\mathbb{A}]X$ to $[\mathbb{A}]Y$. This function satisfies*
$$([\mathbb{A}]f)([a].x) = [a].f(x).$$

For proofs, see [11], Section 5.

B. Fresh oracles

In this section, we prove Proposition V.1. First, we give a precise definition. Formally, a Turing machine with the *fresh oracle* is a nondeterministic Turing machine, whose work alphabet B contains $\mathbb{A}$, and state space Q contains two distinguished states q_{fresh} and q_0, both with empty support. Moreover, the transition relation $\delta \subseteq Q \times B \times Q \times B \times \{-1, 0, 1\}$ is of the form $\delta = \delta_0 \cup \delta_{\text{fresh}}$, where δ_0 is a transition relation with no transitions leaving from the state q_{fresh}, and
$$\delta_{\text{fresh}} = \{(q_{\text{fresh}}, b, q_0, a, 0) : \quad b \in B, a \in \mathbb{A}\}.$$
Note that the transitions in δ_{fresh} involve a nondeterministic choice of $a \in \mathbb{A}$; we view this choice as being made by the oracle and not the machine, so we say that that M is *deterministic* if the transition relation δ_0 is deterministic. We say that M *accepts* an input word if there is an accepting run in which each time the fresh oracle is invoked, the atom chosen by the oracle is fresh with respect to the current configuration.

We will first prove Proposition V.1 in the special case, when the machine invokes the fresh oracle only once, in the first step of the computation. Simplifying this case further, we assume that the atom a replied by the oracle is not written on the tape, but instead, the machine moves to a state q_a, which equivariantly depends on a. This special case is handled by the lemma below. The general case can be deduced by an appropriate nesting of deterministic Turing machines.

Lemma C.2. *Let M be a deterministic Turing machine, and let $\{q_a\}_a \in \mathbb{A}$ be an equivariant family of states of M. For $a \in \mathbb{A}$, let M_a denote the machine M, but whose initial state is q_a.*

Then the following language is recognizable by a deterministic Turing machine:
$$\{w : \ L(M_a) \textit{ for some } a \in \mathbb{A} \textit{ which is fresh with respect to } w\}.$$

Proof: We construct a deterministic Turing machine M' which recognizes the language in the above statement. By *fresh atom* we mean an atom which is fresh with respect to the input word for the machine M'.

Let the transition function of the machine M be $\delta \colon Q \times B \to Q \times B \times \{-1, 0, 1\}$, where Q is the state space and B is the work alphabet. The constructed machine M' has state space $[\mathbb{A}]Q$ and work alphabet $[\mathbb{A}]B$. The initial state of M' is $\tilde{q}_0 = [a].q_a$. We view the input alphabet A of M as a subset of $[\mathbb{A}]B$, via the natural embedding which maps a letter x to the letter $[a].x$, where $a \notin \text{supp}(x)$.

We now define the transition function of M'. Recall that according to the notation from Section C-A,
$$[\mathbb{A}]\delta \subseteq [\mathbb{A}](Q \times B \times Q \times B \times \{-1, 0, 1\}).$$
Using the isomorphism from the third item of Proposition C.1, $[\mathbb{A}]\delta$ can be viewed as a subset of the set
$$[\mathbb{A}]Q \times [\mathbb{A}]B \times [\mathbb{A}]Q \times [\mathbb{A}]B \times [\mathbb{A}]\{-1, 0, 1\}.$$
Observe that $[\mathbb{A}]\{-1, 0, 1\}$ is naturally isomorphic to $\{-1, 0, 1\}$, since all elements $-1, 0, 1$ have empty support. According to the last item of Proposition C.1, the relation $[\mathbb{A}]\delta$ is in fact a function from $[\mathbb{A}]Q \times [\mathbb{A}]B$ to $[\mathbb{A}]Q \times [\mathbb{A}]B \times \{-1, 0, 1\}$. Hence indeed, $[\mathbb{A}]\delta$ defines a deterministic Turing machine M'.

Fix an input word w and an atom a which is fresh with respect to w.

Claim C.2.1. *In every step of the computation, the atom a is fresh with respect to the configuration of M'.*

This follows from the fact that the machine M' is deterministic, so the n-th configuration of M is a function of the input word w. In particular, the support of the n-th configuration is contained in the support of the input word, which, in turn, does not contain the atom a. According to the second item of Proposition C.1, we can therefore apply the state in the n-th configuration to the atom a, obtaining a state of the machine M. Similarly, we can apply the each tape symbol to the atom a.

We now describe an invariant which binds the behavior of M' and of M_a, at each moment of the computation.

The following invariant will be satisfied throughout the computation.

> Suppose that n steps of computation have elapsed. If we feed a to the functions defined by the state and the symbols in the tape cells, we recover the n-th computation step of the machine M_a on the original input (the head position will be the same).

Initially, the invariant is granted thanks to choice of the initial state, and the interpretation of the input symbols as elements of $[\mathbb{A}]B$.

We argue that a step of the computation maintains the invariant. Suppose that the current state and tape symbol of M' constitute the pair $([a].q, [a].b)$. By Claim C.2.1, a is not in the least support of this pair. On the other hand, by the invariant, the current state and tape symbol of M_a is (q, b).

From the second and last item of Proposition C.1 it follows that

$$\big(([\mathbb{A}]\delta)([a].q, [a].b)\big)(a) = \delta(q, b),$$

since a is not in the support of $([a].q, [a].b)$. From this it follows that the invariant is maintained in the $(n+1)$-th step of the computation. ■

C. Choice oracle

In this section, we prove Proposition V.2.

To simplify the description, again we assume that the choice oracle is invoked only once, and at the beginning of the run. Note however that we will define a nondeterministic machine, but one which uses "non-atomic nondeterminism", which involves guessing a number in $\{0, 1\}$. Non-atomic nondeterminism can be eliminated, using the standard determinisation construction. Actually, our machine will be such that it can be determinised with polynomial slowdown.

Simulating a bounded number of such invocations can be done by nesting obtained machines. However, since the machines are nondeterministic, only a bounded number of nestings of such machines can be simulated by a deterministic machine.

Lemma C.3. *Let M be a deterministic Turing machine, and let $\{q_a\}_a \in \mathbb{A}$ be an equivariant family of states of M. For $a \in \mathbb{A}$, let M_a denote the machine M, but whose initial state is q_a.*

Then the following language is recognizable by a Turing machine using non-atomic nondeterminsm:

$\{w :$ *$L(M_a)$ for some a in the least support of the first letter of w*$\}$.

Proof: Let $d \in \mathbb{N}$ be such that all letters in the input alphabet have a support of size at most d. The machine M' has the following state space:

$$\tilde{Q} \stackrel{\text{def}}{=} \bigcup_{\substack{X \subseteq \mathbb{A} \\ |X| \leq d}} (X \to Q).$$

This set $\tilde{Q}$ orbit-finite, since it is an orbit-finite union of orbit-finite sets. For a state $\tilde{q} \in \tilde{Q}$, which is a function $q : \tilde{X} \to Q$, we will call X the domain of $\tilde{q}$. Observe that the domain is a set of at most d atoms. A state in $\tilde{q}$ should be interpreted as

> The guessed atom is in the domain of $\tilde{q}$. Furthermore, if the guessed atom is x, then the current state of the machine is $\tilde{q}(x)$.

The work alphabet of the new machine is defined in the same way:

$$\tilde{B} \stackrel{\text{def}}{=} \bigcup_{\substack{X \subseteq \mathbb{A} \\ |X| \leq d}} (X \to B).$$

Let w be an input word, and let X denote the least support of the first input letter. The letters of the input alphabet are naturally viewed as elements of the work alphabet, where a letter $a \in A$ is interpreted as the constant mapping which maps each element of X to a. The initial state is the mapping which maps an element $x \in X$ to the state q_x.

The following invariant will be satisfied throughout the computation.

> Suppose that n steps of computation have elapsed and the current state has domain X. Then
> 1) The domain of all letters on the tape is bigger or equal to X.
> 2) Let $x \in X$. If we feed x to the functions in the state and the tape cells, we recover the n-th computation step of the machine M_x on the original input (the head position will be the same).

We now describe a single computation step of the machine. Suppose that its state is $\tilde{q}$ and the letter on the current cell is $\tilde{b}$. Let X be the domain of $\tilde{q}$. For each $x \in X$, the machine M_x makes a certain transition when its state is $\tilde{q}(x)$ and its input letter is $\tilde{b}(x)$; suppose that for $x \in X$, this transition is

$$(\tilde{q}(x), \tilde{b}(x)) \to (\tilde{q}'(x), \tilde{b}'(x), \tilde{d}(x)).$$

This defines functions $\tilde{q}', \tilde{b}', \tilde{d}'$ from X to Q, B and $\{-1, 0, 1\}$, respectively. The function $\tilde{b}'$ is in the work alphabet, so it can potentially be written to the tape. The function $\tilde{q}'$ is in the state space, so it can potentially be chosen as a new state. The interesting issue concerns the direction: depending on the choice of atom $x \in X$, the machine might want to move left, stay, or move right; while the head cannot split. Let us then partition X into three sets:

$$X_{-1}, X_0, X_1,$$

where $X_i = \tilde{d}^{-1}(i)$ for $i = -1, 0, 1$. The key point is that the machine can compute this partition, because it can compute the direction function $\tilde{d}$. Therefore, the machine guesses a number $i \in \{-1, 0, 1\}$ and does the following:

- On the current cell, it writes the function $\tilde{b}'$, but restricted to the set X_i;
- Assumes the state $\tilde{q}'$, but restricted to the set X_i;
- Moves the head in direction i.

This finishes the proof of the lemma. ■

D. Fresh algebras

A result analogous to Proposition V.1 holds for orbit-finite algebras. We will need such a result in Appendix D-C.

The definition of an orbit-finite algebra allows the universe to be orbit-finite, but the set of operations must be finite. Consider an orbit-finite algebra A. If we extend A by adding one constant per each atom in $\mathbb{A}$, we end up with an algebra where the set of operations is no longer finite (although it is still orbit-finite). We denote the resulting algebra $A_{\mathbb{A}}$.

Let val be a valuation with values in A, i.e. a mapping from some set of variables to A. We say that a term t of the algebra $A_{\mathbb{A}}$ is *fresh* with respect to val, if the least support of t is disjoint from the least support of val and from the least support of A.

The following proposition shows that $A_{\mathbb{A}}$ is equivalent to an algebra with finitely many operations, when only fresh terms are concerned.

Proposition C.4. *For every orbit-finite algebra A there exists an orbit-finite algebra A_{fresh} with the same support, having the following property:*

> *For each boolean term t over $A_{\mathbb{A}}$ there is a boolean term t_{fresh} over A_{fresh} such that $t[\mathrm{val}] = t_{\text{fresh}}[\mathrm{val}]$ for every valuation ν for which t is fresh.*

In order to prove the above proposition, first we extend the notions from Section C-A.

a) Higher orders: We extend the notions from Section C-A, allowing to remove from the support of an element x all the atoms which come from a linearly ordered finite set C. The resulting element will be denoted $[C^*].x$. The formal definition follows.

First observe that $[\mathbb{A}]$ can be iterated; we define $[\mathbb{A}]^0 X \overset{\text{def}}{=} X$ and $[\mathbb{A}]^i X \overset{\text{def}}{=} [\mathbb{A}]([\mathbb{A}]^{i-1} X)$ for $i = 1, 2, \ldots$. We say that the elements of $[\mathbb{A}]^k X$ have *order* k. By $[\mathbb{A}]^{\leq i}$ we denote $X \cup [\mathbb{A}]X \cup \ldots \cup [\mathbb{A}]^i X$. Observe that if X is orbit-finite, then so is $[\mathbb{A}]^{\leq i} X$, for $i = 0, 1, 2, \ldots$.

Let $C = (c_1, c_2, \ldots, c_k)$ be a tuple of distinct atoms. By $|C|$ we denote the set $\{1, \ldots, k\}$. For any element x, let $1 \leq i_1 < i_2 < \ldots < i_j \leq k$ be such that $\mathrm{supp}(x) \cap \{c_1, c_2, \ldots, c_k\} = \{c_{i_1}, c_{i_2}, \ldots, c_{i_j}\}$. Define:

$$[C^*].x \overset{\text{def}}{=} ([c_{i_1}].[c_{i_2}].\ldots[c_{i_j}].x, (i_1, i_2, \ldots, i_j)).$$

In particular, $[C^*].x \in [\mathbb{A}]^j X \times |C|^j$. Let $[C^*]X$ denote the set of all elements of the form $[C^*].x$, where $x \in X$. Below, by $|C|^{\leq d}$ we denote the set of tuples of length at most d of elements of $|C|$.

Fact C.5. *If every element of X has a support of size at most d, then*

$$[C^*]X \subseteq [\mathbb{A}]^{\leq d} X \times |C|^{\leq d},$$

and $\mathrm{supp}([C^*]X) = \mathrm{supp}(X) - \mathrm{supp}(C)$.

Proposition C.6. *Let C be a finite tuple of distinct atoms. Then,*

1) *For any x, $\mathrm{supp}([C^*].x) = \mathrm{supp}(x) - C$.*
2) *Any mapping $f\colon X \to Y$ induces a mapping $[C^*]f\colon [C^*]X \to [C^*]Y$, such that*

$$([C^*]f)([C^*].x) = [C^*].f(x).$$

Moreover, the least support of $[C^*]f$ is contained in the least support of f, and the mapping $[C^*]f$ only depends on the length of the tuple C, and not on C itself.

3) *There is a bijection $[C^*](X \times Y) \simeq [C^*]X \times [C^*]Y$, which maps $[C^*].(x, y)$ to $([C^*].x, [C^*].y)$.*

b) Proof of Proposition C.4: Let d be a number such that each element of A has a support of size at most d. The universe of the new algebra A_{fresh} is $[\mathbb{A}]^{\leq d} A$; note that this is an orbit-finite set. Let N be the maximal arity of a function in the algebra A. We equip the algebra A_{fresh} with all functions of arity at most N, whose support is contained in the least support of A. There are only finitely many such functions. It remains to show that the algebra A_{fresh} satisfies the required property, i.e. that a term t of the algebra $A_{\mathbb{A}}$ can be 'simulated' by a term t_{fresh} of the algebra A_{fresh}, when restricted to valuations for which t is fresh.

Let t be a term over the algebra $A_{\mathbb{A}}$. A *fresh constant* of t is an atom a which belongs to $\mathrm{supp}(t) - \mathrm{supp}(A)$. Note that each fresh constant is a subterm of t, but is not a legal term in the algebra A. The aim is to eliminate the fresh constants from the term t.

As t is a finite term, we may linearly order its subterms, for instance by performing a Depth-First-Search. In particular, this induces a linear ordering of the fresh constants. By C_t we denote the tuple of all fresh constants, ordered increasingly according to the above linear order. Then, for an element x, $[C_t^*].x$ can be interpreted as a way of extracting from the support of x all the fresh constants of t.

Fact C.7. *For any element x,*

$$[C_t^*].x = ([c_{i_1}].[c_{i_2}].\ldots,[c_{i_j}].x, (i_1, \ldots, i_j)),$$

where $c_{i_1}, \ldots, c_{i_j}$ are all the fresh constants of $\mathrm{supp}(x)$ *(in the ordering induced by t).*

We will prove the following:

Claim C.7.1. *For any term t over the algebra $A_{\mathbb{A}}$ there exists a term t_{fresh} over the algebra A_{fresh}, such that for any valuation* val *for which t is fresh,*

$$[C_t^*].t[\mathrm{val}] = (t_{\text{fresh}}[\mathrm{val}], I_0) \tag{8}$$

for some tuple of numbers $I_0 \in |C|^{\leq d}$.

If t is a Boolean term, then $t[\mathrm{val}]$ has empty support, so $[C_t^*].t[\mathrm{val}] = (t[\mathrm{val}], \varepsilon)$, where ε is the empty tuple. In particular, (8) implies $t_{\text{fresh}}[\nu] = t[\mathrm{val}]$, proving Proposition C.4.

It remains to prove the above claim. The proof of proceeds by induction on the structure of t.

If $t = x$ for some variable x, then we set $t_{\text{fresh}} = x$. Let val be a valuation which maps x to an element $a \in A$. Then, $t[\mathrm{val}] = a$, so

$$[C_t^*].t[\mathrm{val}] = [C_t^*].a$$

If t is fresh with respect to a, then $\mathrm{supp}(a) \cap C_t = \emptyset$, so $[C_t^*].a = (a, \varepsilon) = (t_{\text{fresh}}[\mathrm{val}], \varepsilon)$. This gives the required property (8), for $I_0 = \varepsilon$.

If $t = a$ for atom a which is not a constant of A, we set $t_{\text{fresh}} = [a].a$. Let val be a valuation for which t is fresh. Then,

$$[C_t^*].t[\mathrm{val}] = [C_t^*].a = ([a].a, (1)) = (t_{\text{fresh}}[\mathrm{val}], (1)),$$

yielding (8) for $I_0 = (1)$.

It remains to consider the interesting case, when $t = f(t^1, \ldots, t^n)$ for some function f of the algebra A. For $i = 1, 2, \ldots, n$, let t^i_{fresh} be the term obtained from t^i by using the inductive assumption. Let val be a valuation for which t is fresh; then, by assumption,

$$[C^*_{t^i}].t^i[\text{val}] = (t^i_{\text{fresh}}[\text{val}], I_i) \qquad (i = 1, 2, \ldots, n), \tag{9}$$

for some tuples of numbers $I_1, I_2, \ldots, I_n$. Since t is fresh for val,

$$\text{supp}\big(t^i\big) \cap \text{supp}\big(t^i[\text{val}]\big) = \text{supp}(t) \cap \text{supp}\big(t^i[\text{val}]\big)\,.$$

This gives:

$$[C^*_{t^i}].t^i[\text{val}] = [C^*_t].t^i[\text{val}].$$

Therefore, equation (9) yields:

$$[C^*_t].t^i[\text{val}] = (t^i_{\text{fresh}}[\text{val}], I_i) \qquad (i = 1, 2, \ldots, n). \tag{10}$$

Recall the mapping $[C^*_t]f$ defined in Proposition C.6; its support is contained in the support of f, which is contained in the support of A. We have:

$$\begin{aligned}
[C^*_t].t[\text{val}] &= [C^*_t].f(t^1[\text{val}], \ldots, t^n[\text{val}]) = \\
&= ([C^*_t]f)\big([C^*_t].t^1[\text{val}], \ldots, [C^*_t].t^n[\text{val}]\big) \\
&\overset{(10)}{=} ([C^*_t]f)\big((t^1_{\text{fresh}}[\text{val}], I_1), \ldots, (t^n_{\text{fresh}}[\text{val}], I_n)\big) \\
&= (\tilde{a}_0, I_0),
\end{aligned}$$

for some $\tilde{a}_0 \in A_{\text{fresh}}$ and some tuple of numbers I_0. Define an n-ary mapping g so that for $\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_n \in A_{\text{fresh}}$,

$$g(\tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_n) = \tilde{a}_0$$

whenever there is a tuple I_0 such that

$$([C^*_t]f)\big((\tilde{a}_1, I_1), \ldots, (\tilde{a}_n, I_n)\big) = (\tilde{a}, I_0).$$

Since the definition of g only refers to A_{fresh}, the mapping $[C^*_t]f$ and the equivariant tuples $I_1, \ldots, I_n$, it follows that g supported by the support of A. Therefore, g is a function of the orbit-finite algebra A_{fresh}. To conclude, there is a function g of the algebra A_{fresh} such that for each valuation val for which t is fresh, there exists a tuple of numbers I_0 such that

$$[C^*_t].t[\text{val}] = (g(t^1_{\text{fresh}}[\text{val}], t^2_{\text{fresh}}[\text{val}], \ldots, t^n_{\text{fresh}}[\text{val}]), I_0).$$

Therefore, setting $t_{\text{fresh}} = g(t^1_{\text{fresh}}, t^2_{\text{fresh}}, \ldots, t^n_{\text{fresh}})$ gives a term with the required property. This ends the proof of Proposition C.4.

Appendix D
Proofs for Section VI

We use the notation $x \sim_S y$ when x and y are in the same S-orbit, i.e. $y = \pi(x)$ for some S-automorphism π. Thus $L_{A,S} = \{wv : w \sim_S v\}$. For two sets of atoms we call them S-disjoint if their intersection is included in S.

A. Equivariant alphabets

For convenience, in the sequel we assume that the alphabets are equivariant. No generality is lost as the alphabet may be restricted to any finitely-supported subset by a language $L \subseteq A^*$.

Every orbit O of A is a surjective image of non-repeating tuples of atoms. This means that there is some $k \in \mathbb{N}$ and a surjective mapping

$$\alpha : \mathbb{A}^{(k)} \to O. \tag{11}$$

Indeed, α is obtained by choosing a pair

$$(t, a) \in \mathbb{A}^{(k)} \times O,$$

where atoms appearing in t are the least support of a, and then closing the pair on all atom automorphisms. The mapping α can be computed by a deterministic machine, but the inverse function α^{-1} requires nondeterminism in general. In the sequel we will use the same symbol α to denote the mapping (11) for every orbit of A.

A set is called *straight* if each of its orbits is isomorphic to $\mathbb{A}^{(k)}$ for some k. This corresponds to the mapping α being a bijection, and implies that the inverse function α^{-1} is computable by a deterministic machine.

B. Straight alphabets

Before dealing with arbitrary alphabets in the sequel, consider here only straight ones. Straight alphabets admit determinisation, as shown below:

Lemma D.1. *Turing machines over straight alphabets admit determinisation.*

Proof: Essentially we prove that in case of straight alphabets, the general nondeterminism of Turing machines may be simulated by two restricted variants: the non-atomic nondeterminism and fresh-atoms-nondeterminism. The former one may be eliminated in the classical way, by an exhaustive enumeration of all computation paths. Then the latter one may be also eliminated, due to Proposition V.1.

Consider a machine M recognizing a language L over a straight alphabet A. We may assume that the work alphabet and state space are also straight. Indeed, if this is not the case we replace the two sets by their inverse images along the mapping (11); then the transition relation is also replaced by its inverse image. We may also assume that the machine has a single initial state, as otherwise we add a new state from which the machine nondeterministically guesses one of the former initial states. Thus we only need to show how a single nondeterministic transition step of M may be simulated deterministically.

Fix a configuration of M and its least support S. The configuration includes the local configuration consisting of a state q, a head position, and a tape content u at the head position. Denote by $N_{q,u}$ the set of all triples $\langle q', u', d\rangle$ such that $\langle q, u, q', u', d\rangle$ is in the transition relation of M, where d stands for the direction of head move. $N_{q,u}$ has finitely many S-orbits, being a subset of an orbit-finite set. The simulation of the step of M consists of two phases:

- Choose one S-orbit of the set $N_{q,u}$; this may require scanning the whole tape content and may involve non-atomic nondeterminism.
- Build q' and u'; this may require using fresh-atoms-nondeterminism.

As both kinds of nondeterminism involved may be eliminated, there is a deterministic machine recognizing L. ■

C. Optimal least supports

Before showing Theorem VI.3 we prove two auxiliary lemmas, Lemma D.4 and D.5, that state that the support of a deterministic machine, or an orbit-finite algebra, needs not be greater than the support of the language recognized. The idea of both the proofs below is the same: atoms in the least support of the algebra/machine that are not in the least support of the language are irrelevant and thus may be renamed arbitrarily without changing the language recognized by the algebra/machine.

Lemma D.2. *Let M be a deterministic Turing machine with the least support T. For any $S \subseteq T$, there is a equivalent deterministic machine whose state-space, work alphabet and transition function are supported by S.*

(Note however that we do not claim the initial state or the accepting states to be supported by S.)

Proof: Let Q, B and δ denote the the state-space, work alphabet and the transition function of M. Wlog. assume that a tuple containing all the atoms from $T - S$ is permanently explicitly stored in the state of M.

The machine M' is obtained from M by essentially closing on S-orbits, i.e. by applying all S-automorphisms. Namely, let Q' be the union of all S-orbits of all elements of Q (clearly $Q \subseteq Q'$), and similarly let B' the the union of S-orbits of elements of B, and let δ' be the union of all S-orbits of all tuples in the graph of δ. Clearly in state $\pi(q)$, for an S-automorphism π, the machine M' stores, instead of the atoms from $T - S$, the atoms from $\pi(T - S)$.

That δ' is a function, follows easily from the assumption that M stores atoms from $T - S$. ■

The following lemma is an analog of the last one for algebras. It is proved similarly, hence we omit the proof.

Lemma D.3. *Let $\mathcal{A}$ be an orbit-finite algebra with the least support T. For any $S \subseteq T$, there is an algebra with constant operations, one for every atom from $T - S$, such that the universe of the algebra and all other operations are S-supported.*

Lemma D.4. *If a language L is recognized by a deterministic TM, then the language is recognized by some deterministic TM with the same least support as L.*

Proof: Suppose the language, say L, with the least support S, is recognized by a deterministic TM M with the least support T. Necessarily $S \subseteq T$ as the language of a machine is an equivariant function of the machine. S-automorphisms do not change L, hence:

$$L(\pi(M)) = L(M) \qquad \text{for every } S\text{-automorphism } \pi. \quad (12)$$

Consider all injections ρ from T to $\mathbb{A}$ that fix S. We write $\rho(M)$ for action of any extension of ρ to all atoms; as T supports M, this notation is unambiguous. The set of all such injections ρ is orbit-finite.

By Lemma D.2 we may assume that the atoms from T are stored initially in state of M, and the the transition function of M is supported by S. In other words, the transition function of $\rho(M)$ is the same as the one of M.

Construct a nondeterministic machine M' that works as follows, for a given input word w: it guesses $|T - S|$ fresh atoms (these atoms are thus outside of S and outside of the least support of w), stores them in place of atoms from $T - S$ (any such guessing defines an injection ρ from T to $\mathbb{A}$ that fixes S), and then runs the machine $\rho(M)$ (whose transition function is the same as the one M). The machine M' is supported by S and recognizes the same language as M. The equivalence of M and M' follows by (12).

We may apply Proposition V.1 to the machine M' to get an equivalent deterministic one. As the construction in the proof of Proposition V.1 does not increase the least support, we are done. ■

In fact the same simple argument works for nondeterministic machines as well. In this section however we will only need the deterministic version.

Lemma D.5. *If a language L is recognized by a an orbit-finite algebra then the language is recognized by some algebra with the same least support as L.*

Proof: Similarly as in the proof of Lemma D.4, but using Proposition C.4 instead of Proposition V.1. Let S and T denote the least supports of the language L, and an algebra A recognizing L, respectively. $S \subseteq T$.

Due to Lemma D.3 we may assume without loosing generality that atoms in $T - S$ are constants in the algebra $\mathcal{A}$ recognizing L, and that all the remaining operations and the universe are supported by S. Then define an orbit-finite algebra $\mathcal{A}'$ that differs from $\mathcal{A}$ only with the following: instead of every constant for $T - S$, there is a one-orbit set of atom-constants, one per every atom outside S. The orbit-finite algebra obtained in this way is supported by S and still recognizes L.

Finally, using Proposition C.4 we transform the orbit-finite algebra $\mathcal{A}'$ into an ordinary algebra $\mathcal{A}''$ with finitely many operations. As the construction of Proposition C.4 does not increase the least support, the algebra $\mathcal{A}''$ is supported by S, i.e. invariant under S-automorphisms. We thus only need to show that $\mathcal{A}''$ still recognized L.

Let t_n be a term recognizing L over words of length n in $\mathcal{A}'$. We know that there is a term t'_n of $\mathcal{A}''$ such that for any $v \in A^n$ whose least support is S-disjoint from the least support of t_n, that is S-disjoint from atom-constants appearing in t_n,

$$t_n[v] = t'_n[v].$$

We claim that the term t'_n of $\mathcal{A}''$ recognized L over words of length n. Indeed, applied to a word v that is S-disjoint from t'_n yields the correct result. Then by invariance of the algebra

$\mathcal{A}''$ under S-automorphisms one gets the correct result of $t_n[v]$ for every $v \in A^n$. ■

D. Distinguishing by algebras

Let A be now an arbitrary orbit-finite alphabet. For an orbit-finite algebra with the universe including A, we say that the algebra distinguishes two words $w, v \in A^n$ if there is a term t with

$$t[w] = \textit{true} \quad \text{and} \quad t[v] \neq \textit{true}.$$

Lemma D.6. *For every algebra, there is a polynomial-time deterministic TM that decides, for given words w, v, whether the algebra distinguishes w and v.*

Proof: We show that for every orbit-finite algebra, there is a deterministic polynomial-time Turing machine that decides, for given two words, whether the algebra distinguishes these words.

Fix an algebra for the rest of the proof. Let A be the universe of the algebra. For a word $w = a_1 \dots a_n \in A^*$, define the subalgebra generated by w, as the least subalgebra containing $\{a_1 \dots a_n\}$. As the algebra is fixed, we claim the following:

Claim D.6.1. *There is a polynomial p, such that the subalgebra generated by any word $w \in A^*$ of length n has at most $p(n)$ elements.*

Proof: Denote by T_w the union of the least supports of the algebra and of the word w. The size of T_w is linear with respect to the length of w. For any term t, the value $t[w]$ in the algebra is supported by T_w, and thus every element in the subalgebra generated by w is supported by T_w. The claim follows if one proves that there are only polynomially many elements in the algebra supported by T_w, which in turn follows from the following simple observations:

- the size of the least support of every element in the subalgebra is bounded by a constant;
- the number of k-element subsets of T_w is polynomial, for a constant k;
- for every orbit of the universe of the algebra, the number of its different elements sharing the least support is constant (at most $k!$ if the size of the least support is k).

■

The remaining part of the proof is easy with the above claim. The deterministic machine works as follows, given input words w and v. It computes the following relation between elements of the subalgebras generated by w and v, respectively:

$$\{\langle t[w], t[v] \rangle : t \text{ a term}\}.$$

By the above claim, the relation may be computed incrementally in polynomial time, using values $t[w]$, $t[v]$ of only polynomially many terms t. If the value *true* is related to any element different that *true*, the machine decides 'yes'. Otherwise *true* is related (possibly) only to *true*, hence the machine decides 'no'. ■

We say that an algebra *distinguishes S-orbits of A^** if the algebra distinguishes precisely those pairs of words that are not in the same S-orbit: for every two words w, v of the same length,

$$w \sim_S v \iff \forall t \ (t[w] = \textit{true} \Leftrightarrow t[v] = \textit{true}). \quad (13)$$

E. Proofs of Theorems VI.1 and VI.3

The two theorems follow from Theorems D.7, Lemma D.8 and Theorem D.9 formulated and proved below.

Theorem D.7. *For any finite subset S of atoms and any orbit-finite alphabet A, the following conditions are equivalent:*

1) *the language $L_{A,S}$ is in* P*;*
2) *the language $L_{A,S}$ is recognized by a deterministic TM;*
3) *the language $L_{A,S}$ is recognized by some orbit-finite algebra;*
4) *there is an orbit-finite algebra that distinguishes S-orbits of A^*;*

Proof: We aim at the circular sequence of implications:

$$1) \implies 2) \implies 3) \implies 4) \implies 1).$$

The first implication The implication 2) $\implies$ 3) follows by Theorem III.2. Finally, the implication 4) $\implies$ 1) follows by Lemma D.6. Thus we only need to show the single implication 3) $\implies$ 4).

For the proof of the implication 3) $\implies$ 4), fix $\mathcal{A}$ an algebra that recognizes $L_{A,S}$. By Lemma D.5 we may assume the algebra to be S-supported. It is enough to show that there is another algebra that is supported by S and recognizes any S-orbit of A^*, i.e. such that for every $w \in A^n$ there is a term t_w such that

$$t_w[v] = \begin{cases} \textit{true} & \text{if } v \sim_S w \\ \textit{false} & \text{if } v \not\sim_S w \end{cases} \quad \text{holds for every } v \in A^n. \quad (14)$$

As the first step we define, using the algebra $\mathcal{A}$, another algebra $\mathcal{A}'$ with an orbit-finite set of operations (thus we relax here the definition from Section III-B). The algebra $\mathcal{A}'$ will recognize every S-orbit of A^*. The universe of $\mathcal{A}'$ is the same as the universe of $\mathcal{A}$. The operations of $\mathcal{A}'$ are all the operations of $\mathcal{A}$, extended with two families of new operations. The first family is infinite but orbit-finite, and contains one constant operation for every atom. The second family is finite and contains the equivariant mappings α, as in (11), one for every orbit of the alphabet A. Thus, for every element a of the alphabet A, there is a term t'_a without variables denoting the value a.

To show that $\mathcal{A}'$ recognizes every S-orbit of A^* consider any $w = a_1 \dots a_n \in A^n$ and the term t_{2n} that recognizes $L_{A,S}$ over words of length $2n$. Indeed, the term

$$t_w(x_1, \dots, x_n) \ = \ t_{2n}(x_1, \dots, x_n, t'_{a_1}, \dots, t'_{a_n}), \quad (15)$$

obtained by substituting $t'_{a_1} \dots t'_{a_n}$ in place of $x_{n+1} \dots x_{2n}$, has the required property (14). Note that the particular choice of the word w inside the S-orbit is inessential. The idea used below is that for any v, one could choose a word w such that the least supports of v and w are S-disjoint. In short, we will say that v and w are S-disjoint.

As the second step, we apply Proposition C.4 to obtain an algebra $\mathcal{A}''$ with only finitely many operations. The algebra $\mathcal{A}''$ is still supported by S and thus all its operations are invariant under S-automorphisms. Moreover, $\mathcal{A}''$ witnesses the following property: for any boolean term t of $\mathcal{A}'$ there is a corresponding term t' of $\mathcal{A}''$ such that for any v whose least support is S-disjoint from the least support of t,

$$t[v] = t'[v].$$

We claim that the term t' of $\mathcal{A}''$ corresponding to the term t_w (15) of $\mathcal{A}'$ satisfies the property (14). Indeed, applied to a word v that is S-disjoint from w yields the correct result. Then by invariance of the algebra $\mathcal{A}''$ under S-automorphisms one gets the correct result of $t[v]$ for any $v \in A^n$.

The proof of Theorem VI.3 is thus completed. ■

In fact the equivalent conditions in Theorem VI.3 hold either for all supports S, or for none, which follows from:

Lemma D.8. *For a finite set S of atoms and any orbit-finite alphabet A, the following conditions are equivalent:*

1) *the language $L_{A,S}$ is recognized by a deterministic TM*
2) *the language $L_{A,\emptyset}$ is recognized by a deterministic TM*

As a conclusion, if one of the conditions 1)–4) in Theorem VI.3 holds for some S then every condition holds for every S. The conditions define thus the property of an alphabet A, independently from the set S.

Proof: For the implication $1 \implies 2$, observe that

$$L_{A,\emptyset} = \bigcup_T L_{A,T}$$

where T ranges over finite sets of atoms of the same cardinality as S. In other words, two words are in the same $\emptyset$-orbit if and only if they are in the same T-orbit, for some set of atoms T disjoint with the least supports of the words. Indeed, if a permutation of atoms maps a word to a word, then a permutation may be chosen to fix some T outside of the least supports of the words. As $L_{A,S}$ is recognized by a deterministic machine, we know that each of languages $L_{A,T}$ is recognized by a deterministic TM too. Thus by Corollary V.3 we get a deterministic TM for $L_{A,\emptyset}$.

For the opposite implication $2 \implies 1$, suppose there is a deterministic TM M that recognizes $L_{A,\emptyset}$. For a fixed S, we define a machine M_S to recognize $L_{A,S}$ as follows. In the preprocessing, the machine M_S will append to each half w, v of the input word wv a suffix $u_S \in A^*$ that satisfies the following condition:

$$\pi(u_S) = u_S \quad \Leftrightarrow \quad \pi \text{ is an } S\text{-automorphism} \tag{16}$$

for any automorphism π of atoms. Then M_S runs the machine M on the input wu_Svu_S. Indeed, condition (16) guarantees correctness:

$$w \sim_S v \Leftrightarrow wu_S \sim_\emptyset vu_S.$$

It remains only to show that a word satisfying (16) exists and may be computed by the machine.

Choose any infinite orbit O of A. We may assume that cardinality of S is strictly greater than the dimension of O (by dimension we mean cardinality of the least support of its elements). Indeed, otherwise the machine M_S guesses the necessary number of fresh atoms, artificially extending the set S. Then by Proposition V.1 the guessing may be eliminated.

The word u_S is any word from O^* such that:

- the least support of every letter is a subset of S;
- for every letter a except the last one, there is exactly one atom that belongs to the least support of a but does not belong to the least support of the next letter (call such atom *leaving* a);
- every atom from S is leaving some letter.

The conditions enforce (16), as every permutation that maps a word to itself must necessarily fix every atom leaving any letter. ■

Theorem D.9. *Let A be an orbit-finite set. If A satisfies the conditions 1)–4) of Theorem VI.3, then*

i) *Turing machines over alphabet A admit determinisation;*
ii) P=NP *over alphabet A iff* P=NP *classically.*

If A does not satisfy the conditions 1)–4) of Theorem VI.3 then there are languages in NP *that are not deterministically semi-decidable. (In particular,* P≠NP.*)*

Proof: If the input alphabet A does not satisfy the conditions 1)–4) of Theorem VI.3, then the canonical language $L_{A,\emptyset}$ is a witness that belongs to NP, due to Lemma VI.2, but is not deterministically semi-decidable. Assume thus for the following that the alphabet does satisfy the conditions 1)–4) of Theorem VI.3.

i) Turing machines over A admit determinisation.

Suppose that a language $L \subseteq A^*$ over a determinizable alphabet A is recognized by a nondeterministic TM M. We will construct an equivalent deterministic TM M'. For convenience we will use the non-atomic nondeterminism, which may be easily eliminated in the classical way, and the fresh-atoms-nondeterminism, which may be eliminated by Proposition V.1.

Let S be the least support of L. For an input word $w = a_1 \dots a_m$, denote by n the number of elements of the least support of w that do not belong to S.

In the first phase, the machine M' constructs a word $v = b_1 \dots b_n$ that is S-disjoint from w and such that $w \sim_S v$. This is done in three steps:

- Guess n fresh atoms that do not appear in the least support of w; this involves fresh-atoms-nondeterminism.
- For each position i of the word w, guess a tuple t_i and a letter $b_i \in A$ such that $t_i \in \alpha^{-1}(b_i)$; the letter b_i may be an arbitrary element of A supported by the union of S and the fresh atoms. This step involves non-atomic nondeterminism.
- Check whether $w \sim_S v$.

After the successful test above, $w \in L$ if and only if $v \in L$. Thus after the first phase the input word might be safely replaced by v. The advantage is that for each letter of v, the machine knows a tuple of atoms that belongs to that letter. This will enable simulation of nondeterministic behavior of M using Lemma D.1, as described below.

After the first phase described above, the machine M' actually ignores both words w and v, and only works with the word $t = t_1 \dots t_n$. The idea is to check whether the latter word belongs to the *straightening* of L, defined as the inverse image of L along the mapping (11):

$$\{u_1 \dots u_m : u_1 \in \alpha^{-1}(b_1), \dots, u_m \in \alpha^{-1}(b_m),\ b_1 \dots b_m \in L\}.$$

This is doable due to the following:

Claim D.9.1. *If a language L is recognized by a TM then the straightening is also so.*

Indeed, the machine recognizing the straightening may be constructed from the machine recognizing L by taking the inverse image, along the mapping α (11), of the work alphabet, of the state space, of the subsets of initial and accepting states, and of the transition relation.

To complete the construction of machine M', consider the nondeterministic machine recognizing the straightening of L. By Lemma D.1 there is an equivalent deterministic machine. Thus in the second phase, the machine M' executes that deterministic machine on the input t.

For correctness, recall that the mapping α maps the word t to v. Thus the word t is accepted by M' in the second phase if and only if $v \in L$, which in turn is equivalent to $w \in L$.

ii) P=NP over A if and only if P=NP classically.

We start with an easier 'only if' direction: assuming P=NP over the input alphabet A, we will show that P=NP classically. The proof works in fact for an arbitrary alphabet A.

Observe that a finite alphabet is a special case of an orbit-finite equivariant set, each orbit being a singleton. Thus a language L over a finite alphabet may be considered as a language over an orbit-finite equivariant alphabet.

It is sufficient to show that every deterministic machine M with atoms over a finite equivariant alphabet is actually a classical machine, without atoms. Indeed, applying Lemma D.4 allows to assume that M is equivariant. Then, as every configuration of a deterministic machine is supported by the least support of input, we deduce that every configuration of M is equivariant. Thus the state space and work alphabet are necessarily finite.

Now we turn to the 'if' direction. The proof is based on atom-less encoding, as in Section IV. In the sequel we assume that operations on encodings are implemented efficiently in polynomial time. This is legitimate due to the representation theorem of [4]. Namely, due to the result of [4] one can assume without loss of generality that elements of A are hereditarily finite sets. Such sets may be described using essentially only atoms, the empty set $\emptyset$, and symbols '{' and '}', for instance

$$\{\{a, b, \emptyset\}, \{a\}, b\}.$$

Then an efficient encoding may be obtained by replacing atoms with natural numbers, via an arbitrary bijection between the two sets.

The proof is based on the two claims stated below, both being refinements of the two implications in Theorem IV.1.

Claim D.9.2. *If a language $L \in A^*$ is recognized by a nondeterministic polynomial-time TM with atoms, then $encode(L)$ is recognized by a nondeterministic polynomial-time TM without atoms.*

Claim D.9.3. *For a finitely-supported language $L \in A^*$, if $encode(L)$ is recognized by a deterministic polynomial-time TM without atoms, then L is recognized by a deterministic polynomial-time TM with atoms.*

The two claims easily prove the theorem. Assume that P=NP classically and let L be a language over an orbit-finite alphabet A that belongs to NP, i.e. is recognized by a nondeterministic polynomial-time TM. We need to prove that L is recognized by a deterministic polynomial-time TM. By the first claim, $encode(L)$ is in NP classically. By the assumption that P=NP classically, we get that $encode(L)$ is in P. Finally, by the second lemma we obtain that L is in P with atoms, as required.

Now we justify the claims. The first claim is proved analogously to the first implication of Theorem IV.1 and actually works for an arbitrary alphabet.

The proof of the second claim is again analogous to the second implication of Theorem IV.1, but this time a little more care is needed. In particular we will have to use the assumption on the alphabet A.

Let S be a support of L. We first demonstrate that the language

$$\{w\#encode(u) : w, u \in A^* \text{ and } w \sim_S u\} \qquad (17)$$

is recognized by a deterministic TM in polynomial time. On input $w\#v$, the machine guesses fresh atoms, one per every atom appearing in w, and deterministically produces a word $u \in A^*$, supported by the union of S and the guessed fresh atoms, such that

$$v = encode(u') \quad \text{for some } u' \sim_S u.$$

Then the machine tests if $w \sim_S u$, in deterministic polynomial time due to the assumption on A. Thus we have shown that the language (17) is recognized in deterministic polynomial-time.

Now assume that L is finitely supported and $encode(L)$ is recognized by a deterministic polynomial-time TM without atoms. A natural way to recognize L would be to guess first an encoding v of the input word w, up to S-automorphism, then check correctness by the membership of $w\#v$ test in (17), and finally run the deterministic machine without atoms on v. We claim the the nondeterministic guessing in the first step may be eliminated in polynomial time if, instead of guessing the whole encoding, the machine would incrementally add encodings of consecutive letters. The correctness check, using the membership test in (17), would be done after each letter. We will thus only describe how encoding of one letter is added.

An crucial observation is that it is sufficient to use only encoding of polynomial size, as we are only interesting in encoding of a word up to S-permutation (the polynomial encodings again follow by the representation theorem of [4]). Thus, when encoding of a next letter is guessed, there is only polynomially many possible results of guessing. In consequence, the guessing may be eliminated while staying in P. ∎

Example D.1. We end this section with an example of an alphabet A for which the langauge $L_{A,S}$ is in P, but the decision procedure is not as apparent as in the examples from Section VI. The alphabet is similar to the one used in Section III-A, with the only difference that a letter will have two side sets instead of three. Formally, the alphabet A contains sets of the form:

$$\{(a,a'),(b,b')\}$$

but the two sets:

$$\{(a,a'),(b,b')\} \quad \text{and} \quad \{(a',a),(b',b)\}$$

are not distinguished. The alphabet has one-orbit. Equivalently, A may be presented in canonical form as $\mathbb{A}^{(4)}/\equiv$, where the group inducing $\equiv$ is generated by two permutations:

$$(1\,2)\,(3\,4) \qquad (1\,3)\,(2\,4).$$

We demonstrate that the language $L_{A,S}$ is recognized by a deterministic TM; by our results shown later in this section if follows that the language is necessarily in P. Assume for simplicity that the side sets of different letters are either disjoint or equal, and that $S = \emptyset$.

The idea is to consider side sets as nodes in a graph, and letters in the input word as undirected edges between side sets. Note however that the edges are ordered according to their appearance in the input word.

The deterministic decision procedure for $w \sim_S v$ runs in two phases. In the first phase two graphs are built, for w and v, with the side sets as vertices and letters as edges. Then it is checked whether the two graphs are related by an isomorphism that respects the order of edges. The isomorphism test is easily done in polynomial time.

The second phase of the procedure computes, for every cycle in the graphs, the multiplication of edges along the cycle. The multiplication of two neighbouring edges is defined as the equivariant mapping:

$$\{(a,a'),(b,b')\}\{(b,b'),(c,c')\} \quad \mapsto \quad \{(a,a'),(c,c')\}.$$

(This defines a mapping from $A \times A$ to A.) For a fixed vertex on a cycle, say $\{a,a'\}$, there are two possible results of multiplication along the cycle:

$$\{(a,a'),(a,a')\} \quad \text{or} \quad \{(a,a'),(a',a)\}$$

which defines two types of cycles. The procedure checks if every pair of corresponding cycles in the two graphs has the same type. This ensures that the two input words w and v are in the same orbit, i.e. there is a permutation of atoms that maps w to v. Indeed, in any connected component choose any pair of corresponding vertices, that is two side sets, and any of two possible bijections between the two side sets. Then extend this bijection along edges. As all cycles have the same types, the extension will never produce a conflict.

Appendix E
Appendix to Section VII

In this part of the appendix, consider sets with bit-vector atoms. We show that the language

$$D \stackrel{\text{def}}{=} \{a_1 \cdots a_n \in \mathbb{A} : a_1, \ldots, a_n \text{ are linearly dependent}\}.$$

is not recognised by a deterministic polynomial time Turing machine. To simplify the notation, we only show a slightly weaker result: the language D is not recognised by an *equivariant* deterministic polynomial time Turing machine. (A machine is equivariant if all of its ingredients are equivariant: the work alphabet, the states, the initial state, the final states and the transition function.) For the rest of the proof, we only consider bit-vector atoms, and therefore we use the term atom to refer to bit-vectors.

The proof has two steps.

1) The input alphabet is the set of atoms, which is a straight set. One could imagine a Turing machine that quickly recognises dependence by using a non-straight work alphabet or state space. The first step shows that this is not the case: every deterministic machine can be made straight (i.e. have a straight work alphabet and state space), without affecting the running time.
2) In the second step, we show that a straight deterministic machine needs exponential time to reject a sequence of linearly independent vectors[4].

Step 1: Reduction to straight machines: The reduction to straight machines breaks down into two further substeps. We first show that the bit vector atoms have a property called *least closed supports*. Then, we show that when the atoms have least closed supports, then every deterministic machine over a straight input alphabet can be made straight without affecting the running time.

The *closure* of a set S of atoms is the set of all atoms that are supported by S. Of course the closure contains S, because every atom is supported by itself. A set of atoms is called *closed* if it is its own closure.

Lemma E.1. *For every set with atoms X, there is a finite closed support that is contained in all finite closed supports of X.*

Proof
We first show that there exists a finite closed support. Indeed: take any finite support, and use its closure. This closure is finite, because the closure of a set S of atoms is the linear space spanned by S, i.e. the set of vectors of the form

$$\sum_{v \in T} v \qquad \text{for some } T \subseteq S.$$

We now show that there exists a least closed support with respect to inclusion. The reason is that closed supports are closed under intersection, as stated in the following lemma.

Lemma E.2. *If an element x of a set with atoms is supported by closed sets S and by T, then it is also supported by $S \cap T$.*

□

To complete step 1, we will show the following result.

[4]For every length of the input, there is actually only one rejecting run, up to automorphism. A rejected input is a tuple of independent vectors. All independent n-tuples of vectors are in the same equivariant orbit. If the transition function is equivariant, then all runs over independent n-tuples of vectors are in the same orbit. In particular all of these runs have the same behaviour of the head (when it moves left or right, and when it produces a zero vectors).

Lemma E.3. *Every equivariant deterministic machine M over a straight alphabet can be transformed into an equivariant straight deterministic machine M', with the same running times.*

To prove the above lemma, we show that least closed supports (as in Lemma E.1) allow us to represent non-straight sets in a canonical way by straight sets. A *canonical straight representation* for an equivariant set Y is defined to be a surjective function $r : X \to Y$ such that X is a straight set, and the function preserves and reflects supports: for every $x \in X$, both x and $r(x)$ have the same supports.

Lemma E.4. *Every equivariant set has a canonical straight representation.*

Proof
It suffices to prove the lemma for single-orbit equivariant sets. Let X be a single-orbit equivariant set. Choose some $x \in X$, and let $\{a_1, \ldots, a_n\}$ be the least closed support of x, as postulated by Lemma E.1. The function mapping $(a_1, \ldots, a_n) \mapsto x$ extends uniquely to an equivariant function

$$r : \mathrm{Aut} \cdot (a_1, \ldots, a_n) \to \mathrm{Aut} \cdot x = X.$$

The function r is a canonical straight represenation, because both $(a_1, \ldots, a_n)$ and x have the same least closed support, and this extends to the whole orbit by equivariance. □

The following lemma shows that an equivariant function can be lifted to canonical straight representations.

Lemma E.5. *Let $r_1 : Y_1 \to X_1$ and $r_2 : Y_2 \to X_2$ be canonical straight representations, and let $f : X_1 \to X_2$ be an equivariant function. There is an equivariant function $g : X_1 \to X_2$ which makes the following diagram commute*

$$\begin{tikzcd} X_1 \arrow[r, "g"] \arrow[d, "r_1"] & X_2 \arrow[d, "r_2"] \\ Y_1 \arrow[r, "f"] & Y_2 \end{tikzcd}$$

Proof
We define the function g separately for each orbit of X_1. Choose some x_1 in some orbit of X_1. Because r_2 is surjective, there must be some $x_2 \in X_1$ with

$$r_2(x_2) = f(r_1(x_1)).$$

Because r_2 is a canonical straight representation, x_2 has the same support as the $f(r_1(x_1))$. Because r_1 is preserves supports, and f is equivariant, it follows that the least closed support of $f(r_1(x_1))$ is included in the least closed support of x_1. Summing up, the least support of x_2 is included in the least closed support of x_1. Therefore, the mapping $x_1 \mapsto x_2$ can be extended to an equivariant function on the orbit of x_1. We do the same for the other orbits. □

Proof (of Lemma E.3)
By applying Lemma E.4 to the work alphabet and the state space, and then applying Lemma E.5 to the transition function. □

Step 2.: In this step, we show that deterministic straight machines need exponential time to recognise if vectors are linearly independent, as stated in the following lemma.

Lemma E.6. *When run on independent vectors $a_1 \cdots a_n$, an equivariant straight deterministic machine for the language D must make exponentially many steps.*

The rest of this section is devoted to proving the above lemma. Fix $a_1, \ldots, a_n$ as in the statement of the lemma. We begin with the following sublemma.

Lemma E.7. *Let $n \in \mathbb{N}$. For every $I \subseteq \{1, \ldots, n\}$ there is a set $b_1, \ldots, b_n$ of vectors such that*

$$\sum_{i \in J} b_i = 0 \qquad \textit{iff} \qquad J = \emptyset \textit{ or } J = I$$

Proof (of Lemma E.7)
When I is empty, the lemma is immediate.

Otherwise, we can assume without loss of generality that $n \in I$. Choose some independent vectors $b_1, \ldots, b_{n-1}$. Define

$$b_n \stackrel{\text{def}}{=} \sum_{i \in I - \{n\}} b_i.$$

This means that

$$\sum_{i \in I} b_i = b_n + b_n = 0.$$

We now show that

$$\sum_{i \in J} b_i \neq 0$$

holds whenever J is nonempty and not equal to I. If J contains some $j \notin I$ then the sum cannot be zero, because b_j is linearly independent from all the remaining vectors. Otherwise, J is a proper subset of I. If J does not contain n, then the sum cannot be zero, because the vectors with indexes in $I - \{n\}$ are linearly independent. Finally if J contains n but omits some $i \in I$, then the sum from the statement of the lemma will be nonzero on coordinate b_i. □

Proof (of Lemma E.6)

Consider a run of deterministic normal form machine on an input $a_1 \cdots a_n$ which consists of independent vectors. Since the vectors are independent, the machine should reject.

As usual, a computation can be visualised as a rectangular grid, where each tile of the grid gets a colour which consists of a symbol of the work alphabet and a possibly a state (if the tile coincides with the head). The set of colors is an orbit-finite set. Since the machine is straight, the set of colors is also straight, i.e. each color is a tuple of atoms. We say an atom appears in a color if it is found in one of the coordinates of the tuple of atoms that is the color.

We say that an atom is *locally supported* by a computation if it is supported by the atoms that appear in the colours of two neighbouring tiles (neighboring either vertically or horizontally). The number of atoms that can appear in the colours of two tiles is bounded by the Turing machine. The number of colours supported by these atoms is therefore also bounded by the machine, if exponentially bigger. This exponential depends only on the fixed Turing machine and not its input, and therefore can be treated as a constant with respect to the input. It follows that the number of atoms that are locally supported by a computation is quadratic in the length of the run, assuming that the machine is fixed.

The transition function of the machine is equivariant, so everything it produces has smaller or equal support to its arguments. It follows that every atom which appears in the computation, or is locally supported by it, is already supported by the input, and therefore each of these atoms is a linear combination of the input vectors. Since the computation is not exponentially long, then some linear combination must be missing, i.e. there must be some nonempty $I \subseteq \{1, \dots, n\}$ such that the atom

$$\sum_{i \in I} a_i$$

is not two-supported by the computation.

Apply Lemma E.7, to the set I, yielding a tuple of linearly dependent atoms $b_1, \dots, b_n$. We will show that this input is also rejected by the machine, which contradicts the assumption that the machine accepts all dependent tuples. Consider the computations of the machine on inputs $a_1 \cdots a_n$ and $b_1 \cdots b_n$, encoded as coloured grids, call them ρ and σ.

Claim E.7.1. *For every two tiles x, y the pairs of colours*

$$(\rho(x), \rho(y)) \qquad (\sigma(x), \sigma(y))$$

are in the same equivariant orbit.

Proof
By induction on the distance of the tiles from the top row of the grid, which contains the input. □

By taking $x = y$, the claim implies that for every individual tile, the colors in both grids are in the same equivariant orbit. Whether or not a state is accepting is an equivariant property, and therefore one computation contains an accepting state if and only if the other computation contains an accepting state. □