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.**

***Keywords*-Sets with atoms; Turing machines;**

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.

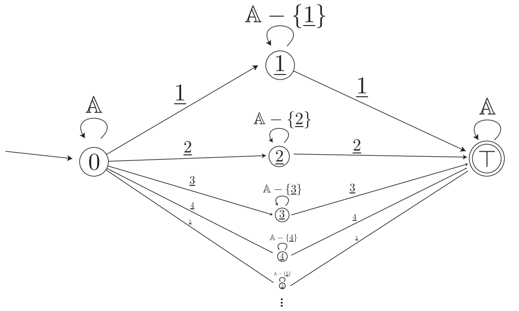
More complex examples are used in the following sections. Note that each shape of letters comes with an obvious action of bijective atom renaming. Sometimes this action is trivial even if the renaming is not; for example, the permutation that swaps $\underline{3}$ and $\underline{5}$ does not alter the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

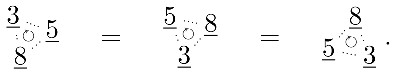
The language L can also be recognised by a deterministic Turing machine that, for each letter in the input word, stores it in the state and checks whether it appears more than once. An extended work alphabet is needed for this to allow marking of the currently processed letter; more importantly, a deterministic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

The above is quite similar to how classical Turing machines recognise 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:

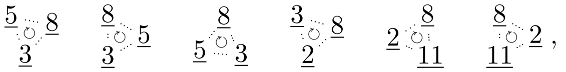
$$\{(a,b,c),(b,c,a),(c,a,b)\} \text{ for } a,b,c \in \mathbb{A} \text{ distinct.} \qquad (1)$$

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

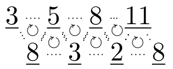


(2)

Consider the language of those sequences of such triples:



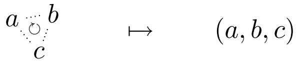
where every two consecutive triples share at least two atoms, that can be glued together in a matching chain like this:



(3)

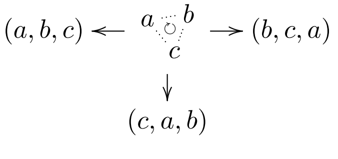
To recognise this language, a Turing machine first nondeterministically "freezes" the leftmost triangle in some position (or, equivalently, chooses an atom from it that shall not touch the second triangle in the chain), and then progresses to the right deterministically, checking that each subsequent letter can be affixed to the emerging chain.

One might think that a deterministic Turing machine can recognise 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 alter the triangle, but it does change the resulting triple. Intuitively, a function is unable to distinguish one of the three possible outcomes, as it can only access equality tests on atoms; "freezing" a rotating triangle 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 recognised 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 generalise: 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, 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. An 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. (Gaussian 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 [11], 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. $\{\{\underline{3}\},\{\underline{3},\underline{5}\}\}$),

(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 $\mathcal{P}_{\mathrm{fs}}\mathbb{A}$ 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. [11, 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.

A relation or 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 $\mathcal{P}_{\mathrm{fs}}\mathbb{A}$ 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[1]). 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 recognised 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 from $\underline{5}$, 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.

[1] 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 $\underline{1}$.

– 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 $\emptyset$-orbits: one orbit for the atoms, and one singleton orbit for the blank symbol.

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

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

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:

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

The set above also has one $\{\underline{5}\}$-orbit, since every pair of distinct atoms that are both different from $\underline{5}$ 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:

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

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

$$(\mathit{return}, \mathit{blank}) \to (\mathit{init}, \mathit{blank}, 1),$$

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

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

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 $\overset{a}{\triangle}_{b}$ 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:

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

(recall (2) to see that this defines three outgoing transitions for any input letter). The machine continues deterministically according to transitions:

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

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

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

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.

Even in the presence of atoms, the number of letters in a 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. [13], [14]. 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.

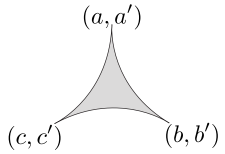
The rest of Section III is devoted to proving Theorem III.1. In Section III-A, we define a language L that witnesses the difference between NP and deterministic decidability, and we show that L is in NP. Then we prove that L is not deterministically semi-decidable. The proof is in two steps. In Section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

A. The language

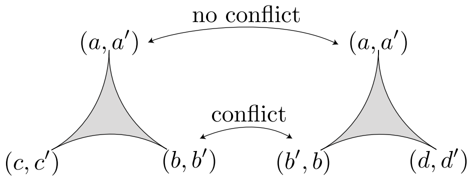
The input alphabet. We begin by defining the input alphabet. We use the name *triangle* for an unordered triple of ordered pairs of atoms, where all six atoms are distinct. In other words, a triangle is a set

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

We define the *side sets* of the triangle to be the unordered pairs $\{a, a'\}$, $\{b, b'\}$ and $\{c, c'\}$. 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 semi-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 recognising 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.

Our inspiration for the language L came from yet another source: it is a generalisation 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.

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

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

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 finite 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 $[\mathcal{T}']_\approx = [\mathcal{T}]_\approx$.

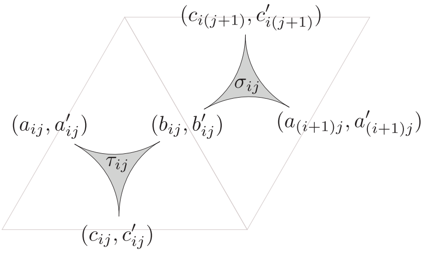
Torus. To construct an input that will confuse an algebra, we will place triangles in a torus-like arrangement. Let $n \in \mathbb{N}$. An n-torus is a set of $2n^2$ $\approx$-triangles defined as follows. Let us begin with $6n^2$ distinct atoms

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

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.

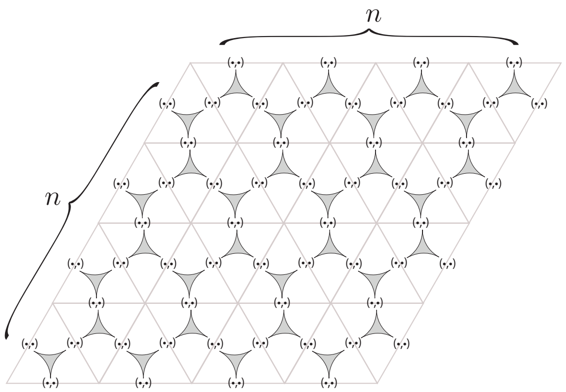


Fig. 1. An n-torus.

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 $\mathrm{val}\colon \mathit{variables}(t) \to \mathcal{T}$ 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.

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 recognises L, and consider terms t_n recognising 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_n 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 $t[\mathrm{val}]$. By induction on the size of the term t, one can show that

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

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 $t[\mathrm{val}]$. As a result, flipping τ' has the same effect on $t[\mathrm{val}]$ 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

$$\mathit{encode} : X \to 2^*,$$

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. Note that in (1) we could have used any other Turing-complete mechanism without atoms, such as deterministic Turing machines.

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 subsets of A^*. It turns out that, as language recognisers, both 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. In particular, deterministic Turing machines can simulate guessing a bounded number of (fresh or not) 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.

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. To save space, 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 compute a linear ordering of atoms from 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 $X = \mathrm{supp}(b)$, where $\mathrm{supp}(b)$ is the least support of b; the mapping $b \mapsto \mathrm{supp}(b)$ 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 similar to that 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 recognisable 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 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 determinise. 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 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 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[2].

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 recognised 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

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

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 enough 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

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

letters has the same cardinality in both words.) This decision procedure generalises easily to any alphabet $\binom{\mathbb{A}}{k}$ for $k > 2$.

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 $(\mathbb{N}, =)$, 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 semi-decidable is equal to nondeterministic semi-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 distinguishing 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:

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

It is easy to see that the language D is recognised by a nondeterministic polynomial time Turing machine that guesses a linear combination of vectors witnessing dependence.

On the other hand, checking all linear combination is inevitable: every deterministic Turing machine will need an exponential number of computation steps to recognise the language D.

Acknowledgment

This work has been partially supported by the Polish MNiSW grant N N206 567840 and by the ERC Starting Grant *Sosna*.

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.