Imperative Programming in Sets with Atoms*

Mikołaj Bojańczyk and Szymon Toruńczyk

University of Warsaw, Warsaw, Poland

Abstract

We define an imperative programming language, which extends while programs with a type for storing atoms or hereditarily orbit-finite sets. To deal with an orbit-finite set, the language has a loop construction, which is executed in parallel for all elements of an orbit-finite set. We show examples of programs in this language, e.g. a program for minimising deterministic orbit-finite automata.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.4.1 Mathematical Logic

Keywords and phrases Nominal sets, sets with atoms, while programs

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2012.4

Introduction

This paper introduces a programming language that works with sets with atoms, which appear in the literature under various other names: Fraenkel-Mostowski models [2], nominal sets [7], sets with urelements [1], permutation models [9].

Sets with atoms are an extended notion of a set – such sets are allowed to contain "atoms". The existence of atoms is postulated as an axiom. The key role in the theory is played by permutations of atoms. For instance, if a, b, c, d are atoms, then the sets

$$\{a, \{a, b, c\}, \{a, c\}\} \qquad \{b, \{b, c, d\}, \{b, d\}\}$$

are equal up to permutation of atoms. In a more general setting, the atoms have some structure, and instead of permutations one talks about automorphisms of the atoms. Suppose for instance that the atoms are real numbers, equipped with the successor relation $x = y + 1$ and linear order $x < y$. Then the sets

$$\{-1, 0, 0.3\} \qquad \{5.2, 6.2, 6.12\}$$

are equal up to automorphism of the atoms, but the sets

$$\{0, 2\} \qquad \{5.3, 8.3\}$$

are not.

Here is the definition of sets with atoms. The definition is parametrized by a notion of atoms. The atoms are given as a relational structure, which induces a notion of automorphism. (One can also consider atoms with function symbols, but we do not do this here.) A *set with atoms* is any set that can contain atoms or other sets with atoms, in a well-founded way[1]. The key notion is the notion of a *legal* set of atoms, defined below. Suppose that

* Supported by ERC Starting Grant "Sosna".

[1] Formally speaking, sets with atoms are defined by induction on their rank, which is an ordinal number. Sets of a given rank can contain atoms and sets of lower rank.

32nd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012).
Editors: D. D'Souza, J. Radhakrishnan, and K. Telikepalli; pp. 4–15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

X is a set with atoms. If π is an automorphism of atoms, then π can be applied to X, by renaming all atoms that appear in X, and appear in elements of X, and so on. We say that a set S of atoms is a *support* of the set X if X is invariant under every automorphism of atoms which is the identity on S. (For instance, the set of all atoms is supported by the empty set, because every automorphism maps the set to itself.) 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.

Sets with atoms were introduced in set theory by Fraenkel in 1922. Fraenkel gave a set of axioms for set theory, call it Zermelo-Fraenkel with Atoms (ZFA), which is similar but not identical to the standard axioms of Zermelo-Fraenkel (ZF). One difference is that ZFA does not have the extensionality axiom: two objects (think of different atoms) might have the same elements, but not be equal. Fraenkel gave two models of ZFA: one which has the axiom of choice, and one which does not. The first model contains all sets with atoms, while the second model restricts to the legal ones. Legal sets with atoms were further developed by Mostowski, which is why they are sometimes called Fraenkel-Mostowski sets.

In this paper, we are exclusively interested in sets with atoms that are legal. Therefore, from now on all sets with atoms are assumed to be legal.

Sets with atoms (as remarked above, we implicitly restrict to legal ones) were rediscovered for the computer science community, by Gabbay and Pitts [7]. In this application area, atoms have no structure, and therefore automorphisms are arbitrary permutations of atoms. 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. Sets with atoms are now widely studied in the semantics community, under the name of nominal sets (the name is so chosen because atoms describe variables names).

Sets with atoms turn out to be a good framework for other applications in computer science. These other applications have roots in database theory, but touch other fields, such as verification or automata theory. The motivation in database theory is that atoms can be used as an abstraction for data values, which can appear in a relational database or in an XML document. Atoms can also be used to model sources of infinite data in other applications, such as software verification, where an atom can represent a pointer or the contents of an array cell.

Sets with atoms are a good abstraction for infinite systems because they have a different, more relaxed, notion of finiteness. A set with atoms is considered finite if it has finitely many elements, up to automorphisms of atoms. (The formal definition is given later in the paper.) We call such a set *orbit-finite*. Consider for example sets with atoms where the atoms have no structure, and therefore automorphisms are arbitrary permutations. The set of atoms itself is orbit-finite, actually has only one orbit, because every atom can be mapped to every other atom by a permutation. Likewise, the set of pairs of atoms has two elements up to permutation, namely (a, a) and (a, b) for $a \neq b$. Another example is the set of λ-terms which represents the identity, with variables being atoms:

$$\{\lambda a.a : a \text{ is an atom}\};$$

this set has one orbit. Yet another example concerns automata with registers for storing atoms, as introduced by Francez and Kaminski in [10]: up to permutation, there are finitely many configurations of every such automaton.

The language of sets with atoms is so robust that one can meaningfully restate all definitions and theorems of discrete mathematics replacing sets by (legal) sets with atoms and finite sets by orbit-finite sets, see [3, 5] for examples in automata theory. Some of the

restated theorems are true, some are not. Results that fail after adding atoms include all results which depend on the subset construction, such as determinisation of finite automata, or equivalence of two-way and one-way finite automata. Results that work after adding atoms include the Myhill-Nerode theorem, or the equivalence of pushdown automata with context free grammars (under certain assumptions on the structure of atoms, which will be described below).

The papers [3, 5] were concerned with generalisations of finite state machines, like finite monoids and finite automata. But what about general computer programs? Is there are notion of computability for sets with atoms? One attempt at answering this question was [4], which described a functional programming language equipped with types for storing orbit-finite sets. The present paper gives an alternative answer: an imperative programming language, called while programs with atoms[2].

What is the advantage of having a programming language which can handle sets with atoms, be it functional or imperative? Consider the following algorithmic tasks coming from automata theory:

1. Given a nondeterministic finite automaton, decide if it is nonempty.
2. Given a deterministic finite automaton, compute the minimal automaton.
3. Given a finite monoid, decide if it is aperiodic.
4. Given a context-free grammar, compute an equivalent pushdown automaton.

Each of these tasks can be studied in the presence of atoms. (Finite automata with atoms are defined in [5], finite monoids with atoms are defined in [3], context-free grammars and pushdown automata can be defined in the same spirit.) Without a computation model of some sort, it is not clear what it means that the above tasks are decidable. In the papers [3, 5], the computational model depended on coding: a finite automaton with atoms was encoded as a normal string over the alphabet $\{0, 1\}$, and then the remainder of the algorithm used a standard Turing machine. Such coding is not a satisfactory solution, since algorithms and correctness proofs that involve coding are tedious and error-prone.

The picture becomes much simpler when using a language that manipulates directly objects with atoms. The coding issues have to be dealt with only once; when designing the language and proving that its programs can be simulated by usual computers. What is also interesting, if the syntax of the programming language is based on a classical syntax without atoms (in this paper, we add atoms to while programs), then one can easily compare programs for the same task with and without atoms. For instance, in the four tasks described above, one can write a program with atoms that actually has the same code as the corresponding program without atoms, only the interpreter used to execute it has different semantics.

1 Orbit-finite sets with atoms

In this section, we define orbit-finite sets with atoms. The definition of an orbit-finite set can be stated for any notion of atoms (modelled as a relational structure). However, without additional assumptions on the atoms, the notion of orbit-finite set might not be well behaved, e.g. orbit-finite sets might not be closed under products or finitely supported subsets. An assumption that guarantees good behaviour of orbit-finite sets is called homogeneity, and is defined below.

[2] We believe that the two programming languages are equivalent, under a suitable encoding, but we do not prove this in the paper.

Homogeneous structures

Recall that the notion of a set with atoms is parametrized by a relational structure for the atoms. Examples of atoms are:

$(\mathbb{N}, =)$ natural numbers (or any countably infinite set) with equality

$(\mathbb{Q}, \leq)$ the rational numbers with their order

The two kinds of atoms listed above will be called, respectively, the *equality atoms* and the *total order atoms*. In this paper, we require the atoms to be a *homogeneous* relational structure, i.e. one which satisfies the following property:

> Any isomorphism between two finite substructures of the atoms extends to an automorphism of the atoms.

Moreover, we assume that the atoms are countable, and the vocabulary of the relational structure is finite. The programming language we describe in this paper will work with any atoms satisfying the conditions above, plus an additional decidability condition to be defined below[3].

The equality and total order atoms are countable, homogeneous and have a finite vocabulary. An example of a structure which is not homogenous is $(\mathbb{Z}, \leq)$, the set of integers with their order. Indeed, the subsets $\{1, 2\}$ and $\{1, 3\}$ induce isomorphic substructures of $(\mathbb{Z}, \leq)$; however, there is no automorphism of $\mathbb{Z}$ which maps 1 to 1 and 2 to 3. In the rest of the paper, we always assume that the atoms are a countable homogeneous structure.

An interesting example of a homogeneous structure is the *random (undirected) graph*, also called the Rado graph.

▶ Example 1 (Random graph). The universe of this structure – representing the vertices of the graph – is the set of natural numbers. There is one binary relation, representing the edges of the graph; it is symmetric and irreflexive. The edges are constructed as follows: independently for each pair of vertices v, w, with probability $1/2$ we declare that v and w are connected by an edge. One can prove that two graphs constructed as above are isomorphic with probability 1; which is why we talk about *the* random graph and not *some* random graph. Moreover, the random graph is homogeneous.

An important property of homogeneous structures is that finitely supported relations coincide with sets of atoms definable by quantifier-free formulas which can use constants from the atoms. More precisely, we have the following.

▶ Proposition 1. Assume that the atoms are a homogeneous structure over a finite vocabulary. Let S be a finite set of atoms, and R a set of n-tuples of atoms. Then, R is S-supported if and only if it is defined by a quantifier-free formula over the vocabulary of the atoms, extended by constant names for elements of S.

For a given S and n, there are finitely many quantifier-free formulas over n-variables which use the (finite) vocabulary and constants from S. From the above proposition it follows that there are finitely many S-supported sets of n-tuples of atoms, for any given S and n.

[3] With slightly more work, the programming language would also work for the more general notion of atoms that are *oligomorphic* or, equivalently, *ω-categorical*.

Hereditarily orbit-finite sets

The reason why we are interested in sets with atoms is that there is an interesting new notion of finiteness, which is described below. Recall that for each legal set X there is a finite support S, i.e. the set X is invariant under the action of automorphisms which are the identity over S. We call such automorphisms *S-automorphisms.* Stated equivalently, X is a union of *S-orbits*, i.e. equivalence classes of the following relation $\sim_S$:

$$x \sim_S y \qquad \text{if } \pi \cdot x = y \text{ for some } S\text{-automorphism } \pi.$$

We say that X is *orbit-finite* if the union is finite, i.e. X is union of finitely many S-orbits for some finite set of atoms S. One of the important properties of homogeneous atoms is that the definition of orbit-finiteness does not depend on the choice of support S:

▶ **Lemma 2.** *Let X be a (legal) set, which is supported by sets S and T. Then X has finitely many S-orbits if and only if it has finitely many T-orbits.*

Other advantages of homogeneous atoms include:

- For every $n \in \mathbb{N}$, every finitely supported subset of n-tuples of atoms is orbit-finite;
- Orbit-finite sets are closed under products and finitely supported subsets.

In our programming language, we deal with sets that are *hereditarily orbit-finite*, i.e. sets which are orbit-finite, whose elements are orbit-finite, and so on recursively until an atom or empty set is reached. (One can show that for every hereditarily orbit-finite the nesting of set brackets is a natural number, as opposed to other sets, where the nesting may be an ordinal number.) As we will show, such sets can be presented in a finite way, and manipulated using algorithms.

Decidable homogeneous structures

By the theorem of Fraïssé [6], a homogenous structure is determined uniquely (up to isomorphism) by its *age*, which is the class of structures that embed into it. To permit computation, we will require an effective representation of this family.

▶ **Definition 3** (Decidable homogeneous structure)**.** A homogeneous structure $\mathfrak{A}$ is called *decidable* if its vocabulary is finite and its age is decidable, i.e. one can decide whether a given finite structure embeds into $\mathfrak{A}$.

The equality atoms and the total order atoms are decidable: the age of the equality atoms is all finite structures over an empty vocabulary, while the age of the total order atoms is all finite total orders. By Fraïssé's theorem, an algorithm deciding the age defines uniquely (up to isomorphism) the structure of Atoms. It follows that there are, up to isomorphism, countably many decidable homogenous structures. By [8], there are uncountably many non-isomorphic homogeneous structures over the signature containing one binary symbol, so some of them are undecidable.

The point of considering hereditarily orbit-finite sets in decidable homogeneous structures is that they can be represented without atoms, and these representations can be manipulated by algorithms.

▶ **Theorem 4.** *Suppose that the atoms are a decidable homogeneous structure over a finite vocabulary. Then there are data structures for representing atoms and hereditarily orbit-finite sets, which admit the following operations:*

1. *Given atoms $a_1, \ldots, a_n$ and a n-ary relation R from the vocabulary of the atom structure, decide if $R(a_1, \ldots, a_n)$ holds;*
2. *Given hereditarily orbit-finite sets X, Y, decide if $X = Y$;*
3. *Given X, which is a hereditarily orbit-finite set or an atom, compute $\{X\}$;*
4. *Given hereditarily orbit-finite sets X, Y, compute $X \cup Y$;*
5. *Given a hereditarily orbit-finite set X, compute some finite support;* [4]
6. *Given a hereditarily orbit-finite set X and a finite set S of atoms, produce all S-orbits that intersect X;* [5]
7. *Decide if a hereditarily orbit-finite set X is empty. If it is nonempty, produce an element.*

For instance, under the equality atoms, an atom can be represented as a natural number, encoded as its binary representation. The total order atoms are rational numbers, so they can also be represented, and the order relation can be computed. The representation for hereditarily orbit-finite sets is more involved.

2 Imperative programming with atoms

In this section, we present the contribution of the paper, an imperative programming language with atoms. The language extends while programs with two types: one for storing atoms and one for storing hereditarily orbit-finite sets. To deal with an orbit-finite set, the language has a loop construction, which is executed in parallel for all elements of an orbit-finite set. We end the paper with examples of programs in this language, e.g. a program for minimising deterministic orbit-finite automata.

2.1 Definition of the imperative language

The language is called *while programs with atoms*. The definition of the language depends on the choice of atoms (but not too much). We assume that the atoms are a decidable homogeneous relational structure over a finite vocabulary, as in the assumptions of Theorem 4.

The datatype.

We only have two datatypes in our language: `atom` and `set`. A variable of type `atom` stores an atom or is *undefined*. A variable of type `set` stores a hereditarily orbit-finite set. To have a minimal language, we encode other types inside `set`, using standard set-theoretic encodings. For example, the boolean `true` is encoded by $\{\emptyset\}$, and the boolean false is encoded by $\emptyset$.

Syntax.

The language contains the following constructions:

[4] This finite support is represented as a list of atoms.

[5] The S-orbits that intersect X are given as a list of sets. We claim that this list is finite. Indeed, the set X has some support, say T. Without loss of generality, we may assume that $S \subseteq T$, because supports are closed under adding elements. By assumption that X is orbit-finite and by Lemma 2, X is a finite union of T-orbits. Since $S \subseteq T$, every S-orbit is a union of T-orbits. It follows that X intersects at most finitely many S-orbits.

- **Constants.** There are infinitely many constants of type `atom`: one constant for every atom. (These constants depend on the choice of atom structure, e.g. there will be different constants for the equality atoms and different constants for the total order atoms.) There are constants $\emptyset$ and `Atoms` of type `set`, representing the empty set and the set of all atoms.
- **Expressions.** Expressions (which have values in the type `atom` or `set`), can be built out of variables and constants, using the following operations:
  1. Variables and constants are expressions. We assume that the types of the variables are declared in a designated preamble to the program; variables of type `atom` are initially undefined, while variables of type `set` are initially set to $\emptyset$.
  2. For every symbol σ in the vocabulary of the atom structure, if σ has arity n and $\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n$ are expressions of type `atom`, then
  $$\sigma(\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n)$$
  is an expression which evaluates to `true` or `false` (such an expression is of type `set`). For instance, when the atom structure is the total order atoms, and `x` and `y` are variables of type atom, then $\mathtt{x} \leq \mathtt{y}$ is an expression (we write $\leq$ using infix style).
  3. Comparisons $\mathtt{e} \in \mathtt{f}$ and $\mathtt{e} = \mathtt{f}$, which evaluate to `true` or `false`. In these comparisons, `e` and `f` can be either of type `atom` or `set`.
  4. Union $\mathtt{e} \cup \mathtt{f}$, intersection $\mathtt{e} \cap \mathtt{f}$ and set difference $\mathtt{e} - \mathtt{f}$, for `e` and `f` of type `set`.
  5. A unary singleton operation which adds one set bracket $\{\mathtt{e}\}$. Here, `e` can be either of type `atom` or `set`.
  6. An operation which extracts the unique atom from a set:
  $$\mathtt{theunique}(\mathtt{e}) = \begin{cases} \mathtt{f} & \text{when } \mathtt{e} = \{\mathtt{f}\} \text{ for some } \mathtt{f} \text{ of type } \mathtt{atom} \\ \mathtt{undefined} & \text{otherwise.} \end{cases}$$
- Values of expressions can be assigned to variables using the instruction $\mathtt{x} := \mathtt{e}$, provided that the types match.
- **Programming constructions.**
  1. A conditional `if e then I else J`. If the value of expression `e` is `true` $\stackrel{\text{def}}{=} \{\emptyset\}$, then program `I` is executed, otherwise program `J` is executed.
  2. A while loop `while e do I`, which executes the program `I`, while the value of expression `e` is `true`.
  3. A parallel for loop `for x in X do I`. Here `X` is an expression of type `set` and `x` is a variable of either type. The general idea is that the instruction `I` is executed, in parallel, with one thread for every element `x` (of appropriate type) of the set `X`. The question is: how are the results of the threads combined? We answer this question in more detail below.

Semantics.

We now sketch a semantics (operational style) for the language. In a given program, a finite number of variables is used. A state of the program is a valuation ν which assigns atoms (or the undefined value) to variables of type `atom` and hereditarily orbit-finite sets to variables of type `set`. Essentially, a valuation is a (finite length) tuple containing atoms and hereditarily orbit-finite sets, and therefore the set of all valuations is a legal set with atoms (but not orbit-finite). A state of the program can be represented in a finite way using the data structures from Theorem 4.

The semantics of a program is a partial function, which maps one valuation to another valuation. (The function is partial, because for some valuations, the program might not terminate.) We will say that *executing* the program `P` on the valuation ν results in a valuation μ if the semantics of the program transforms the valuation ν to the valuation μ.

We only explain the semantics for programs of the form

```
for x in X do I,
```

the other semantics are defined in the standard way. Suppose that we want to execute the program on a valuation ν. Two cases need be considered: when `x` is a variable of type `set` or when `x` is variable of type `atom`. The set $\nu(\mathtt{X})$ might store elements of both types `set` and `atom`. We say that an element $x \in \nu(\mathtt{X})$ is *appropriate* if it matches the type of the variable `x`. We define the valuation resulting from executing the above instruction on the valuation ν as follows. For every appropriate $x \in \nu(\mathtt{X})$, we execute the instruction `I` on the valuation ν_x which is obtained from the valuation ν by putting value x in the variable `x`. If for some appropriate x the program `I` does not terminate, then the whole `for` program does not terminate. Otherwise, for each appropriate $x \in \nu(\mathtt{X})$, we get a valuation μ_x obtained from ν_x by executing `I`. We now want to aggregate the valuations μ_x into a single valuation μ, which will be the result of executing the `for` program. If `y` is a variable of type `atom`, then in order for $\mu(\mathtt{y})$ to be defined, we require that all valuations agree on the value of `y`:

$$\mu(\mathtt{y}) \stackrel{\text{def}}{=} \begin{cases} a & \text{if for all appropriate } x \in \nu(\mathtt{X}),\ \mu_x(\mathtt{y}) = a \\ \mathtt{undefined} & \text{otherwise} \end{cases} \tag{1}$$

Set variables are aggregated using set union, i.e. every variable `y` of type `set` gets set to

$$\mu(\mathtt{y}) = \bigcup_{\text{appropriate } x \in \nu(\mathtt{X})} \mu_x(\mathtt{y}). \tag{2}$$

The definition of the language is now complete.

Results

One can show that our semantics is well-defined, i.e. executing instructions of our programming language does not cause a set variable to be assigned a set that is not hereditarily orbit-finite. We also prove that the programs can be simulated without atoms, in the following way. Thanks to Theorem 4 the code of a program, as well as a valuation of the variables, can be represented in a finite way without atoms.

▶ **Theorem 5.** *There is a normal program (without) atoms* `P`*, which inputs:*

- *a while program with atoms* `I`*;*
- *a valuation ν of the variables that appear in* `I`*;*

represented using the data structures of Theorem 4, and does the following:

- *if* `I` *does not terminate when starting in ν, then also* `P` *does not terminate;*
- *if* `I` *terminates when starting in ν, reaching valuation μ, then also* `P` *terminates, and produces a representation of valuation μ.*

Since the representations do not use atoms, they can be seen as standard bit strings, i.e. words over the alphabet $\{0, 1\}$. Therefore `P` can be modelled as a Turing machine, which inputs two bit strings and outputs a single bit string (and possibly does not terminate). In the proof of the above theorem we use the properties of the representations which are listed in Theorem 4.

The rest of the paper is devoted to example programs.

2.2 Example programs

Before writing example programs, we introduce some syntactic sugar which makes programming easier.

Notational conventions

Like in Python, we use indentation to distinguish blocks in programs. We write `{x,y}` instead of `{{x}∪{y}}`. We extend the syntax with functions (with the usual semantics); the syntax of functions is illustrated on the Kuratowski pairing function

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

We write `(a,b)` instead of `pair(a,b)`. Here is the function which projects a Kuratowski pair of sets into its first coordinate, and returns $\emptyset$ if its argument is not a Kuratowski pair of sets. All the variables are assumed to be of type `set`.

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

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

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

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

▶ Example 2 (The diagonal). As a warmup, we write a program that produces a specific set, namely

$$\{(a, a) : a \in \text{Atoms}\}.$$

The following program calculates this set in variable `X`.

```
for x in Atoms do X := X ∪ {(x,x)}
```

The same effect would be achieved by the following program.

```
for x in Atoms do X := {(x,x)}
```

▶ Example 3 (Programs that use order on atoms). In the same spirit, we can produce sets that refer to some structure on the atoms.

Consider the total order atoms. Recall that there is an expression `x` $\leq$ `y` that says if the atom stored in variable `x` is smaller than the atom stored in variable `y`. For instance, the following program generates the growing triples of atoms.

```
for x in Atoms do
 for y in Atoms do
  for z in Atoms do
    if (x ≤ y) and (y ≤ z) then X := {(x,y,z)}
```

▶ Example 4. Consider the total order atoms. The following program produces in variable `Y` the family of all closed intervals.

```
for (x,y) in Atoms do
   for z in Atoms do
      if (x ≤ z) and (z ≤ y) then X := X ∪ {z}
   Y := Y ∪ {X}
```

Actually, for every atom structure and for every hereditarily orbit-finite set X there exists a program which produces the set X.

▶ Example 5 (Reachability). We write a program which inputs a binary relation `R` and a set of source elements `S`, and returns all elements reachable (in zero or more steps) from elements in `S` via the relation `R`. The program is written using `until`, which is implemented by `while` in the obvious way.

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

The program above is the standard one for reachability, without any modifications for the setting with atoms. Why is the program still correct in the presence of atoms?

Suppose that S is a finite set of atoms that supports both the relation `R` and the source set `S`. Let X be the set that contains `S` and every element that appears on either the first or second coordinate of a pair from `R`. The set X is easily seen to be supported by S and to have finitely many S-orbits. Let $X_1, X_2, \ldots, X_k$ denote the S-orbits of X. Therefore,

$$X = X_1 \cup X_2 \cup \cdots \cup X_k. \tag{3}$$

It is easy to see that after every iteration of the `repeat` loop, the values of both variables `New` and `Old` are subsets of X that are supported by S. Therefore the values of these variables are obtained by selecting some of the orbits listed in (3). In each iteration of the `repeat` we add some orbits, and therefore the loop can be iterated at most k times.

▶ Example 6 (Automaton emptiness). Following [5], we define an *orbit-finite nondeterministic automaton* the same way as a nondeterministic automaton, with the difference that all of the components (input alphabet, states, initial states, final states, transitions) are required to be hereditarily orbit-finite sets. Using reachability, it is straightforward to write an emptiness check for nondeterministic orbit-finite automata:

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

▶ Example 7 (Monoid aperiodicity). An *orbit-finite monoid* is a monoid where the carrier is a hereditarily orbit-finite set, and the graph of the monoid operation is a finitely supported. (It follows that the graph of the monoid operation is a hereditarily orbit-finite set, because hereditarily orbit-finite sets are closed under finitely supported subsets.) Such a monoid is called *aperiodic* if for every element m of the monoid, there is a natural number n such that

$$m^n = m^{n+1}. \tag{4}$$

In [3] it was shown that an orbit-finite monoid is aperiodic if and only if all of the languages it recognises are definable in first-order logic. The following program inputs a monoid (its carrier and the graph of the monoid operation) and returns true if and only if the monoid is aperiodic. The program simply tests the identity (4) for every element in the carrier.

```
function aperiodic (Carrier,Monop)
  for m in Carrier  do
    X:=∅
    new:=m
    repeat
       old:=new
       X:=X ∪ {old}
       new := Monop(old,m)
    until new ∈ X
    if new = old then ret := true else ret := false
  return ret
```

In the program above, the line `new := mult(old,m)` is actually syntactic sugar for a subroutine, which examines the graph of the multiplication operation `mult`, and finds the unique element `new` which satisfies `(old,m,new)` $\in$ `mult`.

In the program, the set `X` is used to collect consecutive powers $\mathtt{m}, \mathtt{m}^2, \mathtt{m}^3, \ldots$. To prove termination, one needs to show that this set is always finite, even if the monoid in question is not aperiodic. This is shown in [3].

Finally, the program relies on the particular encoding of the booleans: `true` is the nonempty set $\{\emptyset\}$, while `false` is $\emptyset$. If the for loop sets `ret` to `true` for some `m` in the carrier of the monoid, then the whole program will return `true`, since the aggregation operation is union, which behaves like $\vee$ for booleans.

▶ Example 8 (Automaton minimisation). The programs for automaton emptiness and monoid aperiodicity were for yes/no questions. We now present a program that transforms one automaton into another. An *orbit-finite deterministic automaton* is the special case of the nondeterministic one where there is one initial state, and the transition relation is a function. As shown in [5], such automata can be minimised. We describe the minimisation procedure using a program in our language, i.e. a function

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

which inputs an orbit-finite deterministic automaton and returns the minimal automaton[6]. We assume that all states are reachable, the non-reachable states can be discarded using

[6] This program is particularly interesting when comparing the imperative programming language from this paper with the functional programming language from [4]. In [4], we were unable to correctly type a program for minimisation.

the emptiness procedure described above. We will also assume that all states are sets (and not atoms), so all variables in the code below are declared as `set`. The code is a standard implementation of Moore's minimisation algorithm. The only point of writing it down here is that the reader can follow the code and see that it works with atoms.

In a first step, we compute in the variable `equiv` the equivalence relation, which identifies states that recognise the same languages.

```
for p in Q do
 for q in Q do
   for a in A do
      R := R ∪ {((delta(p,a),delta(q,a)),(p,q))}

base := (F × (Q-F)) ∪ ((Q-F) × F)
equiv := (Q × Q) - reach(R,base)
return ∅ = reach(R,q0) ∩ F
```

(The code above uses $\times$, which is implemented using `for`.) For the states of the minimal automaton, we need the equivalence classes of the relation `equiv`, which are produced by the following code.

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

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

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. Mikolaj Bojanczyk. Data monoids. In *STACS*, pages 105–116, 2011.
4. Mikolaj Bojanczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. Towards nominal computation. In *POPL*, pages 401–412, 2012.
5. Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. Automata with group actions. In *LICS*, pages 355–364, 2011.
6. Roland Fraïssé. *Theory of relations*, volume 145 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, revised edition, 2000. With an appendix by Norbert Sauer.
7. Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.
8. C. Ward Henson. A family of countable homogeneous graphs. *Pacific J. Math.*, 38:69–83, 1971.
9. Thomas Jech. *The Axiom of Choice*. North-Holland, 1973.
10. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.