LOIS:

Syntax and Semantics *

Eryk Kopczyński
University of Warsaw, Poland
erykk@mimuw.edu.pl

Szymon Toruńczyk
University of Warsaw, Poland
szymtor@mimuw.edu.pl

Abstract

We present the semantics of an imperative programming language called LOIS (Looping Over Infinite Sets), which allows iterating through certain infinite sets, in finite time. Our semantics intuitively correspond to execution of infinitely many threads in parallel. This allows to merge the power of abstract mathematical constructions into imperative programming. Infinite sets are internally represented using first order formulas over some underlying logical structure, and SMT solvers are employed to evaluate programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Sets with atoms, definable sets

1. Introduction

Modern imperative programming languages allow easy manipulation of arrays, lists, sets, trees etc. using `for` loops, mimicking the intuitive set-builder notation used in mathematical formulas such as

$$Y = \{2x + 1 \mid x \in X, x > 3\}.$$

The code below is an example of how this could be imitated in C++ using the range-based for loop (available since C++11).

```
set<int> Y;
for (int x : X)
  if (x>3) Y.insert(2*x+1);

if memberof(10,Y)
  cout << "10 is in Y";
```

Many programming languages, e.g. Python or Scala, support list comprehension, closely resembling set-builder notation. Such constructions are more general than set-builder notation, in that they allow executing instructions (with possible side effects) within the body of the loop. On the other hand, set-builder notation is more powerful, as it can be equally well applied to infinite sets, which is crucial in abstract mathematical reasonings. For instance, the set X in the mathematical formula above might be the set of all integers or real numbers, yielding a meaningful definition of Y. To the best of our knowledge, no existing imperative programming language combines both strengths, allowing to evaluate instructions (possibly, nested) similar to the one above, when X denotes an infinite set. Note that although lazy evaluation allows defining certain infinite objects, few tests can be performed on them, e.g., one cannot determine in finite time whether an infinite stream contains the element 10, by analogy to `memberof(10,Y)` test above (although see [16] for an extension of OCaml allowing certain effective tests).

In this paper, we present LOIS – a programming language manipulating infinite sets, in which the above code is executable, in finite time even for some infinite sets X. Below we give further illustrating examples of imperative mathematical constructions involving iteration over infinite sets, written in pseudocode which is very close to LOIS.

Example 1. For a region F of $\mathbb{R}^n$, a *ball packing* of F is an inclusion-maximal family of pairwise-disjoint unit balls contained in F. An example of a ball packing of the disk of radius $2 + \sqrt{5}$ in the plane is depicted below.

The pseudocode below implements a function which takes as argument a region F of $\mathbb{R}^n$, and returns the set of *all* ball packings of F – usually, an infinite set.

```
set packings(F) {
  set packings = ∅;
  set partial = {∅};

  for P in partial do {
    bool maximal = true;

    for x in F do
     if Ball(x,1)⊆F and Ball(x,1)∩union(P)=∅
     {
       maximal = false;
       partial +=(P∪{Ball(x,1)})
     };
    if (maximal)
      packings += P
  }
  return packings
}
```

The semantics of the code is as follows. The variable `packings` stores the packings found so far, and the variable `partial` stores partial (possibly not maximal) packings. In each iteration, consider a partial packing P. For all points $x \in F$, if $P \cup \{Ball(x, 1)\}$ is a partial packing, add it to `partial`. If no such point x exists,

* This work is supported by Poland's National Science Centre grant 2012/07/B/ST6/01497

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009876

then P is maximal, so add it to `packings`. Note that the sets `F` and `partial`, over which the loops iterate, are typically infinite.

Example 2. Take an infinite countable set of vertices $\mathbb{V}$, and for each pair of vertices, choose randomly whether there is an edge between them or not. It is well known that, up to isomorphism, with probability 1 we get the same graph, known as the random graph, the Rado graph, or the Erdős-Rényi graph [12, 24]. To make things more interesting, assign each vertex randomly to one of two parts and remove edges between pairs of vertices which are in the same part. We obtain the random bipartite graph $\mathcal{G} = \langle \mathbb{V}, E \rangle$.

Is this graph connected? We could check this using the function `is-connected` described in pseudocode below.

```
bool is_connected(V,E) {
  for v in V do
    if (reach(E,{v})!=V)
      return false;
  return true;
}
```

Here we use the function `reach`, implemented by the pseudocode below as the BFS algorithm for computing reachable nodes starting from a set of vertices I, in a directed graph with edges $E \subseteq V \times V$.

```
set reach(E,I) {
  set R=I;
  set S=∅;

  while (R≠S) {
    S=R;
    for e in E do
      for v in S do
        if (e.first=v)
          R += e.second;
  }
  return R;
}
```

Note that we wish to run this function on infinite graphs. Both examples above can be implemented in LOIS, which, in the briefest summary, allows performing **for** loops over infinite sets. Intuitively, such a loop executes all of its branches in parallel.

Example 3. This example is inspired by formal verification of infinite state systems. Below is a LOIS program constructing an automaton with infinitely many states, which for a given sequence over the alphabet $\Sigma = \mathbb{N} \cup \{\#\}$, computes the maximal sum of an infix not interrupted by $\#$. For example, for the input sequence $1, 3, 2, \#, 5, 6, 11, \#, 1, 2$ the automaton ends in state $(22, 3)$.

```
set Σ = ℕ∪{'#'};
set Q = ∅;
set I = {(0,0)};
set δ = ∅;

for m in ℕ do
  for n in ℕ do
    Q+= (m,n);

for (m,n) in Q do
  for x in Σ do
    if (x=='#')
      δ+=((m,n),x,(m,0))
    else
      δ+=((m,n),x,(max(m,n+x),n+x));

set E = ∅;

for (p,a,q) in δ do
  E += (p,q);

print(reach(E,I));
```

The statespace Q consists of all pairs (m, n) with $m, n \in \mathbb{N}$. There is one initial state, $(0, 0)$. The transition relation $\delta \subseteq Q \times \Sigma \times Q$ is such that reading $\#$ resets the second component of the state, and reading a letter $x \in \mathbb{N}$ increases the second component by x, and accumulates the maximal seen value in the first component. The tuple (Σ, Q, δ, I) is an automaton (without accepting states), with infinite-state space.

In the last three lines of the code above, we compute the set of reachable states of our automaton, using the function `reach` defined in Example 2.

The output is $\{(m, n) \mid m \in \mathbb{N}, n \in \mathbb{N}, m \geq n\}$. Whereas **for** loops should be interpreted as executing in parallel, **while** loops are executed sequentially. In particular, in a terminating program, they are executed finitely many times. In our case, the while loop of the `reach` function iterates three times, with R taking values $\{(0, 0)\}$, $\{(n, n) \mid n \in \mathbb{N}\}$, and $\{(m, n) \mid m, n \in \mathbb{N}, m \geq n\}$.

We specify a set of accepting states, e.g. $F = \{(m, 3) \mid m \in \mathbb{N}\}$, which can be constructed in LOIS, similarly to Q. Now $A = (\Sigma, Q, \delta, I, F)$ is a deterministic, infinite-state automaton accepting those sequences, in which the maximal sum of an infix uninterrupted by $\#$ is 3.

What is the minimal automaton equivalent to A? To find out, we can try to run the well-known *partition refinement* algorithm on A, implemented by the code below.

```
function minimize(Σ,Q,q0,F,δ)
{
  set E = ∅;
  for (p,q,a) in Q×Q×Σ do
    E+=((δ(p,a),δ(q,a)),(p,q));
  set S=(F×(Q-F))∪((Q-F)×F);
  set equiv=(Q×Q)-reach(S,E);

  set classes=∅;
  for q in Q do {
    set class = ∅;
    for p in Q do
      if ((p,q)∈equiv)
        class += p;
    classes += class
  }

  return classes
}
```

Since this only works for deterministic automata, here we treat δ as a function $Q \times \Sigma \to Q$ and q_0 is the unique initial state. In the first phase, we compute in the variable `equiv` the equivalence relation which identifies states that recognise the same languages, i.e., $(p, q) \in$ `equiv` iff for all words $w \in \Sigma^*$, reading w from the state p, ends in an accepting state iff it does from the state q. To compute `equiv` we use the function `reach` described earlier. In the second phase, we compute the equivalence classes of the relation `equiv` on Q, which are the states of the minimal automaton; the transitions can be computed similarly. For the automaton A described above, this returns a minimal automaton with 11 states. Note that the pseudocode in the `reach` and `minimize` functions implements the classical algorithms for finite automata. One of the benefits of LOIS is that it demonstrates that these algorithms are also meaningful and correct for some infinite-state systems. We note that these functions can be readily converted into a very similar, executable LOIS program [21] with no need of auxiliary data structures.

Contributions We introduce the concept and formal semantics of a programming language manipulating infinite sets which are definable in an underlying first order structure, and allowing to iterate over such sets in finite time. Actually, we describe two seman-

tics, LOIS^0 and LOIS. Whereas LOIS^0 has a simpler mathematical description, LOIS is closer to an implementation, can simulate LOIS^0, and is more flexible. We briefly describe how LOIS and LOIS^0 can be implemented by employing SMT (Satisfiability Modulo Theories) solvers.

We have actually implemented LOIS as a C++ library (see [20] and [21]) allowing the users to combine pseudoparallel computation with the full power of C++11. Furthermore, our C++ library allows programming constructs such as recursion, function calls, expressions with side effects, complex data structures, etc. Since these can be handled in the standard way, they have been omitted from this paper.

Applications LOIS is a high level programming language allowing manipulation of abstract mathematical objects. We believe that it may be useful in the design and implementation of algorithms involving abstract mathematical reasoning, as well as for for didactic and expository purposes.

Verification and research. We believe that the syntax and semantics of LOIS are useful for presenting high-level algorithms in research papers, and proving their correctness. In particular, we believe that LOIS might find many applications in verification of infinite state systems. As LOIS allows to work directly with infinite sets, many theoretical arguments used in the verification literature can be readily translated into executable LOIS programs, avoiding the burden of implementing symbolic data structures. Moreover, the same piece of code can implement various algorithms, depending on the chosen underlying structure.

Aiding education. Several programming languages are based on the idea of allowing the programmer to manipulate sets. The principal example here is SETL [25], and its variants, such as ISETL [13]. Whereas these languages only allow using finite sets, LOIS extends this idea by introducing infinite sets, which are definable in an underlying logical structure $\mathcal{A}$. ISETL has been proposed [13] as a tool for teaching discrete mathematics. We believe that by allowing the manipulation of infinite definable sets LOIS can be a good tool for teaching logic, set theory, and abstract mathematics.

Outline In the following section, Section 2, we describe the intuition underlying pseudoparallel semantics. Next, we introduce the formal definition of the *fully pseudoparallel semantics* of LOIS^0, in Section 3. The expressive power of LOIS^0, from the logical point of view, is studied in Section 4, whereas the effectiveness of the semantics of LOIS^0 is demonstrated in Section 5. In the following Section 6, we introduce a variant of the fully pseudoparallel semantics, called *hybrid pseudoparallel semantics*, which is close to the actual implementation of LOIS, discussed in Section 7. In Section 8 we mention many potential applications of LOIS to formal verification. Related work is discussed in Section 9.

2. Pseudoparallel Semantics – Intuition

We start with a brief intuitive introduction of the main ideas of the pseudoparallel semantics. On an intuitive level, the semantics of LOIS^0 can be explained simply by saying that the instruction **for** `(x in A)` executes (possibly infinitely) many threads – one per each element x of the iterated set `A` – perfectly synchronously. The term "thread" here does not refer to parallel computation, but to the intuition associated to it; therefore we call those *pseudoparallel* threads. The pseudoparallel thread corresponding to the element x of `A` is executed with the control variable `x` set to x. In the fully *pseudoparallel* semantics of LOIS^0, perfect synchronization means that two different threads execute corresponding instructions of the loop's body exactly at the same moment.

The above intuition suffices to understand the examples in the introduction, and to write programs in LOIS. We give some more examples below, using the actual syntax of LOIS^0 and LOIS. For technical reasons, our C++ library uses a slightly different syntax [21]. The formal syntax and semantics of LOIS^0 will be given in Section 3, and of LOIS in Section 6.

Let $\mathbb{A}$ denote an infinite set, and let `A` be a variable representing it. Consider evaluating the following pseudocode below.

```
set X;
for x in A do
  X += x;
```

The type `set` denotes a type for (possibly infinite) sets. The instruction `X += x` adds the element `x` to the set `X`.

Naturally, after executing this program, the variable `X` should evaluate to $\mathbb{A}$. However, this cannot be evaluated by sequential iteration, since the set $\mathbb{A}$ is infinite. Intuitively, we create a separate thread a for each $a \in \mathbb{A}$. The instruction `X += x`, performed in all threads in the same time, adds all the possible values $x \in \mathbb{A}$ to the set `X`. As expected, `X` evaluates to $\mathbb{A}$.

Now consider the following pseudocode.

```
set X;
for x in A do {
  set Y;
  for y in A do {
    if ¬(x=y)
      Y += y;
    X += Y;
  }
}
```

This program demonstrates several new concepts. First, the threads may branch: in the inner loop, there is a thread xy for each pair $x, y \in \mathbb{A}$. Second, threads can have local variables: each thread $x \in \mathbb{A}$ has its own copy of the variable `Y`. Next, notice the use of a conditional. The inner statement is executed only for those threads which satisfy the condition, while for the other threads, an empty instruction will be executed. In thread x, the final value of its local copy of the variable `Y` is $\{y : y \in \mathbb{A}, y \neq x\}$. The last line adds the value of each local variable `Y` to `X`. The final value of `X` is $\{\{y : y \in \mathbb{A}, y \neq x\} : x \in \mathbb{A}\}$.

The outcomes of the above two programs are intuitively clear, since their code resembles the set-builder notation used for describing the resulting sets. To define a meaningful semantics, however, some conceptual challenges need to be overcome when truly imperative constructions are executed within an infinite loop. Consider for instance the following pseudocode.

```
int parity = 0;
for x in A do
  parity = 1 - parity;
```

Since $\mathbb{A}$ is an infinite set, the parity of its cardinality is ill-defined. What should the outcome of the program be? According to the *fully* pseudoparallel semantics of LOIS^0, all the threads $x \in \mathbb{A}$ are fully synchronous, so the last instruction sets the value of `parity` to 1 in each thread simultaneously and unanimously, so the overall result is 1. On the other hand, in the *hybrid* semantics of LOIS which we also present in this paper, the full synchronicity condition is dropped, and the result of the above program can be either 0 or 1, depending on the internal representation of `A`. For example, when `A` in fact represents a finite set in a standard way (similar to a list), the code above will compute the parity of the set, as in this particular case the **for** loop is executed sequentially, rather than in parallel. In general, however, some threads can be executed in parallel, and some sequentially, hence the name *hybrid* semantics. The main advantage of the hybrid semantics of LOIS is that it tends to produce more succinct outputs, which in turn results in more efficient programs. The fully parallel semantics of LOIS^0, on the

$$\mathrm{S} ::= \mathcal{V}ar,\ \emptyset,\ \mathbb{A},\ \mathrm{S}\cup\mathrm{S},\ \mathrm{S}\cap\mathrm{S},\ \mathrm{S}-\mathrm{S},\ (\mathrm{S}),\ \{\mathrm{S}\},\ f(\mathrm{S},\ldots,\mathrm{S}),$$
$$\mathrm{B} ::= \bot, \top,\ \mathrm{B}\vee\mathrm{B},\ \mathrm{B}\wedge\mathrm{B},\ \neg\mathrm{B},\ (\mathrm{B}),$$
$$\mathrm{S}=\mathrm{S},\ \mathrm{S}\in\mathrm{S},\ \mathrm{S}\subseteq\mathrm{S},\ R(\mathrm{S},\ldots,\mathrm{S}).$$

Figure 1. Formal syntax of expressions of type **set** (S) and **bool** (B). Above, $\mathbb{A}$ denotes a name of a sort in $\mathcal{A}$ and f and R denote function/relation symbols in the signature of $\mathcal{A}$ of arity matching the number of arguments.

other hand, has a more elegant mathematical description. We begin by formally defining the fully pseudoparallel semantics.

3. LOIS^0: Fully Pseudoparallel Semantics

Although the semantics of LOIS^0 may seem natural on an intuitive level, it turns out that defining it formally is a nontrivial endeavour. In this section, we give a big-step operational semantics for LOIS^0, which we call the *fully pseudoparallel semantics*. This language has a fully pseudoparallel **for** loop, conditional (**if**), sequencing (;), while-loop (**while**), local variables (type **set**), expressions, assignment, and set insertion (`+=`). For simplicity, we omit recursion, and assume that expressions have no side effects, and do not allow variables of types other than **set**. Semantics for these can be given in the standard way. Although this is not necessary for understanding the semantics, it is crucial in our applications to allow LOIS^0 to access an infinite logical structure, as described below.

Underlying structure As we want programs in LOIS^0 to manipulate complex mathematical objects, such as the rationals, the reals, infinite random graphs, etc., it will be useful to allow LOIS^0 to access an underlying logical structure $\mathcal{A}$. For instance, in order to implement Example 1 from the Introduction, one would take $\mathcal{A}$ to be the ordered field of reals $\mathcal{R} = \langle \mathbb{R}, +, \cdot, 0, 1, \leq \rangle$, which is rich enough to talk about certain regions in the plane (such as squares, disks), sets of regions, ball packings, etc. During the execution of the program, new infinite sets can be produced basing on the underlying structure $\mathcal{A}$. For instance, as we will see, the disk of radius $2+\sqrt{5}$ centered at the origin of the plane, considered in Example 1, can be constructed when the underlying structure is $\mathcal{R}$.

Fix an underlying structure $\mathcal{A}$, which is a many-sorted structure $\mathcal{A}$ over a signature containing relational and functional symbols. We refer to the literature (e.g. [15]) for structures, variables, sorts, terms, and formulas. Throughout this paper, formulas are assumed to be first order. Each sort of $\mathcal{A}$ has a *name* which is a symbol, and a *domain*, which is a set. We use the boldface letters $\mathbb{A}, \mathbb{B}$, etc. to denote sort names. An element of $\mathcal{A}$ is an element of any of its domains. Formally, elements of $\mathcal{A}$ are sets (as in set theory, every mathematical object is a set). We call elements of $\mathcal{A}$ *atoms* (this terminology comes from set theory) as according to our semantics (see Section 3.2), in programs they will be viewed as sets which do not contain any elements.

3.1 Syntax

Expressions Fix a set of variable names $\mathcal{V}ar$. We consider expressions of two types: **set** and **bool**, whose syntax is presented in Figure 1. Expressions of type **set** denote sets (which include atoms, i.e., elements of $\mathcal{A}$). Expressions of type **bool** are used as conditions for flow control operations.

Statements We skip the formal syntax of LOIS^0 statements, as it is very similar as for while-programs. Below is an informal description. A statement is either of the following:

- A sequence statement of the form $S_1; S_2$, where S_1 and S_2 are statements;
- A variable declaration instruction of the form **set** $x; S$ where $x \in \mathcal{V}ar$ is a variable name. This declares a local variable x in S. We consider ; to be right associative, thus the scope of x extends to the end of the block;
- flow-control instructions **if** (e) S and **while** (e) S, where e is an expression of type **bool** and S is a statement;
- A statement **for** $(x$ **in** $e)$ S, where $x \in \mathcal{V}ar$ is a variable name and e is an expression of type **set**, and S is a statement;
- An assignment instruction `x=y` or insertion instruction `x+=y`, where x is a variable name and y is an expression of type **set**.

3.2 Semantics

Let $\mathbb{U}$ be the universe of set theory, i.e., its elements are arbitrary sets (we will see in Section 4 that in fact not all sets are necessary). A *thread* is formally a sequence $\gamma \in \mathbb{U}^*$ of elements of $\mathbb{U}$, where $\mathbb{U}^*$ denotes the class of tuples (finite sequences) of elements of $\mathbb{U}$. If $w, v \in \mathbb{U}^*$, then wv is the concatenation of the tuples w, v. A state of the program is a partial function $s : \mathcal{V}ar \times \mathbb{U}^* \to \mathbb{U}$. Informally, $s(v, \gamma)$ is interpreted as the value of the variable v in thread γ (which can be undefined).

Expressions For an expression e, thread γ and state s, we write $[\![e]\!]_{s,\gamma}$ for the denotational semantics of the expression e in the given state and thread, as defined formally in Figure 2 below. We use the convention that a set operation $\cup, \cap, -$ can be applied only to sets which are not atoms (i.e., elements of $\mathcal{A}$), and function and relation symbols from $\mathcal{A}$ can only be applied to elements of $\mathcal{A}$ of appropriate sorts (and not to sets). Violating these natural restrictions results in undefined semantics. In an interpreter, this would be raised as a runtime error.

$$[\![x]\!] = s(x,\eta), \quad \text{where } \eta \text{ is the longest prefix of } \gamma \text{ for which } s(x,\eta) \text{ is defined}$$
$$[\![\emptyset]\!] = \emptyset \qquad [\![\mathbb{A}]\!] = \mathbb{A}^{\mathcal{A}} \qquad [\![\{e\}]\!] = \{[\![e]\!]\}$$
$$[\![e \# f]\!] = [\![e]\!] \# [\![f]\!] \quad \text{for } \# \in \{\cup, \cap, -\} \text{ if } [\![e]\!] \notin \mathcal{A} \text{ and } [\![f]\!] \notin \mathcal{A}$$
$$[\![f(e_1,\ldots,e_n)]\!] = f^{\mathcal{A}}([\![e_1]\!],\ldots,[\![e_n]\!]) \quad \text{if } [\![e_i]\!] \in \mathbb{A}_i^{\mathcal{A}} \text{ for } 1 \leq i \leq n$$
$$[\![R(e_1,\ldots,e_n)]\!] = R^{\mathcal{A}}([\![e_1]\!],\ldots,[\![e_n]\!]) \quad \text{if } [\![e_i]\!] \in \mathbb{A}_i^{\mathcal{A}} \text{ for } 1 \leq i \leq n$$
$$[\![e = f]\!] = \begin{cases} \top & \text{if } [\![e]\!] = [\![f]\!] \\ \bot & \text{otherwise} \end{cases} \qquad [\![e \in f]\!] = \begin{cases} \top & \text{if } [\![f]\!] \notin \mathcal{A}, [\![e]\!] \in [\![f]\!] \\ \bot & \text{otherwise} \end{cases}$$
$$[\![e \subseteq f]\!] = \begin{cases} \top & \text{if } [\![e]\!], [\![f]\!] \notin \mathcal{A}, [\![e]\!] \subseteq [\![f]\!] \\ \bot & \text{otherwise} \end{cases}$$

Figure 2. Semantics of expressions. Fix a state s and a thread γ; we write simply $[\![e]\!]$ to denote $[\![e]\!]_{s,\gamma}$. Above, $x \in \mathcal{V}ar$, and $\mathbb{A}$ is a sort name whose corresponding domain in $\mathcal{A}$ is $\mathbb{A}^{\mathcal{A}}$; f is a function symbol in the signature of $\mathcal{A}$ of type $\mathbb{A}_1 \times \cdots \times \mathbb{A}_n \to \mathbb{A}_0$ whose interpretation in $\mathcal{A}$ is $f^{\mathcal{A}}$, and R is a relation symbol in the signature of $\mathcal{A}$ interpreted as a function $R^{\mathcal{A}}$ of type $\mathbb{A}_1 \times \cdots \times \mathbb{A}_n \to \{\top, \bot\}$. The semantics of boolean expressions is defined in the obvious way for $\bot, \top, \vee, \wedge, \neg$.

Statements The rules of semantics of LOIS^0 statements will be given as $\Gamma, S, s \to t$, indicating that if the current set of threads is Γ, and we execute the statement S, the state s changes to t. In Figure 3, we list the rules of the semantics of LOIS^0 statements.

Note that according to the definition, the assignment $x = e$ cannot be applied when there are two threads η_1, η_2 in Γ extending a thread γ in which the variable x is defined, and which write different values to x, i.e., $[\![e]\!]_{s,\eta_1} \neq [\![e]\!]_{s,\eta_2}$. Formally, in this case, the semantics of such a program is undefined. In an interpreter, this would be raised as a runtime error.

Typically, when considering the semantics of a program P in LOIS^0, we start with the set of threads Γ_0 consisting only of the

$$\frac{\Gamma = \emptyset}{\Gamma, S, s \to s}\ (\textit{no-threads}) \qquad \frac{\Gamma, S_1, s \to t \qquad \Gamma, S_2, t \to u}{\Gamma, S_1;\ S_2, s \to u}\ (\textit{sequencing})$$

$$\frac{\Gamma, S, s * [(x,\gamma) \mapsto \emptyset \text{ for } \gamma \in \Gamma] \to t}{\Gamma, \textbf{set}\ x; S, s \to t \setminus x}\ (\textit{declaration})$$

$$\frac{\{\gamma \in \Gamma : [\![e]\!]_{s,\gamma} = \top\}, S, s \to t}{\Gamma, \textbf{if}\ (e)\ S, s \to t}\ (\textit{if}) \qquad \frac{\forall \gamma \in \Gamma. [\![e]\!]_{s,\gamma} = \bot}{\Gamma, \textbf{while}\ (e)\ S, s \to s}\ (\textit{while-done})$$

$$\frac{\{\gamma \in \Gamma : [\![e]\!]_{s,\gamma} = \top\}, S, s \to t \qquad \Gamma, \textbf{while}\ (e)\ S, t \to u}{\Gamma, \textbf{while}\ (e)\ S, s \to u}\ (\textit{while})$$

$$\frac{\Delta, S, s * [(x,\gamma g) \mapsto g \text{ for } \gamma g \in \Delta] \to t,\ \Delta = \{\gamma g : \gamma \in \Gamma, g \in [\![e]\!]_{s,\gamma}\}}{\Gamma, \textbf{for}\ (x\ \textbf{in}\ e)\ S, s \to t \setminus x}\ (\textit{for})$$

$$\frac{t = [(X,\gamma) \mapsto (s(X,\gamma) \cup \{[\![e]\!]_{s,\eta} : \eta \in \Gamma, \gamma \preccurlyeq \eta\}) \text{ for } \gamma \in \mathrm{dom}(s(X,-))]}{\Gamma, X \texttt{+=} e, s \to s * t}\ (\textit{insertion})$$

$$\frac{t = [(x,\gamma) \mapsto [\![e]\!]_{s,\eta} \text{ for } \gamma \in \mathrm{dom}(s(x,-)), \eta \in \Gamma, \gamma \preccurlyeq \eta]}{\Gamma, x \texttt{=} e, s \to s * t}\ (\textit{assignment})$$

Figure 3. Semantics of LOIS^0. For two threads γ, η, write $\gamma \preccurlyeq \eta$ if γ is a prefix of η, that is, η is one of the sub-branches of the thread γ. For a state t and variable x, let $t \setminus x$ denote t where values for (x, γ) become undefined for all γ. By $s * t$ we denote the partial mapping which maps (y, γ) to $t(y, \gamma)$ if this is defined, and to $s(y, \gamma)$ otherwise. By $[d]$ we denote the partial mapping as described by the description d. If d does not define a partial mapping, then $[d]$ is undefined (this can happen only in the assignment rule). Except for the no-threads rule, we assume $\Gamma \neq \emptyset$.

empty tuple $\varepsilon \in \mathbb{U}^*$, indicating that no branching has occurred yet, and with the state $s_0 : \mathcal{V}ar \times \mathbb{U}^* \to \mathbb{U}$ consisting of the empty partial mapping. We say that the program P *results in* state s if one can derive $\Gamma_0, s_0, P \to s$ using the rules of the semantics of LOIS^0. Of course, there might be no such state s, for example if the program P loops forever or a runtime error occurs. However, a program P results in at most one state, since the semantics of LOIS^0 is deterministic. Another issue, discussed in Section 5, is whether the resulting state s can be computed, given P – this may depend on the underlying structure $\mathcal{A}$.

Example 4. We showcase our semantics on a simple example. Consider the following statement, described earlier in Section 2.

```
set X;
for (x in Y)
  X += x;
```

We execute this statement in some initial set of threads Γ, and some initial state s, such that $s(\mathtt{Y}, \gamma) = \{a, b\}$ for all $\gamma \in \Gamma$, where a, b are some two distinct sets. In other words, in every thread in Γ, the variable `Y` evaluates to the set $Y = \{a, b\}$. According to the declaration rule, when the variable X is declared, a pseudoparallel copy for each thread $\gamma \in \Gamma$ is created, where the value of the copy in thread γ is given by $s(x, \gamma)$. Then, we are performing a pseudoparallel **for** loop, hence in the body of the loop, the set of threads Γ changes to $\Delta = \Gamma \cdot Y$: each thread $\gamma \in \Gamma$ branches into two threads $\gamma a, \gamma b \in \Delta$, and $s(x, \gamma i) = i$ for $i = a,\ b$. In other words, in the thread γi, the control variable x has value i. Finally, according to the insertion rule, $s(x, \gamma a) = a$ and $s(x, \gamma b) = b$ are inserted to $s(X, \gamma)$ in parallel, so the final value of X in every thread $\gamma \in \Gamma$ is $\{a, b\}$. A similar analysis would be valid if Y were infinite rather than finite.

Suppose now that we modify the last line of the code to the assignment `X=x`. At the moment of assignment, $s(x, \gamma a)$ and $s(x, \gamma b)$ are different, so the assignment rule cannot be applied (when trying to apply this rule, the description inside the brackets $[\,]$ does not define a partial mapping, since it maps (x, γ) to two different values). Intuitively, two different assignments to `x` are performed at exactly the same moment, resulting in a runtime error.

Finally, if we modify the last line of the code to the statement **if** `(x=a) X=x` (assuming `a` evaluates to the element a of Y in every thread $\gamma \in \Gamma$, i.e., $s(\mathtt{a}, \gamma) = a$ for $\gamma \in \Gamma$) then the condition `x=a` is satisfied only in the thread γa, so the program would be valid again, and the final value of X would be a, in every thread $\gamma \in \Gamma$.

4. Definable Sets

We now turn to characterizing the sets which can be computed by a program in LOIS^0, with underlying structure $\mathcal{A}$. Our characterization will shed light on how to represent these sets internally in an interpreter, and how to evaluate LOIS^0 programs. We will see that these sets are precisely those sets which can be constructed using set-builder notation with first-order guards ranging over elements of $\mathcal{A}$ and finite unions, in a nested fashion. We start with defining precisely this class of sets.

We assume an infinite set of variables, each of which is assigned to a specific sort (these are *first-order* variables, not to be confused with the *programmatic* variables in $\mathcal{V}ar$). If x is assigned to a sort named $\mathbb{A}$, we write $x \in \mathbb{A}$. A *context* is a set of either variables, called *bound variables*, or formulas in the signature of $\mathcal{A}$, called the *constraints*. A *valuation* of a set of variables V is a mapping $v : V \to \mathcal{A}$ which respects the sorts. If C is a context, then by $\mathrm{Val}(C)$ we denote the set of valuations of the bound variables in C which satisfy every constraint in C.

We define *set-builder expressions*, *variable elements* (or *v-elements*) and *variable sets* (or *v-sets*) by mutual induction. A *set-builder expression* is an expression of the form $\{e \mid C\}$, where e is a v-element and C is a context. *V-sets* are formal unions of set-builder expressions of the form $\{e_1 \mid C_1\} \cup \ldots \cup \{e_k \mid C_k\}$, also denoted $\{e_1 \mid C_1; \ldots; e_k \mid C_k\}$ for brevity. A *v-element* is either of the following: a term built of variables and function symbols from $\mathcal{A}$, a tuple of v-elements, an integer, or a v-set. If e is a v-set or v-element and v is a valuation of its free variables, then by $e[v]$ we denote its value under this valuation (formally defined by structural induction on e), which is a mathematical object (i.e., $e[v]$ is an atom in $\mathcal{A}$ or is a set).

Definable sets Fix a structure $\mathcal{A}$. Let e be a v-element. If $a_1, \ldots, a_n$ are elements of $\mathcal{A}$ and v is a valuation of its free variables with values in $\{a_1, \ldots, a_n\}$, then we say that the element $e[v]$ is *definable* (in $\mathcal{A}$) with parameters $a_1, \ldots, a_n$.

Example 5. Let $\mathcal{Q} = \langle \mathbb{Q}, < \rangle$ be the rationals with their linear order. The expression e equal to $\{x \mid x \in \mathbb{Q}, x < y\} \cup \{z\}$ is a v-set with free variables y, z. Under the valuation $v : \{y, z\} \to \mathbb{Q}$ such that $v(y) = 2$ and $v(z) = 3$, $e[v]$ evaluates to the set $X = \{x \mid x \in \mathbb{Q}, x < 2\} \cup \{3\}$. The set X is definable, with parameters $2, 3$. The definable subsets of $\mathcal{Q}$ are precisely the finite unions of open (possibly half-bounded) intervals and points.

Now consider the *ordered field of reals* $\mathcal{R} = \langle \mathbb{R}, +, \cdot, 0, 1, \leq \rangle$. An example definable subset of $\mathbb{R}^3$ is the half-ball $\{(x, y, z) \mid x \in \mathbb{R}, y \in \mathbb{R}, z \in \mathbb{R}, x > 0 \wedge x^2 + y^2 + z^2 \leq 1\}$. Another example is the disk of radius $2 + \sqrt{5}$ centered at $(0, 0)$, from Example 1, which is defined by the expression $\{(x, y) \mid x \in \mathbb{R}, y \in \mathbb{R}, r \in \mathbb{R}, r^2 - 4r - 1 = 0 \wedge r > 0 \wedge x^2 + y^2 \leq r\}$. A celebrated result of Tarski characterizes definable subsets of $\mathbb{R}^k$ as precisely the finite

unions of sets defined by systems of equalities and inequalities between k-variate polynomials.

To cast definable sets in the realm of set theory, we may use the standard set-theoretic encodings of tuples and integers, using sets. In the definitions above, we allow tuples and integers for notational convenience, and in programs, for efficiency.

4.1 Closure Properties

We argue that the class of definable sets, besides having a simple mathematical definition, also enjoys many useful closure properties, which makes it suited for performing mathematical constructions. We will later use these properties to prove in Theorem 4 that LOIS^0 computes precisely all definable sets, and to prove in Theorem 10 that the semantics of LOIS^0 is effective.

As in set theory, we may speak of *definable relations* (definable subsets of $X \times Y$), *definable functions* (definable relations which are functional), *definable structures* (structures, in which the domain, the relations and functions are definable), etc.

Proposition 1. *Definable sets are effectively closed under boolean combinations, cartesian products, quotients under definable equivalence relations and images under definable mappings.*

The proof of the following result proceeds by recursively reducing set equality to inclusion, inclusion to membership, and membership to equality.

Proposition 2. *Let e, f be two v-elements with free variables contained in V. There exists an effectively computable (in* PSPACE*) formula $\tau_=$ such that for any valuation $v : V \to \mathcal{A}$, the equality $e[v] = f[v]$ holds iff v satisfies $\tau_=$ in $\mathcal{A}$. Similarly, there are formulas $\tau_\in, \tau_\subseteq$ for $\in, \subseteq$ in place of $=$.*

Below, by boolean expression we mean an expression as described by the syntax given in Section 3, and $[\![e]\!]$ denotes the denotational semantics of e according to the semantics given in Section 3, with $[\![v_i]\!] = x_i$, for $i = 1, \ldots, n$.

Proposition 3. *Let $X \subseteq X_1 \times \cdots \times X_n$ be a definable set. If e is a boolean expression with free variables $v_1, \ldots, v_n$, then the set $W = \{(x_1, \ldots, x_n) \in X : [\![e]\!] = \top\}$ is definable.*

Proof. By definition, e is a boolean combination of atomic predicates of the form $=, \in$ or R (where R is a relation symbol from $\mathcal{A}$) applied to terms obtained from variable names using functions symbols from $\mathcal{A}$. Since definable sets are closed under boolean combinations, we only need to consider expressions which are atomic predicates. Consider first the case when e is the relation $=$ applied to terms, i.e., e is of the form $s = t$, where s, t are terms using function symbols from the vocabulary of $\mathcal{A}$ and variables $x_1, \ldots, x_n$. In particular, s and t are v-elements.

Since X is a definable set, it is a finite union of sets described by a set-builder expression. We consider the special case when X is described by a single set-builder expression $\{(e_1, \ldots, e_n) \mid C\}$; the general case follows from this special case easily by taking unions. Apply Proposition 2 to the v-elements s, t, yielding a formula $\tau_=$. Then $W = \{(x_1, \ldots, x_n) \mid [\![s]\!] = [\![t]\!]\}$ is described by the set-builder expression $\{(x_1, \ldots, x_n) \mid C, \tau_=\}$. This finishes the case when e is of the form $s = t$. The case when e is of the form $s \in t$ is similar.

Finally, consider the case when e is of the form $R(s_1, \ldots, s_k)$, where R is a relation symbol of $\mathcal{A}$. In order for $R(x_1, \ldots, x_k)$ to hold, it must be the case that $x_1, \ldots, x_k$ are elements of $\mathcal{A}$, of appropriate sorts. Therefore, it is enough to consider the case when X is described by a set-builder expression of the form $\{(e_1, \ldots, e_n) \mid C\}$, where e_i is a term built of variables and function symbols from $\mathcal{A}$ (for set-builder expressions of the remaining forms, the resulting set W is empty). Then, the set W is described by the set-builder expression $\{(e_1, \ldots, e_n) \mid C, R(e_1, \ldots, e_n)\}$. □

The above propositions demonstrate that the setting of definable sets is suitable for performing basic mathematical constructions. Finally, let us remark that structures which are definable in $\mathcal{A}$ correspond (up to isomorphism) to structures which *interpret* in $\mathcal{A}$ via interpretations (see e.g. [15] for a definition).

4.2 LOIS^0 Computes Definable Sets

The following theorem shows that the values computed by LOIS^0 are exactly the definable sets. We say that a state partial function $s : \mathcal{V}ar \times \mathbb{U}^* \to \mathbb{U}$ *is definable* if its graph is definable (where $\mathcal{V}ar$ is identified with the set of naturals $\{0, 1, 2, \ldots\}$, and $\mathbb{U}^*$ is the class of tuples of elements of $\mathbb{U}$).

Theorem 4. *Fix an underlying structure $\mathcal{A}$. For a LOIS^0 program P, if P results in a state partial function s, then s is definable without parameters. Conversely, for every set X which is definable without parameters there is a LOIS^0 program P which results in a state s such that $X = s(x, \varepsilon)$ (where x is a variable).*

The second part of Theorem 4 proceeds by constructing a LOIS^0 program simulating the expression defining the set X, basically, by replacing set-builder notation by **for** loops with additional conditions. We remark that if X is definable with parameters, i.e., $X = e[v]$ for some v-element e and valuation v of the free variables V of e, then the set $\{e[v] : v : V \to \mathcal{A}\}$ is definable without parameters and contains X, and can be constructed in LOIS^0 according to the theorem above.

The first part is straightforward from the following proposition:

Proposition 5. *Let Γ and s be definable, and P be a statement. If $\Gamma, P, s \to t$, then t is definable too.*

Let Γ, s and P be as in the proposition. The following lemma is useful in a formal proof of the proposition, and follows from the fact that an expression defining Γ can only involve tuples of bounded length.

Lemma 6. *There is a bound $l \in \mathbb{N}$ such that every tuple in Γ has length at most l. Moreover, for each $0 \leq k \leq l$, the set $\Gamma_k = \Gamma \cap \mathbb{U}^k$ of tuples in Γ of length k is definable.*

Proof of Proposition 5. The proof proceeds by induction on the structure of the statement P. In the inductive step, we study each rule given in Figure 3.

Let us consider first the case when P is of the form **if**(e) S. In this case, the expression defining the thread sets is of the form $\Delta = \{\gamma \in \Gamma : [\![e]\!]_{s,\gamma} = \top\}$, which is definable by Lemma 7 below.

Lemma 7. *The set $\Delta \subseteq \mathbb{U}^*$ is definable.*

By applying the inductive hypothesis to S and $\Delta, S, s \to t$, we conclude that t is definable, finishing the considered case.

Proof of Lemma 7. This follows general principles of manipulating definable sets, and their good closure properties, similarly to Proposition 1, and is shown below.

Let l and Γ_k (for $k = 0..l$) be as in Lemma 6. Then $\Delta = \Delta_0 \cup \cdots \cup \Delta_l$, where $\Delta_k = \{\gamma \in \Gamma_k : [\![e]\!]_{s,\gamma} = \top\}$. It suffices to show that each Δ_k is definable, for $k = 0..l$.

Let

$$X = \{(s(x_1, \gamma), s(x_2, \gamma), \ldots, s(x_r, \gamma)) \mid \gamma \in \Gamma_k\}, \qquad (1)$$

where $x_1, x_2, \ldots, x_r \in \mathcal{V}ar$ are all the variables appearing in the expression e. We claim that X is a definable set, since both s and Γ_k are definable.

Let $G(s) \subseteq \mathcal{V}ar \times \mathbb{U}^* \times \mathbb{U}$ be the graph of s; by definition, $G(s)$ is a definable set. Then $G(s)$ can be written as a union

$$G(s) = G_0(s) \cup \cdots G_l(s),$$

where $G_k(s) = G(s) \cap (\{0, \ldots, l\} \times \mathbb{U}^k \times \mathbb{U})$ is the graph of the restriction of s to $\{0, \ldots, l\} \cap \Gamma_k$. Observe that since the length of tuples in $G_k(s)$ is fixed (and equal to $k+2$), the projection P_i of $G_k(s)$ onto a coordinate i, where $i = 1..k+2$ is a definable set.

Then, for $i = 1..r$, the set

$$X_i = \{s(x_i, \gamma) \mid \gamma \in \Gamma_k\}$$

is definable by Proposition 1, as the projection onto the $k+2$-nd coordinate of the set $G_k(s)$. Similarly, one can define an expression defining the set X in (1). Applying Proposition 3 to $X \subseteq X_1 \times \cdots \times X_r$, we obtain that Δ_k is definable. □

We now consider the case when P is of the form **for** $(x$ **in** $e)$ S. Let $\Delta = \Gamma \cdot [\![e]\!]_{s,\gamma}$ consist of all tuples $(u_1, \ldots, u_n, u)$ such that $(u_1, \ldots, u_n) \in \Gamma$ and $u \in [\![e]\!]_{s,\gamma}$.

Lemma 8. *The set $\Delta = \Gamma \cdot [\![e]\!]_{s,\gamma}$ is definable.*

Consider the partial function $r = [(x, \gamma g) \mapsto g \text{ for } \gamma g \in \Delta]$, where x is fixed. It is easy to see that r is a definable state function, and that consequently, $s' = s * r$ is also a definable state function. By applying the inductive hypothesis to S and $\Delta, S, s' \to t$, we conclude that t is definable, too.

In the case of the remaining rules, we proceed similarly. This ends the inductive proof of Proposition 5, finishing the proof of Theorem 4. □

Theorem 4 says that in principle, in order to implement an interpreter of LOIS^0, one could use definable sets as an internal representation for the type **set**. Note that the procedure described in Proposition 5 is not effective – although exactly one of the *while* and *while-done* rules can be applied, to determine which one should be applied, we need to test validity of the sentence $\forall \gamma \in \Gamma. [\![e]\!]_{s,\gamma} = \bot$. A similar problem arises when we want to determine whether to apply the *no-threads* rule or one of the remaining rules, as this requires testing nonemptiness of a definable set. Indeed, a certain effectiveness assumption concerning the underlying structure $\mathcal{A}$ is required in order to guarantee computability, as we describe in Section 5.

5. Effectiveness

Whereas Section 3 provides a formal semantics of LOIS^0, in this section, we show that the semantics is effective, i.e., the programs can be simulated by classical computers. This requires some mild assumptions on the underlying structure $\mathcal{A}$, namely, that its theory is decidable. Let us first recall some standard notions from model theory.

Theories A *theory* T is a set of sentences (formulas without free variables) over a fixed signature. *The theory* of the structure $\mathcal{A}$ is the set of all sentences which hold in the structure. We say that a theory T is *decidable* if there is an algorithm which decides whether a given sentence belongs to T. Such an algorithm is called a (first order theory) *solver* for T, or an *SMT solver* (for *Satisfiability Modulo Theory*) with background theory T. There is a rich literature concerning SMT solvers (see e.g. [4] for an overview), with many working implementations. Structures with decidable theories include $\langle \mathbb{N}, \leq, + \rangle, \langle \mathbb{Q}, \leq, + \rangle, \langle \mathbb{R}, +, \cdot, \leq \rangle$, the Rado graph (see Example 2). On the other hand, the theory of $\langle \mathbb{N}, +, \cdot \rangle$ is undecidable, by Gödel's theorem.

The following proposition shows that decidability of the theory of $\mathcal{A}$ is necessary for effective evaluation of LOIS^0 programs. Theorem 10 below also shows that it is a sufficient condition.

Proposition 9. *Fix an underlying structure $\mathcal{A}$ with at least two elements. The problem of deciding emptiness of a given definable set is computationally equivalent to deciding the theory of $\mathcal{A}$, and is* PSPACE*-hard.*

Proof sketch. By Proposition 2, deciding emptiness reduces (in PSPACE) to testing satisfiability of a first order formula in $\mathcal{A}$. In the other direction, a sentence φ is satisfiable in $\mathcal{A}$ iff the definable set $\{\emptyset \mid \varphi\}$ is nonempty. PSPACE-hardness follows from the fact that QBF reduces to the theory of $\mathcal{A}$, by interpreting quantifiers as introducing first-order variables ranging over $\mathcal{A}$ rather than propositional variables ranging over $\{\bot, \top\}$, and by replacing each propositional variable x in the body of the QBF formula by the predicate $x = x_0$, where x_0 is an additional, existentially quantified variable. For example, the QBF formula $\forall x \exists y. x \vee (y \to x)$ is translated to the formula $\exists x_0 \forall x \exists y. (x = x_0) \vee ((y = x_0) \to (x = x_0))$. □

Theorem 10. *Suppose that the underlying structure $\mathcal{A}$ has a decidable theory. Given a program P one can effectively compute the resulting state s, if it exists.*

We remark that runtime errors (as described in Section 3) can be also effectively detected; the only case when a resulting state cannot be computed is when a while-loop never terminates.

Proof. We observe that in the proof of Proposition 5, we can effectively determine which rule to apply. To distinguish between applying the *while* and *while-done* rules, we test for emptiness of the (effectively) definable set $\{\gamma \in \Gamma : [\![e]\!]_{s,\gamma} = \top\}$. To distinguish between applying the *no-thread* rule and the remaining rules, we test for emptiness of the definable set Γ. By Proposition 9, emptiness of definable sets reduces to testing validity of a first-order formula in the underlying structure $\mathcal{A}$. □

Theorem 10 allows evaluating LOIS^0 programs. However, this approach might not be practical: thread sets Γ and state functions s may become very complicated. This is the main reason why we introduce the hybrid semantics in Section 6.

6. LOIS: Hybrid Pseudoparallel Semantics

In this section, we define the hybrid pseudoparallel semantics of LOIS. The syntax of LOIS is the same as of LOIS^0, and the only difference lays in the semantics. Although LOIS lacks the mathematical elegance of LOIS^0 in some aspects, it leads to more efficient and simpler implementations. In fact, we have implemented a prototype of LOIS (see [20] and [21]). On the other hand, it is easy to simulate[1] LOIS^0 in LOIS. We remark that the same code can yield different results in LOIS and LOIS^0, due to the hybrid pseudoparallel semantics – see the example mentioned in Section 2, computing the parity of the set[2]. Other differences between LOIS and LOIS^0 are summarized in Section 6.1.

As mentioned, mathematically LOIS is less appealing than LOIS^0. For better readability, and also to give a hint of some of the implementation details, we omit the rule-based definition as in Section 3, and rather give a definition using words, which, while precise, closely follows the actual implementation. To illustrate

[1] Our implementation extends LOIS by an operator fpp, guaranteeing the same outcome as in LOIS^0.

[2] To make this example precise, we could use as underlying structure $\mathcal{A} = \langle \mathbb{N}, 0, 1, - \rangle$ (where $-$ denotes subtraction), or extend the syntax and semantics of LOIS and LOIS^0 to allow manipulation of integers.

the formal definitions, we use a running example. We assume an underlying structure $\mathcal{A}$, for which a theory solver is provided.

The stack LOIS uses a stack for storing contexts (recall from Section 4 that contexts comprise bound variables and constraints). The stack is initially empty and is controlled by the **if**, **while** and **for** constructs. In a given moment of the execution of a program, the *current stack context*, denoted *Current*, is the context defined as the union of all contexts currently on the stack[3]. It is guaranteed that an instruction is executed if and only if Val(*Current*) is nonempty, i.e. there is a valuation which satisfies the constraints currently on stack.

One should keep in mind the following relationship with the semantics of LOIS^0: the set of all satisfying valuations of the current stack context corresponds to the set of threads Γ. In other words, *Current* can be seen as a description of Γ, using formulas. However, whereas threads in LOIS^0 can be indexed by arbitrary sequences of (definable) sets, in LOIS, they are of a specific normal form, namely they are indexed by tuples of elements of $\mathcal{A}$.

Types Variables of type **set** designate v-sets. V-sets are implemented as queues consisting of set-builder expressions; this is relevant for how such expressions are processed (see paragraph *Insertion* below). Initially, v-sets are empty. Variables of type **set** have an associated *inner context*, which at the moment of creation of the variable is set to the current stack context, and is never changed during the life of the variable. The inner context contains all free variables of the v-set or formula designated by the variable. It is an invariant that at any moment of a variable's lifetime, the current stack context contains the variable's inner context.

Example 6 (Running example)**.** To illustrate the definitions, we study a simple running example, by executing the code below. Suppose that `X` is a variable of type **set**, storing the v-set $X = \{a \mid a \in \mathbb{A}, a \neq b\} \cup \{(a,b,c) \mid c \in \mathbb{A}, a = b, b \neq c\}$, and that the current stack context is (for some reason) equal to $C = \{a \in \mathbb{A}, b \in \mathbb{A}, a \neq b\}$.

```
set Y;
for (x in X)
  Y += x;

if (X = Y) S1;
if (¬(X = Y)) S2
```

S_1 and S_2 denote statements, which we imagine to report the answer in some way (if we allowed side effects, as in the actual implementation of LOIS, they could print something on the screen). Initially, the inner context of Y is equal to C, and Y designates the empty v-set $\emptyset$. □

For loop An instruction of the form `for (x in X) I` is executed as follows. Sequentially loop over all set-builder expressions forming the v-set X. For such an expression $\{e \mid D\}$, rename consistently all bound variables in D using fresh (unused) variables, yielding an equivalent expression $\{e' \mid D'\}$. Push D' onto the stack, and set the loop's control variable `x` to the v-element e'. Check whether Val(*Current*) is nonempty, by invoking the solver for the theory of $\mathcal{A}$. If so, execute the loop's body. Remove D' from the stack, and proceed to the next expression comprising X.

(Running example) In the first iteration, the first expression comprising X, i.e., $\{a \mid a \in \mathbb{A}, a \neq b\}$ becomes $\{d \mid d \in \mathbb{A}, d \neq b\}$. This is desired, since the bound variable a is already used in the current stack context C. The context $\{d \in \mathbb{A}, d \neq b\}$ is pushed onto the stack, so the current stack context becomes $\{a \in \mathbb{A}, b \in \mathbb{A}, a \neq b, d \in \mathbb{A}, d \neq b\}$. This context has a satisfying valuation, so the instruction `Y += x` is executed with `x` set to d. □

Insertion The insertion `Y += x` is executed as follows: append the set-builder expression $\{x \mid D\}$ to the queue associated to `Y`, where D is the set difference between the current stack context and the inner context of `Y` (bear in mind that the latter is always a subset of the former).

(Running example) The difference between the current stack context and inner context C of `Y` is $\{d \in \mathbb{A}, d \neq b\}$, and `x` is set to d. Therefore the instruction `Y += x` results in adding $\{d \mid d \in \mathbb{A}, d \neq b\}$ to (previously empty) `Y`. Afterwards, the stack context reverts to C.

In the next iteration, the expression $\{(a,b,c) \mid c \in \mathbb{A}, a = b, b \neq c\}$ is renamed to $\{(a,b,e) \mid e \in \mathbb{A}, a = b, b \neq e\}$ (in this case, the renaming is not essential), and the stack context becomes $\{a \in \mathbb{A}, b \in \mathbb{A}, e \in \mathbb{A}, a \neq b, a = b, b \neq e\}$, which has no satisfying assignment. Therefore, no instruction is performed, and the stack context reverts to its original value C. The final value of `Y` is $Y = \{d \mid d \in \mathbb{A}, d \neq b\}$. Observe how the final value of `Y` differs from the value of `X`, even though the performed instruction seems to simply copy the content of `X` to `Y`. This should not be disturbing, since the v-sets X and Y are equivalent in the context C, i.e., yield the same result under every valuation in Val(C). In fact, the obtained expression Y can be seen as a simplification of the expression X, in the context C. □

Tests and conditionals The tests (`x`$\in$`y`, `x`=`y`, `x`$\subseteq$`y`) result in producing a formula φ, as described in Proposition 2. The conditional **if** and the while loop **while** are executed as follows. Push a context D consisting of the condition φ onto the stack, and check (using a solver) whether the resulting stack context *Current* has a satisfying valuation, i.e., if Val(*Current*) is nonempty. If so, execute the body of the instruction. Finally, remove D from the stack, and – in the case of **while** – repeat.

(Running example) The test `X = Y` results in a formula ϕ such that $\phi(v)$ holds if and only if $X[v] = Y[v]$, for any valuation v. In particular, since $X[v] = Y[v]$ for every valuation v satisfying the current stack context C, it follows that every valuation satisfying C also satisfies ϕ. Executing the first **if** conditional results in pushing the formula ϕ onto the stack. The resulting stack context $C \cup \{\phi\}$ has a satisfying assignment, namely, any assignment satisfying C. Therefore, the statement S_1 is executed. When evaluating the second conditional, however, the resulting stack context is equal to $C \cup \{\neg\phi\}$, which does not have any satisfying assignment, so the statement S_2 is not executed. □

This ends the definition of the operational semantics of LOIS.

6.1 Differences Between LOIS and LOIS^0

Below is an overview of the main differences between the two semantics.

- LOIS^0 is defined as an abstract programming language which works on $\mathbb{U}$, which is a model of set theory. On the other hand, LOIS works on v-sets (expressions which define definable sets). As we know from Theorem 4, this is not a drawback, since any set computable by LOIS^0 is definable. On the other hand, using finite representations is necessary for implementation.
- In LOIS, it is possible that looping over v-sets X_1 and X_2 gives different results, even though they represent the same set $X \in \mathbb{U}$. This is because, while in LOIS^0 the **for** $(x$ **in** $X)$ loop always executes fully in parallel, in LOIS, it executes sequentially if X is defined as a union of set-builder expressions. This

[3] In the SMT literature [3], our contexts correspond (roughly) to *assertion-sets*, our stack – to the *assertion-set stack*, and the current stack context – to the *set of all assertions.*

should not be seen as a drawback, however: to simulate the fully pseudoparallel semantics of LOIS^0 in our implementation of LOIS, it is sufficient to write **for** $(x$ **in** $\mathrm{fpp}(X))$, where fpp is an operator which changes the representation of X so that all threads will be executed fully pseudoparallelly.

- The set of current threads Γ of LOIS^0 is represented by a context in LOIS, which only uses variables ranging through $\mathbb{A}$, and constraints which are first order formulae using these variables. This is much simpler than LOIS^0, where the threads were indexed by sequences of arbitrarily complex (definable) elements of $\mathbb{U}$. We believe that this is elegant in its own right.
- Looping over set-builder expressions sequentially can be viewed as a low-level tool. This is useful when we want LOIS to interact with things which do not work in pseudoparallel, for example, to present the results on the screen. Furthermore, an implementation of LOIS^0 would require us to implement set equality, inclusion, and membership. In case of LOIS, we essentially get this for free – set equality, inclusion, and membership can be implemented in LOIS itself, simply by looping over all the elements of the set. For this reason, we believe that LOIS is easier to implement than LOIS^0.
- Such an access to low-level representations improves the efficiency. Moreover, our experiments show that programs written in LOIS yield nicer representations than LOIS^0 (obtained by forcing parallel computation with the fpp operator), which furthermore improves both the efficiency and the presentation of the end results.
- In a LOIS loop **for** $(x$ **in** $X)$, it is possible to add new elements to X while it is running. Such new elements will be processed after all the old elements are processed (in general, set-builder expressions that X consists of will be processed in first-in first-out order). This turns out to be useful in practice, since many of the potential applications of LOIS are based on some kind of reachability (breadth first search) on an infinite graph.
- It is well known that local variables and recursive calls are usually implemented by using the *execution stack* (or *call stack*). We show that the same concept can also be used to implement pseudoparallel computation effectively. Thus, in a LOIS program, the execution stack contains not only local variables and return addresses, but also information about the currently running pseudoparallel threads.

7. Implementation

LOIS has been implemented as a C++ library, allowing the users to combine pseudoparallel computation with the full power of C++11. In Section 3, we omitted programming constructs such as recursion, function calls, expressions with side effects, complex data structures, etc., allowing us to present the semantics without delving into the irrelevant details. Such constructs are allowed in the actual implementation of LOIS. (See [21] for more details, and also, for a quick tutorial on the use of the LOIS library.) Below we roughly describe how LOIS is implemented. This is done mainly as a proof of concept, and to give an impression of the full potential of LOIS.

C++ allows a programming technique known as RAII, which allows automatic initialization and finalization of variables when a local variable enters or exits the scope. Our implementation uses RAII to change the contents of the LOIS stack when **if**, **for**, and **while** constructs are used.

For technical reasons, the syntax of our C++ library is a bit different than one presented in Section 6. Since **if** and **while** change the current context, we were unable to use C++'s `if` and `while` statements – instead, `If` and `While` macros are used, and `Ife` for if-then-else.

The type `lset` represents a LOIS set together with its inner context, and the polymorphic type `elem` represents a v-element. V-elements can represent an integer (type `int`), a term over $\mathcal{A}$ (type `term`), a pair (type `elpair`), a tuple (type `eltuple`), or a set (type `lset`). Integers, pairs, and tuples are implemented with the corresponding standard C++ types (`int`, `std::pair` and `std::vector`, respectively); and more types can be added by the programmer. So the programmer can for instance extend `elem` to allow a type representing lists or trees of elements, thus allowing sets of type `lset` to store infinite sets of lists or trees. It is well known that integers, pairs and tuples can be encoded in the set theory (using Kuratowski's definition of pair, for example); however, allowing to use them directly in our programs greatly improves both readability and efficiency.

Hybrid pseudoparallel looping over a set X of type `lset` is done with `for(elem x:X)`. This is implemented using the C++11 range-based loop. We can check the specific type of `x`, as well as inspect its components, with functions such as `is<T>`, `as<T>` (where `T` is one of the types listed above) and `isSet`, `asSet`. The syntactic sugar `lsetof<T>` is provided for defining sets which can only include elements of one specific type `T` – this allows static type checking, and eliminates the necessity of using `is` and `as` functions.

In some cases, such low-level representation of elements is not enough: for example, consider the function `extract(X)` which returns the only element of a set X of cardinality 1. If $X = \{a | a = b\} \cup \{0 | a \neq b\}$, then it is not possible to represent `extract(X)` as `elem` in the context $\{a \in \mathbb{A}, b \in \mathbb{A}\}$, since each `elem` has to be of specific type, and in our case, `extract(X)` can be either a term or an integer. In this case, we can use the type `lelem`, which represents *piecewise v-elements* – that is, ones which may have different representations depending on the constraints satisfied by variables in the context. Internally, this type is represented with a set – thus, `extract(X)` simply wraps the set X into a piecewise element. Type `lbool` represent a piecewise boolean variable, which boils down to a formula with free variables from its inner context.

All the conditions appearing in `If` and `While` statements are evaluated into first-order formulas over the underlying structure $\mathcal{A}$. A solver is used to check whether the set of all constraints on the stack is satisfiable (and thus whether to execute a statement or not). Also, a method of simplifying formulas is necessary to obtain readable presentations of results and to efficiently execute the sequel of the program.

The membership function `memberof(X,Y)`, as well as set equality `X==Y` and inclusion `subseteq(X,Y)`, have been implemented straightforwardly using `lbool` and hybrid pseudoparallel iteration over the sets involved. They are defined with a mutual recursion – set equality is a conjunction of two set inclusions, set inclusion $X \subseteq Y$ is evaluated by looping over all elements of X and checking whether they are members of Y, and membership $x \in Y$ is evaluated by looping over all elements of Y and checking whether they are equal to x. Equality and relation symbols applied to terms result in first order formulas.

Furthermore, for technical reasons, types `rbool`, `rset` and `relem` are used – these types are used for rvalues, while `lbool`, `lset` and `lelem` are used for lvalues. This is because lvalues have an inner context, while rvalues do not (their inner context always equals the current context).

To enforce fully pseudoparallel computation, thus simulating LOIS^0, write `for (relem e: fullypseudoparallel(X))`.

The underlying structure $\mathcal{A}$ is not given at the start of the program; instead, it is possible to define new sorts and new relations during the execution. Our prototype includes several relations with decidable theories (order, random partition, random graph, homo-

geneous tree), as well as solvers for these theories. Also, it allows consulting external SMT solvers.

The implementation of the LOIS library described above is available for the interested reader [20]. The prototype includes sample programs for testing various aspects of LOIS, as well as ones based on potential applications (minimizing automata), and ones showing the power of LOIS (reachability on the infinite random bipartite graph).

Performance The current implementation of LOIS is described as a proof of concept. Its performance can be improved by improving the used first order theory solvers, the techniques of simplification of formulas, and other aspects. The currently used external and internal SMT solvers, formula simplification techniques and performance tests are the topic of a separate publication [22]. The interested reader may confer [20], which contains several test cases.

8. Applications

This section serves as an illustration of the potential applications of LOIS to formal verification. We give many examples of classes of infinite-state systems known from formal verification, which can be naturally modeled using definable sets. We also show LOIS algorithms which can be used for solving various problems for those classes of systems, and present several termination proofs. As an important case, we distinguish ω-categorical structures.

Definable automata Fix an underlying logical structure $\mathcal{A}$. A *definable automaton* is defined just as a nondeterministic finite automaton (NFA), but all its components are required to be definable over $\mathcal{A}$, rather than finite – the statespace Q, the alphabet Σ, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial and final states $I, F \subseteq Q$. The automaton from Example 3 is a definable automaton over $(\mathbb{N}, +, \leq, 0)$, and also over $(\mathbb{N}, +)$, as $\leq$ and 0 are definable using $+$.

Definable automata can be presented as input for algorithms, by using the expressions which define them, and in LOIS, simply by using definable sets. As in automata theory, a central problem in verification to which many problems reduce is the *reachability problem*: does a given automaton have an accepting run?

Example 7. *Register automata* of Kaminsky and Francez [17] are (roughly) finite-state automata additionally equipped with finitely many registers which can store data values from an infinite set D, and which process sequences of data values from D. In each step, basing on the current state and equality or inequality tests among the values in the registers and the current input value, the automaton can choose to store the current value in one of its registers (replacing the previous value), change its state, or continue to the next input value. For example, we could consider a register automaton with two registers recognizing the set of those sequences $d_1 d_2 \dots d_n \in D^*$ such that $d_n \in \{d_1, d_2\}$. It is not difficult to prove [17] that the reachability problem for register automata is decidable (in fact, in PSPACE).

Register automata are a special case of definable automata, where the underlying structure $\mathcal{A}$ is $(D, =)$, or equivalently, $(\mathbb{N}, =)$. Indeed, if a register automaton has m states and n registers, then the corresponding definable automaton has statespace $Q = \{q_1, \dots, q_m\} \times \mathcal{A}^n$ ($q_1, \dots, q_m$ are treated as symbols), and input alphabet $\Sigma = D$. The transition relation $\delta \subseteq Q \times \Sigma \times Q$, as it is defined only using equalities and inequalities, is a definable set over $(D, =)$.

Example 8. *Rational relational automata* of Cerans (see [1]) are similar to register automata, but process sequences of rational numbers rather than elements of D, and can base their decisions on comparisons with respect to the linear order. These automata are a special case of definable automata, where the underlying structure is $(\mathbb{Q}, \leq)$, and the definitions involve quantifier-free formulas only. Rational register automata also have a decidable reachability problem, with the same complexity (PSPACE) as register automata.

Example 9. A *Minsky machine* is a model equipped with several counters storing natural numbers, which can be incremented, decremented, and tested for zero. The result of Minsky is that reachability for his machines is undecidable. Minsky machines are also a special case of definable automata, over the structure $(\mathbb{N}, +1)$.

Example 10. Consider the coverability problem for *Vector Addition Systems* (VASs, related to *Petri nets* – see e.g. [1]), defined below. Fix a dimension $k \geq 0$. A *VAS* of dimension k is described by a finite set of vectors $V \subseteq \mathbb{Z}^k$. We say that a vector t is *coverable* from s if, starting from s, one can reach a vector componentwise larger than t, by repeatedly adding vectors from V so that the intermediate result always stays in the non-negative fragment, $\mathbb{N}^k \subseteq \mathbb{Z}^k$. It is known that the coverability problem is decidable for VASs [14, 18].

A VAS gives rise to a definable automaton over the underlying structure $\mathcal{Z} = (\mathbb{Z}, <, 0)$, as follows. For a fixed finite set $V \subseteq \mathbb{Z}^k$ there is a formula ϕ_V with $2k$ free variables such that $\phi_V(u, w)$ holds iff $w \leq u + v$ for some $v \in V$; the formula ϕ_V refers to the relations $<, =$ only, and not to addition or subtraction. Therefore, a VAS induces an automaton with states $Q = \mathbb{N}^k$, transition relation

$$\delta = \{(u, w) \mid u, w \in W, \phi_V(u, w)\},$$

initial states $I = \{u \in W \mid u \leq s\}$ and accepting states $F = \{v \in W \mid v \geq t\}$; this automaton is definable over $\mathcal{Z}$. It is easy to see that this automaton has an accepting run iff t is coverable from s.

Example 11. *Timed automata* [2] and their variation *timed register automata* [8] can be seen as a special case of definable automata, with underlying structure $(\mathbb{R}, \leq, +1)$. The input alphabet is a *timed alphabet* the form $A = \{a_1, \dots, a_r\} \times \mathbb{R}$; a word over this alphabet is called a *timed word* if the timestamps (second coordinates) are increasing. A timed register automaton is a definable automaton over a timed alphabet, whose transition relation is defined by quantifier-free formulas involving $\leq$ and $+1$. Up to a minor encoding, timed automata can be seen as a special case of timed register automata Reachability of timed automata is decidable using the so-called *region construction* (see Example 14).

Fix an underlying structure $\mathcal{A}$. Every definable automaton can be constructed in LOIS, similarly as in Example 3. Therefore, for any definable automaton one can run the standard reachability algorithm described in Example 3. It is clear that the algorithm is correct, i.e., it will produce the right output, whenever it terminates. The aim is therefore to identify classes of definable automata for which the algorithm does terminate. This is a purely mathematical question: the programming part is done, since the reachability algorithm is executable in LOIS, assuming an SMT solver for the underlying structure is provided.

8.1 Proofs of Termination

We now present several termination arguments for the models listed above. As a side effect, we prove Theorem 13 and Theorem 14, i.e., decidability of the reachability problem for automata and pushdown automata which are definable over ω-*categorical* structures with decidable theory, generalizing known results.

Termination for ω-categorical structures We now present a generic termination argument for the reachability algorithm from Example 3, which works for all definable automata over a wide range of underlying structures $\mathcal{A}$, including register automata from Example 7, rational relational automata from Example 8, and many

others, generalizing slightly the results from [6, 7, 10]. This argument uses the notion of ω-*categoricity* from model theory.

For a structure $\mathcal{A}$, its *automorphism* is a bijection of $\mathcal{A}$ to itself, which preserves the sorts, relations and functions of $\mathcal{A}$. An automorphism π of $\mathcal{A}$ can be applied to a tuple $(a_1, \ldots, a_n)$ of elements of $\mathcal{A}$, yielding as a result the tuple $(\pi(a_1), \ldots, \pi(a_n))$; we say that two tuples $\bar{a}, \bar{b} \in \mathcal{A}^n$ are in the *same orbit* if there is an automorphism which maps $\bar{a}$ to $\bar{b}$. An orbit is an equivalence class of this equivalence relation. A countable structure $\mathcal{A}$ is ω-*categorical* if for every $n \in \mathbb{N}$, the set of tuples $\mathcal{A}^n$ has finitely many orbits.

Example 12. The structure $(\mathbb{N}, =)$ is ω-categorical. Its automorphisms are all bijections of $\mathbb{N}$ to itself. Representatives of the orbits of $\mathbb{N}^3$ are $(1,1,1), (1,1,2), (1,2,1), (2,1,1), (1,2,3)$. More generally, the orbits of $\mathbb{N}^k$ correspond to partitions of the set $\{1, \ldots, k\}$. The structure $(\mathbb{Q}, \leq)$ is also ω-categorical. Its automorphisms are the increasing bijections of $\mathbb{Q}$. Representatives of the orbits of $\mathbb{Q}^2$ are $(1,1), (1,2), (2,1)$. In general, the orbits of $\mathbb{Q}^k$ correspond to transitive, reflexive and total relations on $\{1, \ldots, k\}$. More generally, any *homogeneous* structure over a finite relational signature is ω-categorical. The structures $(\mathbb{Z}, \leq), (\mathbb{N}, +1), (\mathbb{R}, +1)$ are not ω-categorical: each pair $(0, n)$, for $n \in \mathbb{N}$, represents a different orbit. Very often ω-categorical structures have decidable theories [22].

Lemma 11. *Let $\mathcal{A}$ be an ω-categorical structure. Then any S-definable set V has only finitely many subsets which are definable with parameters from S.*

We consider the reachability algorithm described in Example 3, which is implemented in LOIS (see [20]).

Corollary 12. *The reachability algorithm terminates in LOIS^0 if the sets $E \subseteq V \times V$ and $I \subseteq V$ are definable over $\mathcal{A}$.*

Proof. Let $S \subseteq \mathcal{A}$ be a finite set such that E, V, I are S-definable. By Theorem 4, in the nth iteration of the loop in the `reach` function, the variable `R` evaluates to an S-definable set $R_n \subseteq V$, giving rise to an increasing sequence:

$$R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots. \tag{2}$$

By Lemma 11, the sequence (2) stabilizes after finitely many steps. □

Corollary 12 and Theorem 4 imply the following.

Theorem 13. *Reachability is decidable for definable automata over a fixed ω-categorical underlying structure with decidable theory.*

This result generalizes the decidability of the reachability problem for the models described in Examples 7 and 8, and by a slightly more detailed analysis which we omit here, yields optimal complexity bounds in each of those cases. Theorem 13 also generalizes slightly a result from [6, 7]. Note that the conciseness of the proof is thanks to the semantics of LOIS^0.

Other termination arguments Termination of the reachability algorithm can be also proved when the underlying structure is not ω-categorical, for some classes of definable automata.

Example 13. To test coverability of a VAS, convert it into a definable automaton as described in Example 10, and compute the reversal of this automaton in the standard wa, by swapping I with F and transposing the transition relation; this is again a definable automaton. The reachability algorithm from Example 3 terminates and correctly decides reachability of the reversed automaton, thus deciding coverability of the original VAS by using an SMT solver for $(\mathbb{N}, \leq)$. The termination argument is the well known (see [14]) well quasi-order argument: the sequence (2) stabilizes by Dickson's lemma, as it consists of upward-closed subsets of $\mathbb{N}^k$ (with respect to the coordinatewise ordering of $\mathbb{N}^k$). This termination argument can be generalized to other definable *well-structured transition systems* (cf. [14]).

Example 14. The reachability algorithm terminates for a fragment of timed register automata (see Example 11), called *constrained* timed register automata, which extend (up to a minor modification) timed automata (cf.[8]). A timed register automaton is *constrained* if its state space has finitely many orbits with respect to the automorphisms of the structure $(\mathbb{R}, \leq, +1)$. Therefore, the same argument as in the proof of Corollary 12 works to prove termination of the reachability algorithm for constrained timed register automata, generalizing the region construction for timed automata. To run this algorithm in LOIS, an SMT solver for $(\mathbb{R}, \leq, +1)$ is needed.

Note that by using LOIS, no specific data structures need to be devised: the algorithm is the standard one from Example 3, and is trivial to implement in LOIS (see [20]). Thanks to this, the termination proofs can be more abstract and precise, since they do not need to discuss the implementation details. This luxury of abstraction can be afforded thanks to the semantics of LOIS and to SMT solvers, which are heavily used when LOIS instructions are executed.

8.2 Other Verification Problems

So far we only investigated one problem: the reachability problem for definable automata. Many other classical algorithms involving finite-state systems can be executed in LOIS on infinite-state definable systems, often yielding interesting theoretical decidability results. We illustrate this phenomenon with two further examples: automata minimization and reachability of pushdown automata.

Example 15. Deterministic definable automata are defined just as finite deterministic automata, i.e., they are nondeterministic definable automata in which the transition relation is the graph of a function $\delta : Q \times \Sigma \to Q$. The automaton from Example 3 is a deterministic definable automaton. The standard Moore minimization algorithm presented in Example 3 can be run for any definable deterministic automaton; it is immediate to implement it in LOIS (see [20]). It follows from Corollary 12 that this algorithm terminates whenever the underlying structure is ω-categorical. The algorithm also terminates for many other automata, such as the one given in Example 3. It is shown in [9] that minimization terminates for deterministic timed automata, represented appropriately.

Example 16. Definable automata can be extended to *definable pushdown automata*, just as normal pushdown automata extend NFAs. Here, we extend the automata by ε-transitions, add to the syntax a definable stack alphabet Γ, and require the transition relation to be a definable subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \mathit{Ops} \times Q$, where $\mathit{Ops} = \{\mathit{push}, \mathit{pop}\} \times \Gamma$ is the set of stack operations. For the underlying structures $(\mathbb{N}, =)$, this model generalizes *pushdown register automata* [23].

We implement in LOIS a standard fixpoint algorithm for pushdown automata which computes the set of pairs (p, q) such that there is a run from state p to state q which starts and ends with an empty stack; this algorithm relies on the branching analogue of the reachability procedure from Example 3 implemented in Figure 4.

For infinite definable arguments over ω-categorical structures, the function `deduce` terminates for exactly the same reason as in the proof of Corollary 12, as a consequence of Lemma 11. Using this standard algorithm, executable in LOIS for infinite arguments, we prove the following theorem analogously to the proof of Theorem 13.

```
function deduce(rules) {
  set R = ∅;
  bool added = true;

  while (added) {
    added = false;
    for (X,x) in rules do
      if ((X⊆R) and (x∉R))
      {
          R += x;
          added |= true;
      }
  }
  return R;
}
```

Figure 4. The function `deduce` above takes as an argument a set of rules, each of the form (X, x), where X is a subset of some set F and $x \in F$, and computes the least $R \subseteq F$ such that $X \subseteq R$ implies $x \in R$ for every rule (X, x).

Theorem 14. *Reachability is decidable for all definable pushdown automata over a fixed ω-categorical underlying structure with decidable theory.*

An analogous result can be proved for *definable tree automata* with infinitely many states and infinite alphabet, or for infinite, *definable grammars*. Similar results were described in [5–7, 10, 11] (see Section 9).

All the above examples serve as a proof of concept, to demonstrate that infinite-state systems can be seamlessly manipulated using a programming language manipulating definable sets, which in turn relies on SMT solvers (we discuss this in Section 4). As a consequence, proofs of termination become more abstract and concise. Two paradigms are particularly useful for termination proofs: the ω-categoricity argument and the well quasi-ordering argument (see [14] for a broad overview), but other combinatorial arguments can be also employed.

By using LOIS, one can come up with many other algorithms for problems involving classes of infinite-state systems. LOIS acts as an intermediary which converts imperative programs into queries to SMT formulas. As far as we know, this provides a new use of SMT solvers in verification (we discuss this in [22]).

9. Related Work

The idea of a programming language which allows working with infinite sets, thus providing a useful tool in verification and in automata theory, comes from the papers [5] – which proposes a functional language called Nλ – and [10] – where an imperative language is proposed. Both languages are capable of handling infinite, but *orbit-finite sets with atoms*. These papers did not propose any efficient implementation. In particular, they propose to represent orbit-finite sets internally as unions of orbits. This has several drawbacks. Firstly, the underlying structure $\mathcal{A}$ needs be homogeneous and over a finite signature, whereas LOIS can work with any structure with decidable theory. For instance, the Example 1 concerning packings does not fit into the homogeneous setting. Secondly, the orbit decompositions proposed in those papers are exponentially less concise than descriptions by formulas, rendering them impractical for most applications. For example, the set $\mathcal{A}^n$ has exponentially (in n) many orbits, whereas in LOIS it is represented by the formula $\top$. On the other hand, for every set with orbit decomposition of size at most n, there is a representation as a definable set of size $O(n)$. This shows that the representation used by LOIS is more succinct, sometimes even exponentially. Lastly, SMT solvers cannot be (easily) employed to manipulate orbit decompositions.

Also, LOIS uses the novel pseudoparallel semantics, which differs from the *union semantics* of [10]: for the pseudocode below, X has final value A in the union semantics, and $\emptyset$ in LOIS (both extended by the obvious semantics of `-=` as removal of an element from the set); we think that the latter semantics is more natural.

```
X = A
for a in A do
  X -= a
```

Declarative programming paradigms offer some form of manipulation of infinite sets. In logic programming and constraint programming, predicates and constraints are typically infinite. In functional programming, the programmer manipulates functions as first-class objects. Furthermore, lazy evaluation allows performing operations on infinite streams. However, our approach is fundamentally different, as it represents sets internally by formulas, allowing to effectively scan through the set. In particular, membership and equality of sets can be effectively tested. On the other hand, we can only handle *definable* sets. Few computable functions $f : \mathbb{N} \to \mathbb{N}$ or streams have a definable graph. In fact, both approaches – functional programming and manipulating infinite sets – are orthogonal, and can be combined, as in Nλ proposed in [5]. We remark that [16] propose yet another, orthogonal extension of functional programming by the ability of testing equality between certain infinite sets, namely regular coinductive datatypes, and uses equation solvers for this purpose.

SETL [25] is a high-level imperative programming language, in which sets and a form of list comprehension are fundamental to the syntax and semantics. One of the objectives of SETL is to provide a high level of abstraction, simplifying the presentation of mathematical algorithms. This is also one of the main goals of LOIS. However, SETL does not allow manipulating infinite sets.

Superficially, LOIS is similar to Kaplan [19] – an extension of the Scala programming language. Its main purpose is to integrate constraint programming into imperative programming. It allows effective manipulation of constraints, and relies on a verification tool Leon, which in turn invokes the SMT solver Z3. Constraints are implemented as boolean valued functions (in Scala, functions are first-class objects) whose arguments are integers or algebraic data types built on top of integers. As such, they can be seen as certain logical formulas which can be defined as programs in a fragment of the Scala language. However, this fragment is incomparable with first order logic, as it allows recursion but not quantification. More importantly, the main objective of LOIS – to allow iterating over infinite sets – is not addressed in Kaplan (one can perform list comprehension in order to iterate through the explicit set of solutions of a constraint, which terminates only if this set is finite). It would be interesting to see whether iteration over infinite sets defined by constraints can be incorporated into Kaplan.

SMT solvers have been successfully applied in various branches of formal verification. In particular, in model checking they are used in predicate abstraction, interpolation-based model checking, backward reachability analysis and temporal induction. LOIS can be seen as yet another application of SMT solvers in model checking: in automata theory and theoretical verification.

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems, 1996.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science,

The University of Iowa, 2010. Available at www.SMT-LIB.org.

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885, 2009.

[5] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin, and Sławomir Lasota. Towards nominal computation. In John Field and Michael Hicks, editors, *POPL*, pages 401–412. ACM, 2012.

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

[7] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets. *Log. Meth. Comp. Sci.*, 10, 2014.

[8] Mikołaj Bojańczyk and Sławomir Lasota. A machine-independent characterization of timed languages. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, volume 7392 of *Lecture Notes in Computer Science*, pages 92–103. Springer, 2012.

[9] Mikołaj Bojańczyk and Sławomir Lasota. A machine-independent characterization of timed languages. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 92–103. Springer Berlin Heidelberg, 2012.

[10] Mikołaj Bojańczyk and Szymon Toruńczyk. Imperative programming in sets with atoms. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *FSTTCS*, volume 18 of *LIPIcs*, pages 4–15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[11] L. Clemente and S. Lasota. Reachability analysis of first-order definable pushdown systems. In *Proc. CSL'15*, pages 244–259, 2015.

[12] P. Erdős and A. Rényi. Asymmetric graphs. *Acta Mathematica Academiae Scientiarum Hungarica*, 14(3-4):295–315, 1963.

[13] William E. Fenton and Ed Dubinsky. *Introduction to discrete mathematics with ISETL*. Springer, 1996.

[14] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[15] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, New York, NY, USA, 1997.

[16] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Programming with coinductive types. Technical Report http://hdl.handle.net/1813/30798, Computing and Information Science, Cornell University, December 2012. Fundamenta Informaticae, to appear.

[17] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[18] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

[19] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 151–164, New York, NY, USA, 2012. ACM.

[20] Eryk Kopczyński and Szymon Toruńczyk. *LOIS* website. See http://www.mimuw.edu.pl/~erykk/lois .

[21] Eryk Kopczyński and Szymon Toruńczyk. *LOIS: technical documentation.* See http://www.mimuw.edu.pl/~erykk/lois .

[22] Eryk Kopczynski and Szymon Toruńczyk. *LOIS:* an application of SMT solvers. In Tim King and Ruzica Piskac, editors, *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories affiliated with the International Joint Conference on Automated Reasoning, SMT@IJCAR 2016, Coimbra, Portugal, July 1-2, 2016.*, volume 1617 of *CEUR Workshop Proceedings*, pages 51–60. CEUR-WS.org, 2016.

[23] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Reachability in pushdown register automata. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2014.

[24] R. Rado. Universal graphs and universal functions. *Acta Arithmetica*, 9:331–340, 1964.

[25] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with Sets; an Introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.